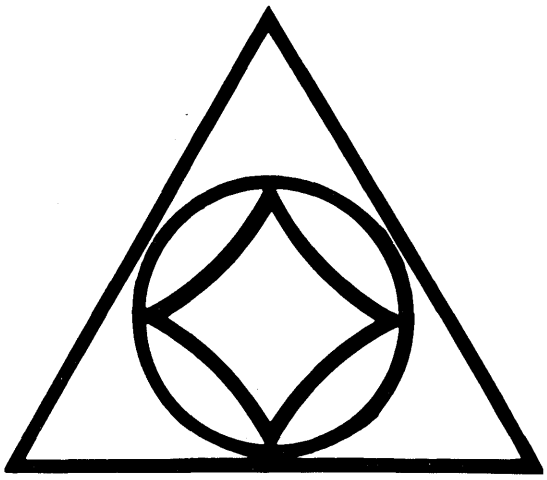


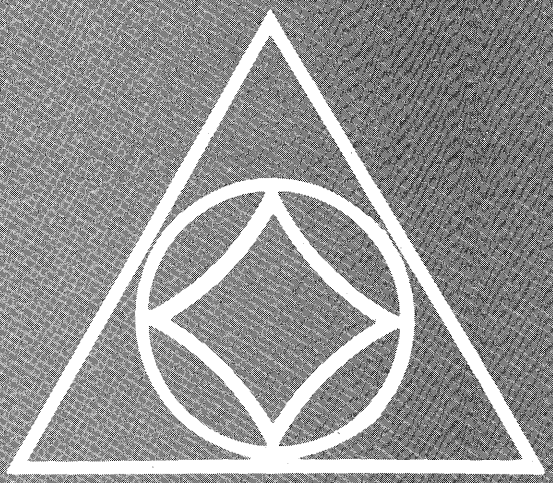
AFIPS

CONFERENCE PROCEEDINGS

VOLUME 37

1970

FALL JOINT COMPUTER CONFERENCE



AFIPS

CONFERENCE PROCEEDINGS

VOLUME 37

1970

FALL JOINT COMPUTER CONFERENCE

November 17-19, 1970
Houston, Texas

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1970 Fall Joint Computer Conference Committee or the American Federation of Information Processing Societies.

Library of Congress Catalog Card Number 55-44701

AFIPS PRESS
210 Summit Avenue
Montvale, New Jersey 07645

Printed in the United States of America

CONTENTS

The macro assembler, SWAP—A general purpose interpretive processor

by M. E. BARTON

Bell Telephone Laboratories
Naperville, Illinois

INTRODUCTION

A new macro assembler, the **SW**itching **A**ssembly **P**rogram (SWAP), provides a variety of new features and avoids the restrictions which are generally found in such programs. Most assemblers were not designed to be either general enough or powerful enough to accomplish tasks other than produce object code. SWAP may be used for a wide variety of other problems such as interpretively processing a language quite foreign to the assembler.

SWAP has been developed at Bell Telephone Laboratories, Incorporated, to assemble programs for three very different telephone switching processors. (SWAP is written in the IBM 360 assembly language and runs on the 360 with at least 256K bytes of memory.) With such varied object machines and the need to have all source decks translatable from the previously used assembler languages to the SWAP language, it is no wonder that the SWAP design includes many features not found in other assemblers. The cumulative set of features provides a powerful interpretive processor that may be used for a wide variety of problems.

DESCRIPTION

The source language is free field. Statement labels begin in column one. Operation names and parameters are delimited by a single comma or one or more blanks. Comments are preceded by the sharp sign (#), and the logical end of line is indicated by the semicolon (;) or physical end of card. A method is provided for user interpretation of other than this standard syntax; SWAP is currently being used as a preliminary version of several compilers.

Inputs

The SWAP assembler may receive its original input from a card, disc, or tape data set. The SOURCE pseudo-operation allows the programmer to change the input source at any point within a program. It is also capable of receiving input lines directly from another program, normally a source editor.

Outputs

Because the input line format is free field, the assembly listing of the source lines may appear quite unreadable. Therefore, the normal procedure is to have the assembler align all the fields of the printed line. The positions of the fields are, of course, a programmer option. There are several classes of statements that may be printed or suppressed at the programmer's discretion. In keeping everything as general as possible, it is natural that any operation, pseudo-operation, or macro may be assigned to any combination of these classes of statements.

In addition to producing the object program, which varies with different applications, and the assembly listing just described, SWAP has the facility to save symbol, instruction, or macro definitions in the form of libraries which may be loaded and used to assemble other programs.

Macro expansions and the results of text substitution functions are another optional output. The programmer completely controls which lines are to be generated and the format of these lines. These lines may be printed separately from the object listing or placed on card, disc, or tape storage. This optional output may be used to provide input to other assemblers,

and in this way SWAP can become a pseudo-compiler for almost any language. This output can also be used to produce preliminary program documents from comments which were originally placed in the source program deck.

Variables

There are numerous types of variable symbols, such as integers, floating point numbers, truth value, and character strings. The programmer may change or assign the type of any symbol as he wishes. For this purpose, the type of a symbol or operation is represented by a character. Each variable symbol may have up to 250 user-defined attributes which are data associated with each symbol. In addition, each symbol represents the top of a push-down list which allows the programmer to make a local use of any symbol.

A string variable would be defined by the TEXT pseudo-operation:

```
VOWELS  TEXT  'AEIOU'
```

while a numeric value is assigned by SET:

```
LIMIT  SET  10
```

The 'functional' notation is used extensively to represent not only the value of a symbol attribute, but also to represent array elements and predefined or user-defined arithmetic functions. In the following statement:

```
ALPHA(SA)  SET  BETA(SB)+10
```

the ALPHA attribute of symbol SA would be assigned a value ten greater than the BETA attribute of symbol SB.

An array of three dimensions would be declared by the statement:

```
ARRAY  CUBE(-1:1, 3, 0:2)=4
```

In this example, the range of the first dimension runs from -1 through $+1$, while the second dimension is from $+1$ through $+3$, and the third is from 0 through 2. Each element would have the initial value 4, but the following statement could be used to assign another value to a particular element of the array:

```
CUBE(-1, 2, 0)  SET  5
```

An attribute, array, or function reference may occur anywhere that a symbol may be used in an arithmetic expression.

Expressions

The following is a hierarchical list of the operators allowed in expressions:

**	or	↑	exponentiation
*	and	/	multiplication and division
unary−	and	unary¬	negation and complement
+	and	−	addition and subtraction
=, >, <, ¬=	or	≠	the six relational operators
=>	or	≥	
=<	or	≤	
&	and	¬	logical AND and AND of complement
|	and	!	logical OR and EXCLUSIVE OR

( ), [ ], and { } may be used in the usual manner to force evaluation in any order.

Four particular rules apply to the use of these operations:

1. Combined relations $A\rho B\rho C$ are evaluated the same as the expression $A\rho B\&B\rho C$ where ρ is any relational operator.
2. Character strings in comparisons are denoted as quoted strings.
3. The type of each operand is used to determine the method of evaluation. (For example, the complement of an integer is the 32-bit complement while the complement of a truth value is a 1-bit complement.)
4. If a TEXT symbol is encountered as an operand in an expression, it is called an indirect symbol, and its value is the result of evaluating the string as an expression.

Predefined Functions

Several built-in or predefined functions are provided to aid in evaluating some of the more common expressions. The following is a partial list of the available functions:

E(EXP)	Results in 2 raised to the EXP power.
MAX($EXP_1, \ldots, EXP_n$)	Returns the maximum of the expressions EXP_1 through EXP_n.

STYP(EXP, C)	Returns the value of EXP, but the type of the result is the character C as discussed in the *Variables* section.
SET(SYMB, EXP)	Returns the value of EXP and assigns that same value to the symbol SYMB. This differs from the SET pseudo-operation in that the symbol is defined during the evaluation of an expression.

Programmer-defined functions

To allow the programmer to define any number of new functions, the DFN pseudo-operation is provided. The general form of a function definition is written:

DFN $F(P_1, P_2, \ldots, P_n) = A_1 : B_1, A_2 : B_2, \ldots, A_n : B_n$

where F is the function name, the Ps are dummy parameter names, and the As and Bs are any valid expressions. These expressions may contain the Ps and other variables as well as other function calls which may be recursive.

To evaluate the function, the Bs are evaluated left to right. The result is the value of the A corresponding to the first B that has a value of true (or nonzero). The colons may be read as the word "if." A simple example would be the function:

DFN $\quad POS(X) = 1 : X > 0, 0 : X \leq 0$

which returns the value 1 if its argument is positive; otherwise, the result is zero. If the expression B_n is omitted, it is assumed to be true. Another example is the following definition of Ackermann's function:

DFN $ACK(M, N) = N+1 : M = 0, ACK(M-1, 1) :$

$N = 0, ACK(M-1, ACK(M, N-1))$

Two features are provided to allow an arbitrary number of arguments in the call of a function. The first is the ability to ask if an argument was implicitly omitted from the call. This feature is invoked by a question mark immediately following the dummy parameter name. If the argument was present, the result of the parameter-question mark is the value true; otherwise, the value is false. For example, the function defined by:

DFN $\quad INC(X, Y) = X+Y : Y?, X+1$

would yield the value 7 when called by $INC(2, 5)$ since Y is present, but the value of INC(3) is 4 since an argument value for Y was omitted.

The other feature which allows an arbitrary number of arguments is the ability to loop over a part of the defining expression, using successive argument values wherever the last dummy parameter name appears in the range of the loop. This feature is invoked by the appearance of an ellipsis (. . .) in the defining expression. The range of the loop is from the operator immediately preceding the ellipsis backward to the first occurrence of the same operator at the same level of parentheses. As an example, consider the following statement:

DFN $\quad SUM(X, Y) = A + \sqrt{X^{**}(Y+C)} + \cdots$

The range of the loop is from the + following the right parenthesis backward to the + between the A and the X. The call SUM(4, 1, 2, 3) would yield the same result as the following expression:

$$A+4^{**}(1+C)+4^{**}(2+C)+4^{**}(3+C)$$

The loop may also extend over the expression between two commas as the next example shows. A recursive function to do the EXCLUSIVE OR of an indefinite number of arguments could be defined by:

DFN $\quad XOR(A, B, C) = A \neg B \mid B \neg A : \neg C?,$

$XOR(XOR(A, B), \sqrt{C,} \ldots)$

Sequencing control

The pseudo-operations that allow the normal sequence of processing to be modified provide the real power of an assembler. In SWAP, the pseudo-operations that provide that control are JUMP and DO. JUMP forces the assembler to continue sequential processing with the indicated line, ignoring any intervening lines. The statement:

JUMP .LINE

will continue processing with the statement labeled: .LINE. The symbol .LINE is called a sequence symbol and is treated not as a normal symbol but only as the destination of a JUMP or DO. Sequence symbols are identified by the first character, which must be a period. A normal symbol may also be used as the destination of a JUMP or DO, if convenient. The destination of a JUMP may be either before or after the JUMP statement.

The JUMP is taken conditionally when an expression is used following the sequence symbol:

JUMP .XX, $INC > 10$ # IS IT OVER LIMIT

The JUMP to *.XX* will occur only if the value of the symbol *INC* is greater than ten.

The DO pseudo-operation is used to control an assembly time loop and may be written in one of three forms:

```
DO  .LOC, VAR=INIT, TEXP, INC     (i)
DO  .LOC, VAR=INIT, LIMIT, INC    (ii)
DO  .LOC, VAR=(LIST)              (iii)
```

Type (*i*) assigns the value of INIT to the variable symbol VAR. The truth value expression TEXP is then evaluated and, if the result is true, all the lines up to and including the line with .LOC in its location field are assembled. The value of INC (if INC is omitted, 1 is assumed) is then added to the value of VAR and the test is repeated using the incremented value of VAR.

Type (*ii*) is the same as type (*i*) except that the value of VAR is compared to the value of LIMIT; the loop is repeated if INC is positive and the value of VAR is less then or equal to the value of LIMIT. If INC is negative, the loop is repeated only while the value of VAR is greater than or equal to the value of LIMIT.

Type (*iii*) assigns to VAR the value of the first item in LIST. Succeeding values are used for each successive time around the loop until LIST is exhausted.

The following are examples of the use of DO:

```
Type (i)   DO  .Y, M=1, M≤10&A(M)>0
Type (ii)  DO  .X, K=1, 100, K+1
Type (iii) DO  .Z, N=(1, 3, 4, 7, 11, 13, 17)
```

Control of optional output

Selected results of macro and text substitution facilities may be used as an optional output. This is accomplished by the use of the EDIT psuedo-operation which may be used in a declarative, global, or range mode.

The declarative mode does not cause any output to be generated, but is used to declare the destination (printer, punch, or file) of the output and the method of handling long lines. It is also used to control the exceptions to the global output mode. For example, the statement:

```
PRINT  EDIT  OFF('ALL'),
       ON('REMARKS', NOTE, DOC),
                    CONT(72, 'X', '---')
```

would indicate that edited output is to be printed, and that any line that exceeds 72 characters is to be split into two print records with an *X* placed at the end of the first 72 characters and the remainder appended to the ---. If EDIT ON, the global form, were to be used with the above declarative, then only lines that contain NOTE or DOC in the operation field as well as all remark statements will be outputted.

The range form of EDIT allows a sequence of lines to be outputted regardless of their syntax. Lines outputted in this mode are then ignored by the remainder of the assembly processes.

Two examples of this form are EDIT .NEXT which causes the next line to be outputted, and EDIT .LINE which causes all lines up to, but not including, the line with the sequence symbol .LINE in its label field. See the Appendix for examples of the use of the EDIT pseudo-operation.

Macros

The macro facilities incorporated in SWAP make it one of the most flexible assemblers available. The macro facilities presented here are by no means exhaustive but only representative of the more commonly used features.

The general form of a macro definition is:

```
MACRO
prototype statement
macro text lines
MEND
```

The prototype statement contains the name of the macro definition as well as the dummy parameter names which are used in the definition. The macro text lines, a series of statements which make up the definition of the macro, will be reproduced whenever the macro is called.

Any operation, pseudo-operation, or macro may be redefined as a macro. Also, there are no restrictions as to which operations are used within a macro definition; this means that it is legitimate for macro definitions to be nested.

Macro operators and subarguments

Macro operators are provided to allow the programmer to obtain pertinent information about macro arguments and some of their common parts. A macro operator is indicated by its name character followed by a period and the dummy parameter name of the operand. For example, if a parameter named ARG has the value (*A, B, C*), then the number operator,

N.ARG, would be replaced by the number of subarguments of ARG; in this example, N.ARG is replaced by 3.

Any subparameter of a macro argument may be accessed by subscripting the parameter name with the number of the desired subargument. Additional levels of subarguments are obtained with the use of multiple indexes. As an example, let the parameter named ARG assume the value $P(Q, R(S, T))$, then:

ARG(0)	is replaced by P
ARG(1)	is replaced by Q
ARG(2)	is replaced by $R(S, T)$
ARG(2, 0)	is replaced by R
ARG(2, 1)	is replaced by S

The macro operators may be used on the results of each other as well as on subparameters; for example, N.ARG (2) would be replaced by 2.

The following is an example of a simple macro to define a list of symbols:

```
        MACRO
        DEFINE LIST
        DO .LP, K=1, N .LIST
LIST(K,1)   SET LIST(K, 2)
.LP         NULL # MARK END OF DO LOOP
        MEND
```

If the macro were called by the following line:
DEFINE ((SYMB, 5), (MATRIX (2), 7), (CC, 11))
it would expand to:

```
SYMB        SET  5
MATRIX(2)   SET  7
CC          SET 11
```

Macro functions

To provide more flexibility with the use of macros, several system parameters and macro functions have been made available. Macro functions are built-in functions that are replaced by a string of characters. This string, called the result, is determined by the particular function and its arguments. The arguments of a macro function may consist of macro parameters, other macro function calls, literal character strings, or symbolic variables. An example would be the DEC macro function, which has one argument, either a valid arithmetic or logical expression. The result is the decimal number equal to the value of the expression; the call DEC (7+8) would be replaced by 15.

Some of the major macro functions are:

1. IS(*expression, string*) is replaced by *string* if the value of *expression* is nonzero; otherwise, the result is the null string.
2. IFNOT(*string*) is replaced by *string* if the *expression* in the previously evaluated IS had a value of zero; otherwise, the result is null.
3. STR(exp_1, exp_2, *string*) is replaced by exp_2 characters starting with the exp_1 character of *string*.
4. MTXT(*tsym*) is replaced by the character string which is the value of the TEXT symbol *tsym*.
5. MTYP(*symb*) is replaced by the character that represents the type of the variable symbol *symb*.
6. MSUB(*string*) is replaced by the result of doing macro argument substitution on *string* a second time.
7. SYSLST(*exp*) is replaced by the *exp*th argument of the macro call.
8. MDO(*DO parameters; string*) is a horizontal DO loop where *string* is the range of the loop. Each time around, the loop produces the value of *string*, which is normally dependent on the DO variable symbol.

Keyword arguments

When the macro is called, keyword arguments are indicated by the parameter name followed by an equal sign and the argument string. An example would be the following calls of a MOVE macro:

```
MOVE  FROM=NEWDATA, TO=OLDDATA
                or
MOVE  TO=OLDDATA, FROM=NEWDATA
```

Both calls will yield the same expansions as the expansion of the MOVE macro using normal arguments:

```
MOVE  NEWDATA, OLDDATA
```

Default arguments

Default strings are used whenever an argument is omitted from a macro call. The default string is assigned on the macro prototype line by an equal sign and the desired default string after the dummy parameter name. Although the notation is the same, default arguments are completely independent of the use of keyword arguments.

Marco pseudo-operations

The ARGS pseudo-operation provides a method of declaring an auxiliary parameter list which supplements the parameter list declared on the prototype statement. These parameters may also be assigned default values.

The parameters defined on an ARGS line may be used anywhere a normal parameter may be used. The parameter values may be reset by the use of keyword arguments.

It is also possible for the programmer to reset his named macro argument values anywhere within a macro by using the MSET pseudo-operation. For example:

```
PARM MSET DEC(PARM)
```

would change the value of PARM to its decimal value.

The following is an example of the use of the ARGS pseudo-operation:

```
    MACRO
    FUN NUMBER
    ARGS WORD=(ONE, TWO, THREE)
#     NUMBER=WORD (NUMBER)
    MEND
```

When the macro is called by FUN 1+1, the following comment would be generated:

```
#  1+1=TWO
```

but the call FUN 1+1, WORD=(EIN, ZWEI, DREI) would generate:

```
#  1+1=ZWEI
```

Text manipulating facilities

Some of the more exotic features provided by SWAP are the character string pseudo-operations and the dollar macro call.

HUNT and SCAN pseudo-operations

The HUNT pseudo-operation allows the programmer to scan a string of characters for any break character in a second string. It will then define two TEXT symbols consisting of the portions of the string before and after the break character. For example, the statements:

```
BRKS  TEXT  '+-*/'
        .
        .
        .
      HUNT  .LOC, TOKEN, REMAIN,
              'LSIZE *ENTS', BRKS
```

will result in the symbols TOKEN and REMAIN having the string values of 'LSIZE' and '*ENTS' respectively. If one of the characters in BRKS could not be found in the scanned string, then a JUMP to the statement labeled .LOC would occur.

The SCAN pseudo-operation provides the extensive pattern matching facilities of SNOBOL3[1] along with success or failure transfer of control. This pseudo-operation is written:

```
SCAN TSYM P1...Pn  GOTO
```

where TSYM is a previously defined string valued variable. The SNOBOL3 notation is used to represent the pattern elements P_1 through P_n as well as the GOTO field. See the references for a more detailed presentation of these facilities.

Dollar functions

Dollar functions are very similar to macro functions in that the result of a dollar function call is a string of characters that replace the call. However, these functions may be used on input lines as well as in macros. The dollar functions provide the ability to call a one-line macro anywhere on a line by preceding the macro name with a dollar sign and following it with the argument list in parenthesis. For example, the macro:

```
                MACRO
                CHECK          A, B
IS(A<B, DEC(B-A) MORE)
                IFNOT (DEC(A-B) OVER)
                MEND
```

could be called by:

```
OP  X # $CHECK(X, 7)
```

For $X=4$, the line would appear in the assembly listing as:

```
OP  X # 3 MORE
```

and when X has the value 9, the line would appear as:

```
OP   X # 2 OVER
```

Special control

A special pseudo-operation has been provided to allow the programmer to ignore most of the SWAP syntax on input lines. The pseudo-operation is called UNIOP for universal operation, and it assigns the macro named in the variable field as the operation to be used for all succeeding lines. This means that regardless of what appears on a statement, that macro is called and may be used to decompose the line into meaningful SWAP statements. The following macro will make a simple test (i.e., the presence of an equal sign) to see if a line is a FORTRAN arithmetic statement and interpretively perform the assignment if it is; otherwise, it will call the macro named OTHER.

```
         MACRO
         ARITH
#        STRIP STATEMENT NUMBER
                  AND LOOK FOR EQUAL
                  SIGN
         HUNT .OTHER, SYMB, RMDR,
                  'STR(7, 64, SYSLIN)', '='
MTXT(SYMB)  SET STR(2, 62, MTXT(RMDR))
                  # PERFORM ASSIGNMENT
         JUMP  .OUT # TERMINATE
                  MACRO EXPANSION
.OTHER   OTHER 'SYSLIN' # NOT
                  ARITHMETIC STATEMENT
         MEND
```

The system macro parameter SYSLIN is replaced by the entire line of the macro call. The HUNT pseudo-operation will search for an equal sign and force a jump to the statement labeled .OTHER whenever the equal sign cannot be found. If UNIOP were initially set to the ARITH macro by the statement:

```
UNIOP      ARITH
```

then the line:

```
100  MTX(2, 3) = MTX(3, 2) + 1
```

would serve as a call to the ARITH macro which would then generate the following line:

```
MTX  (2, 3)    SET    MTX  (3, 2) + 1
```

Approximately 150 lines of SWAP macro definitions (see the Appendix) were used to build an interpreter of a FORTRAN like language. The following is a listing of a sample program and the printout that resulted from interpreting the program.

```
      DIMENSION KOUNT(10, 10)
C
700   FORMAT (3X, 10I4)
C
      DO     50     N=1, 10
      KOUNT(N, 1) =1
50    KOUNT(N, N) =1
C
      DO     100    N=3, 10
      DO     100    M=2, N-1
100   KOUNT(N, M) =KOUNT(N-1, M)
C                  +KOUNT(N-1, M-1)
      DO     200    N=1, 10
200   PRINT  700, (KOUNT(N, M), M=1, N)
C
      STOP
      END
```

```
 1
 1  1
 1  2  1
 1  3  3  1
 1  4  6  4  1
 1  5 10 10  5  1
 1  6 15 20 15  6  1
 1  7 21 35 35 21  7  1
 1  8 28 56 70 56 28  8  1
 1  9 36 84 126 126 84 36 9  1
```

CONCLUSION

The general design and implementation of the SWAP macro assembler has led to three things:

1. The job of writing a program in assembler language has been made easier. This is saving many man hours of programmer effort over the life of a project.
2. The development of intermediate level languages using macros has been made easier. This is aiding in the design of a true higher level language by clarifying the requirements of the new language.
3. The interpretive processing capabilities of the SWAP assembler have been used to solve a wide variety of problems. This is significantly reducing

the number of programs needed and, more importantly, reducing the programmer effort required to produce a given program.

ACKNOWLEDGMENTS

The author wishes to acknowledge the contribution of Messrs. R. E. Archer, A. J. Emrick, N. M. Haller, and E. Walton of Bell Telephone Laboratories, Incorporated, to the design and implementation of SWAP. The author would also like to thank Mr. D. E. Eastwood for his many suggestions and "philosophical arguments."

REFERENCES

1 D J FARBER R E GRISWOLD I P POLONSKY
SNOBOL, a string manipulation language
JACM Vol II No 1 pp 21-30 January 1964

2 D J FARBER R E GRISWOLD I P POLONSKY
The SNOBOL3 programming language
BSTJ Vol XLV No 6 pp 897-901 July 1966

3 M E BARTON N M HALLER G W RICKER
Service programs
BSTJ Vol 48 No 8 pp 2866-2880 October 1969

APPENDIX

A SWAP Program to Interpretively Process a FORTRAN Like Language.

```
SYSPRINT EDIT  OFF(EDIT,'ALL'),ON('MACROS')
FTYPES%  TEXT  'IX'            # FORMAT ITEM TYPES
BRKS%    TEXT  '/,H'"()'       #FORMAT BREAK CHARACTERS
FTYPTB%  TRPAT (X(7D),'Q'),('(','P'),(')','C'),(',','C'),('/',
               'S'),(X(7F),'Q'),(255) # TRANSLATE BREAKS TO
                                       ALPHABETICS
SQZ%     TRPAT (' ',0),(255)  # DELETES ALL BLANKS
EQ%      TEXT  '='            #EQUAL SIGN IS BREAK CHAR
DIMENSION OPSET ARRAY
STOP     OPSET END1
CONTINUE OPSET NULL
#
         MACRO # ALL UNDEFINED OPS ARE ASSUMED TO BE EQUATIONS
           NONOP
 HUNT .OUT V% E% 'MTR('STR(7,99,SYSLIN)',SQZ%)' EQ%   ## SQUEEZ
 OUT BLANKS
IS('MTYP(O.MTXT(V%))'='U', DFN MTXT(V%) MTXT(E%)) IFNOT(MTXT(V%)
 SET1 STR(2,99,MTXT(E%)))
         MEND
#
         MACRO
           GOTO  LOC,VAL=1
         JUMP    LOC(VAL)   ## ALSO TAKES CARE OF COMPUTED GOTOS
         MEND
#
         MACRO
           IF    COND,EQ,GT
TMP% TEXT 'MTR('COND',SQZ%)'
         SCAN    TMP%  *(E%)*  *LT%*   ## GET EXPRESSION
         JUMP    MTXT(LT%)    E%<0
         JUMP    EQ  E%=0
         JUMP    GT  E%>0
         MEND
#
         MACRO
           PRINT FMT
         DO    .X  K%=2,N.SYSLST    ## CHECK FOR ITERATIVE LISTS
IS('STR(1,1,SYSLST(K%))'='(', ITEM%) IFNOT(ITM%:DEC(K%)  TEXT)
 'SYSLST(K%)'
.X       NULL
FMT      OUT_        MDO(K%=2,N.SYSLST;MTXT(ITM%:DEC(K%)) )
         MEND
```

```
        MACRO
FMT       OUT_
K% SET 1;J% SET 0 ;JJ% SET 0
.LP     EDIT     .NEXT    ## GENERATE A LINE OF PRINTOUT
MSUB(MTXT(FMT:_:DEC(K%)))
 JUMP .LP,SET(K%,K%+1)≤FMT:_L  ## HAS FORMAT BEEN EXHAUSTED
        JUMP     .OUT,J%≥N.SYSLST|J%≤JJ%  ## WHEN PRINT LIST
 EXHAUSTED OR NOTHING BEING DONE
JJ%     SET    J%
.RLP    EDIT     .NEXT     ## BACK UP TO LAST LEFT PAREN
MSUB(STR(FMT:_K,500,MTXT(FMT:_:DEC(FMT:_R))))
        JUMP .RLP  SET(K%,FMT:_R+1)>FMT:_L&JJ%<J%<N.SYSLST
        JUMP     .LP,J%<N.SYSLST
        MEND
#
        MACRO
          ITEM% IT              ## PROCESS ITERATIVE PRINT LIST
        HUNT     .LST,VAR%,REM%,'S.Q.IT',EQ%
TMP     MSET     MTXT(VAR%)
VS      MSET     TMP(N.TMP)    ##ISOLATE LOOP INDEX
        MACRO
        FRMNDX VS=I.DEC(VS)
VLST%   TEXT    'R.TMP(1).TMP(N.TMP-1)'
        MEND     FRMNDX
        FRMNDX     ## REPLACE INDEX BY ITS VALUE
ITM%:DEC(K%)  TEXT 'MDO(VS:MTXT(REM%);MSUB(MTXT(VLST%)) )'
        JUMP    .OUT
.LST    NULL
ITM%:DEC(K%)  TEXT IT    ## IT WAS JUST AN EXPRESSION
        MEND
#
        MACRO
FMT       FORMAT LST
        EDIT     SAVE,OFF           ## STOP PRINTING LINES
        MEND     FORT_PROG    ## SUSPEND  PROGRAM DEFINITION
REM%    TEXT     'LST'
A% SET 0;%LINES SET 1;FMT:_R SET 1  ;FMT:_K SET 1
FMT     BRK_OUT                ## BUILD FORMAT DEFINITION
FMT:_L SET    %LINES
FMT:_:DEC(%LINES)  TEXT  'MDO(K%=1,A%;MTXT(ITM%:DEC(K%)))'
FORT_PROG  EXTEND       ## RESUME SOURCE PROGRAM DEFINITION
        EDIT     RESTORE      ## RESUME PRINTING LISTING
        MEND
#
        MACRO
FMT       BRK_OUT
.LP     HUNT     .OUT,TRM%,REM%,'STR(2,99,MTXT(REM%))',BRKS%
FMT     BRK_:MTR(REM%,FTYPTB%,1)   ## GO ON TRANSLATED BREAK
        JUMP     .LP
        MEND
```

```
        MACRO
          BRK_C                  ## COMMA OR RIGHT PAREN
        HUNT     .OUT,DUP%,TYP%,'MTXT(TRM%)',FTYPES%
        FTYP_:MTR(TYP%,FTYPTB%,1)
        MEND
#
        MACRO
FMT       BRK_P                  ## LEFT PAREN
FMT:_R SET     %LINES-1   ## SAVE POSITION FOR AUTO REPEAT
FMT:_K SET     1:MDO(K%=1,A%;+K.MTXT(ITM%:DEC(K%)))
        SCAN   REM%   *(SAVE%)*   *SV2%*      /F(.OUT)
BLMT%   SET     MAX(1,TRM%)    ## DUPLICATION FACTOR
        DO       .BK,B%=1,BLMT%
REM%    TEXT     'MTXT(SAVE%)'
.BK     BRK_OUT
REM%    TEXT     ',MTXT(SV2%)'
        MEND
#
        MACRO
FMT       BRK_S                  ## SLASH
        BRK_C
FMT:_:DEC(%LINES)   TEXT 'MDO(K%=1,A%;MTXT(ITM%:DEC(K%)))'
A%  SET  0 ;%LINES  SET  %LINES+1
        MEND
#
        MACRO
          BRK_Q                  ## QUOTED STRING
ITM%:DEC(SET(A%,A%+1))  TEXT   'Q.MTXT(REM%)'
REM%    TEXT     'STR(K.Q.MTXT(REM%)+2,99,MTXT(REM%))'
        MEND
#
        MACRO
          BRK_H                  ## HOLERITH STRING
ITM%:DEC(SET(A%,A%+1))  TEXT 'STR(2,TRM%,MTXT(REM%))'
REM%    TEXT     'STR(TRM%+1,99,MTXT(REM%))'
        MEND
#
        MACRO
          FTYP_I                 ## INTEGER
LN      MSET     STR(2,10,MTXT(TYP%))
DP      MSET    DEC(MAX(1,DUP%))
ITM%:DEC(SET(A%,A%+1))  TEXT   ':I.MDO(%N=1,MIN(DP,I.N.I.SYSLST-
I.DEC(J%));I.DEC(I.SYSLST(SET(J%,J%+1)),LN,' '))'
        MEND
#
        MACRO
          FTYP_X                 ## BLANKS
ITM%:DEC(SET(A%,A%+1))  TEXT 'MDO(N%=1,MAX(1,DUP%); )'
        MEND
```

```
        MACRO
          END
SYSPRINT EDIT OFF              ## TERMINATE SOURCE LISTING
        MEND    FORT_PROG          ## END OF SOURCE PROGRAM
        FORT_PROG             ## NOW EXECUTE SOURCE PROGRAM
        END1                     ## TERMINATE RUN
        MEND
#
FORMAT    OPBITS ON(ACTIVE)    # ALLOW THESE OPS TO EXPAND
                                DURING MACRO DEFINITION
END       OPBITS ON(ACTIVE)
END       OPBITS OFF(CONT)     # NO CONTINUATION ALLOWED FOR END
                                MACRO
EDIT      OPBITS ON(ACTIVE)
          EDIT  ON(FORMAT,END)
#
        MACRO          # MAKE ENTIRE PROGRAM A MACRO DEFINITION
          FORT_PROG
SYSPRINT EDIT  .NEXT           ## EJECT TO NEW PAGE
1
PRINT     EDIT  ON             ## PRODUCE SOURCE LISTING
```

Definition mechanisms in extensible programming languages

by STEPHEN A. SCHUMAN*

Université de Grenoble
Grenoble, France

and

PHILIPPE JORRAND

Centre Scientifique IBM-France
Grenoble, France

INTRODUCTION

The development of extensible programming languages is currently an extremely active area of research, and one which is considered very promising by a broad segment of the computing community. This paper represents an attempt at unification and generalization of these developments, reflecting a specific perspective on their present direction of evolution. The principal influences on this work are cited in the bibliography, and the text itself is devoid of references. This is indicative of the recurring difficulty of attributing the basic ideas in this area to any single source; from the start, the development effort has been characterized by an exceptional interchange of ideas.

One simple premise underlies the proposals for an extensible programming language: that a "user" should be capable of modifying the definition of that language, in order to define for himself the particular language which corresponds to his needs. While there is, for the moment, a certain disagreement as to the degree of "sophistication" which can reasonably be attributed to such a user, there is also a growing realization that the time is past when it is sufficient to confront him with a complex and inflexible language on a "take it or leave it" basis.

According to the current conception, an extensible language is composed of two essential elements:

1. A *base language,* encompassing a set of indispensable programming primitives, organized so as to constitute, in themselves, a coherent language.
2. A set of *extension mechanisms,* establishing a systematic framework for defining new linguistic constructions in terms of already existing ones.

Within this frame of reference, an *extended language* is that language which is defined by some specific set of extensions to the given base language. In practice, definitions can be pyramided, using a particular extended language as the new point of departure. Implicit in this approach is the assumption that the processing of any extended language program involves its systematic reduction into an equivalent program, expressed entirely in terms of the base language.

Following a useful if arbitrary convention, the extension mechanisms are generally categorized as either *semantic* or *syntactic,* depending on the capabilities that they provide. These two types of extensibility are the subjects of subsequent sections, where models are developed for these mechanisms.

Motivations for extensible languages

The primary impetus behind the development of extensible languages has been the need to resolve what has become a classic conflict of goals in programming language design. The problem can be formulated as

* Present address: Centre Scientifique IBM—France

power of expression versus *economy of concepts*. Power of expression encompasses both "how much can be expressed" and "how easy it is to express". It is essentially a question of the effectiveness of the language, as seen from the viewpoint of the user. Economy of concepts refers to the idea that a language should embody the "smallest possible number" of distinguishable concepts, each one existing at the "lowest possible level". This point of view, which can be identified with the implementer, is based on efficiency considerations, and is supported by a simple economic fact: the costs of producing and/or using a compiler can become prohibitive. Since it is wholly impractical to totally disregard either of these competitive claims, a language designer is generally faced with the futile task of reconciling two equally important but mutually exclusive objectives within a single language.

Extensible languages constitute an extremely pragmatic response to this problem, in the sense that they represent a means of avoiding, rather than overcoming this dilemma. In essence, this approach seeks to encourage rather than to suppress the proliferation of programming languages; this reflects an increasing disillusionment with the "universal language" concept, especially in light of the need to vastly expand the domain of application for programming languages in general. The purpose of extensible languages is to establish an orderly framework capable of accommodating the development of numerous different, and possibly quite distinctive dialects.

In an extensible language, the criteria concerning economy of concepts are imposed at the point of formulating the primitives which comprise the base language. This remains, therefore, the responsibility of the implementer. Moreover, he is the one who determines the nature of the extension mechanisms to be provided. This acts to constrain the properties of the extended languages subsequently defined, and to effectively control the consistency and efficiency of the corresponding compilers.

The specific decisions affecting power of expression, however, are left entirely to the discretion of the user, subject only to the restrictions inherent in the extension mechanisms at his disposal. This additional "degree of freedom" seems appropriate, in that it is after all the language user who is most immediately affected by these decisions, and thus, most competent to make the determination. The choices will, in general, depend on both the particular application area as well as various highly subjective criteria. What is important is that the decision may be made independently for each individual extended language.

At the same time, the extensible language approach overcomes what has heretofore been the principal obstacle to a diversity of programming languages: incompatibility among programs written in different languages. The solution follows automatically from the fact that each dialect is translated into a common base language, and that this translation is effected by essentially the same processor.

Despite the intention of facilitating the definition of diverse languages, the extensible language framework is particularly appropriate for addressing such significant problems as machine-to-machine transferability, language and compiler standardization, and object code optimization. The problems remain within manageable limits, independent of the number of different dialects; they need only be resolved within the restricted scope of the base language and the associated extension mechanisms.

Evolution of extensible languages

An extensible language, according to the original conception, was a high level language whose compiler permitted certain "perturbations" to be defined. Semantic extension was formulated as a more flexible set of data and procedure declarations, while syntactic extension was confined to integrating the entities which could be declared into a pre-established style of expression. For the most part, the currently existing extensible languages reflect this point of departure.

It is nonetheless true that the basic research underlying the development of extensible languages has taken on the character of an attempt to isolate and generalize the various "component parts" of programming languages, with the objective of introducing the property of "systematic variability". A consequence of this effort has been the gradual emergence of a somewhat more abstract view of extensible languages, wherein the base language is construed as an inventory of essential primitives, the syntax of which minimally organizes these elements into a coherent language. Semantic extension is considered as a set of "constructors" serving to generate new, but completely compatible primitives; syntactic extension permits the definition of the specific structural combinations of these primitives which are actually meaningful. Thus, extensible languages have progressively assumed the aspect of a language definition framework, one which has the unique property that an operational compiler exists at each point in the definitional process.

Accordingly, it is increasingly appropriate to regard extensible languages as the basis for a practical language definition system, irrespective of who has responsibility for language development. Potentially, such an environment is applicable even to the definition of non-

extensible languages. Heretofore, it has been implied that any given extended language was itself fully extensible, since its definition is simply the result of successive levels of extension. In conjunction with the progressive generalization of the extension capabilities, however, one is naturally led to envision a complementary set of *restriction mechanisms*, which would serve to selectively disable the corresponding extension mechanisms.

The intended function of the restriction mechanisms is to eliminate the inevitable overhead associated with the ability to accommodate arbitrary extension. They would be employed at the point where a particular dialect is to be "frozen". In effect, such restriction mechanisms represent a means of imposing constraints on subsequent extensions to the defined language (even to the extent of excluding them entirely), in exchange for a proportional increase in the efficiency of the translator. The advantage of this approach is obvious: the end result of such a development process is *both* a coherent definition of the language *and* an efficient, operational compiler.

Within this expanded frame of reference, most of the extensible languages covered by the current literature might more properly be considered as extended languages, even though they were not defined by means of extension. This is not unexpected, since they represent the results of the initial phase of development. The remainder of this paper is devoted to a discussion of the types of extension mechanisms appropriate to this more evolved interpretation of extensible languages. The subject of the next section is semantic extensibility, while the final section is concerned with syntactic extensibility. These two capabilities form a sort of two-dimensional definition space, within which new programming languages may be created by means of extension.

SEMANTIC EXTENSIBILITY

In order to discuss semantic extensibility, it is first necessary to establish what is meant here by the *semantics* of a programming language. A program remains an inert piece of text until such time as it is submitted to some processor: in the current context, a computer controlled by a compiler for the language in which the program is expressed. The activity of the processor can be broadly characterized by the following steps:

1. Recognition of a unit of text.
2. Elaboration of a *meaning* for that unit.
3. Invocation of the *actions* implied by that meaning.

According to the second of these steps, the notion of meaning may be interpreted as the *link* between the units of text and the corresponding actions. The set of such links will be taken to represent the semantics of the programming language.

As an example, the sequence of characters "3.14159" is, in most languages, a legitimate unit of text. The elaboration of its associated meaning might establish the following set of assertions:

—this unit represents an object which is a value.
—that value has a type, which is *real*.
—the internal format of that value is floating-point.
—the object will reside in a table of constants.

This being established, the actions causing the construction and allocation of the object may be invoked. The set of assertions forms the link between the text and the actions; it represents the "meaning" of 3.14159.

With respect to the processor, the definition of the semantics of a language may be considered to exist in the form of a *description* of these links for each object in the domain of the language. When a programming language incorporates functions which permit explicit modification of these descriptions, then that language possesses the property of *semantic extensibility*. These functions, referred to as *semantic extension mechanisms*, serve to introduce new kinds of objects into the language, essentially by defining the set of linkages to be attributed to the external representation of those objects.

Semantic extension in the domain of values: A model

The objects involved in the processing of a program belong, in general, to a variety of categories, each of which constitutes a potential domain for semantic extension. The values, in the conventional sense, obviously form one such category. In order to illustrate the overall concept of semantic extensibility, a model for one specific mechanism of semantic extension will be formulated here. It operates on a description of a particular category of objects, which encompasses a generalization of the usual notion of value. For example, procedures, structures and pointers are also considered as values, in addition to simple integers, booleans, etc.

These values are divided into classes, where each class is characterized by a *mode*. The mode constitutes the description of all of the values belonging to that class. Thus the mode of a value may be thought of as playing a role analogous to that of a data-type. It is

assumed that processing of a program is controlled by syntactic analysis. Once a unit of text has been isolated, the active set of modes is used by the compiler to elaborate its meaning. Typically, modes are utilized to make sure that a value is employed correctly, to verify that expressions are consistent, to effect the selection of operations and to decide where conversions are required.

The principal component of the semantic extension mechanism is a function which permits the definition of new modes. Once a mode has been defined, the values belonging to the class characterized by that mode may be used in the same general ways as other values. That is to say, those values can be stored into variables, passed as parameters, returned as results of functions, etc.

The mode definition function would be used like a declaration in the base language. The following notation will be taken as a model for the call on this function:

mode σ *is* τ *with* π;

The three components of the definition are:

1. the symbol clause "*mode* σ",
2. the type clause "*is* τ",
3. the property clause "*with* π".

The property clause may be omitted.

The symbol clause

In the symbol clause, a new symbol σ is declared as the name of the mode whose description is specified by the other clauses. For example,

mode complex is . . .

may be used to introduce the symbol *complex*. In addition, the mode to be defined may depend on formal parameters, which are specified in the symbol clause as follows:

mode matrix (*int* m, *int* n) *is* . . .

The actual parameters would presumably be supplied when the symbol is used in a declarative context, such as

matrix (4, 5)A;

The type clause

In the type clause, τ specifies the nature of the values characterized by the mode being defined. Thus, τ is either the name of an existing mode or a *constructor* applied to some combination of previously defined modes. There are assumed to be a finite number of modes predefined within the base language. In the scope of this paper, *int*, *real*, *bool* and *char* are taken to be the symbols representing four of these basic modes, standing for the modes of integer, real, boolean and single character values, respectively. Thus, a valid mode definition might be:

mode integer is int;

The model presented here also includes a set of *mode constructors*, which act to create new modes from existing ones. The following list of constructors indicates the kinds of combinations envisioned:

1. Pointers
   A pointer is a value designating another value. As any value, a pointer has a mode, which indicates that:

   —it is a pointer.
   —it is able to point to values of a specified class.

   The notation *ptr* M creates the mode characterizing pointers to values of mode M. For example,

   mode ppr is ptr ptr real;

   specifies that values of mode *ppr* are pointers to pointers to reals.

2. Procedures
   A procedure is a value, implying that one can actually have procedure variables, pass procedures as parameters and return them as the results of other procedures. Being a value, a procedure has a mode which indicates that:

   —it is a procedure.
   —it takes a fixed number (possibly zero) of parameters, of specified modes.
   —it returns a result of a given mode, or it does not return any result.

   The notation *proc* (M_1, . . . , M_n) M forms the mode of a procedure taking n parameters, of modes $M_1 \ldots M_n$ respectively, and returning a value of mode M. As an example, one could declare

   mode trigo is proc (real)real;

for the class of trigonometric functions, and then

mode trigocompose is proc (trigo, trigo)trigo;

for the mode of functions taking two trigonometric functions as parameters, and delivering a third one (which could be the composition of the first two) as the result.

3. Aggregates
Two kinds of aggregates will be described:

a. the *tuples*, which have a constant number of components, possibly of different modes;
b. the *sequences*, which have a possibly variable number of components, but of identical modes.
   a. Tuples
   The mode of a tuple having n components is established by the notation [M_1s_1, . . . , M_ns_n], where $M_1 \ldots M_n$ are the modes of the respective components, and $s_1 \ldots s_n$ are the names of these components, which serve as selectors. Thus, the definition of the mode *complex* might be written.

   mode complex is [*real* rp, *real* ip];

   If Z is a complex value, one might write Z.rp or Z.ip to access either the real part or the imaginary part of Z.
   b. Sequences
   The mode of a sequence is constructed by the notation *seq* (e)M, where e stands for an expression producing an integer value, which fixes the length of the sequence; the parenthesized expression may be omitted, in which case the length is variable. The components, each having mode M, are indexed by integer values ranging from 1 to the current length, inclusively. The mode for real square matrices could be defined as follows:

   mode rsqmatrix (*int* n)

   is seq (n) *seq* (n) *real;*

   If B is a real square matrix, then the notation B(i)(j) would provide access to an individual component.

4. Union
The union constructor introduces a mode characterizing values belonging to one of a specified list of classes. The notation *union* ($M_1, \ldots, M_n$) produces a mode for values having any one of the modes $M_1 \ldots M_n$. Thus, if one defines

mode procir is proc (*union* (*int, real*));

this mode describes procedures taking one parameter, which may be either an integer or a real, and returning no result. A further example, using the tuple, pointer, sequence and union constructors, shows the possibility of recursive definition:

mode tree

is [*char* root,

seq ptr union (*char, tree*) branch];

The list of mode constructors given above is intended to be indicative but not exhaustive. Moreover, it must be emphasized that the constructors themselves are essentially independent of the nature and number of the basic modes. Consequently, one could readily admit the use of such constructors with an entirely different set of primitive modes (e.g., one which more closely reflects the representations on an actual machine). What is essential is that the new modes generated by these constructors must be usable in the language in the same ways as the original ones.

The property clause

The property clause "*with* π" when present, specifies a list of properties possessed by the values of the mode being defined. These properties identify a sub-class of the values characterized by the mode in the type clause. Two kinds of properties are introduced for the present model: *transforms* and *selectors.*

1. Transforms
The transforms provide a means of specifying the actions to be taken when a value of mode M_1 occurs in a context where a value of mode M_2 is required ($M_1 \neq M_2$). If M is the mode being declared, then two notations may be used to indicate a transform:

   a. "*from* M_1 *by* $V.E_1$," which specifies that a value of mode M may be produced from a value of mode M_1 (identified by the bound variable V) by evaluating the expression E_1.

b. "*into* M_2 *by* $V.E_2$," which specifies that a value of mode M_2 may be produced from a value of mode M (identified by the bound variable V) by evaluating the expression E_2. The following definitions provide an illustration:

```
mode complex
  is [real rp, real ip]
  with from real
       by x. [x, 0.0],
     into real
       by y. (if y.ip=0
            then y.rp
            else error);

mode imaginary
  is [real ip]
  with from complex
       by x. (if x.rp=0
            then [x.ip]
            else error),
     into complex
       by y. [0.0, y.ip];
```

By the transforms in the above definitions, all of the natural conversions among real, complex, and imaginary values are provided. It must be noted that the system of transformations specified among the modes may be represented by a directed graph, where the nodes correspond to the modes, and the arcs are established by the *from* and *into* transforms. Thus, the rule to decide whether the transformation from M_1 into M_2 is known might be formulated as follows:

i. There must exist at least one path from M_1 to M_2.
ii. If there are several paths, there must exist one which is shorter than all of the others.
iii. That path represents the desired transformation.

2. Selectors

The notation "*take* M_1s *as* V.E" may appear in the list of properties attached to the definition of the mode M. It serves to establish the name "s" as an additional selector which may be applied to values of mode M to produce a value of mode M_1. Thus, if X is a value of mode M, then the effect of writing "X.s" is to evaluate the expression E (within which V acts as a bound variable identifying the value X) and to transform its result into a value of mode M_1. As an example, the definition of *complex* might be augmented by attaching the following two properties:

take real mag *as* Z. (sqrt (Z. rp ↑ 2+Z.ip ↑ 2)),

take radian ang *as* Z. (arctan (Z.ip/Z.rp));

The mode *radian* is presumed to be defined elsewhere, and to properly characterize the class of angular values.

As with the case of the constructors, the properties presented here are intended to suggest the kinds of facilities which are appropriate within the framework established by the concept of mode.

In summary, it must be stressed that the model developed here is applicable only to one particular category of objects, namely the values on which a program operates. Clearly, there exist other identifiable categories which enter into the processing of a program (e.g., control structure, environment resources, etc.). It is equally appropriate to regard these as potential domains for semantic extensibility. This will no doubt necessitate the introduction of additional extension mechanisms, following the general approach established here. As the number of mechanisms is expanded, the possibility for selective restriction of the extension capabilities will become increasingly important. The development of the corresponding semantic restriction mechanisms is imperative, for they are essential to the production of specialized compilers for languages defined by means of extension.

SYNTACTIC EXTENSIBILITY

A language incorporating functions which permit a user to introduce explicit modifications to the *syntax* of that language is said to possess the property of *syntactic extensibility*. The purpose of this section is to establish the nature of such a facility. It is primarily devoted to the development of a model which will serve to characterize the mechanism of syntactic extension, and permit exploration of its definitional range.

It must be made explicit that, when speaking of modifications to the syntax of a language, one is in fact talking about actual alterations to the *grammar* which serves to define that syntax. For a conventional language, the grammar is essentially static. Thus, it is conceivable that a programmer could be wholly unaware of its existence. The syntactic rules, which he is nonetheless constrained to observe (whether he likes them or not), are the same each time he writes a pro-

gram in that language, and no deviation is permitted anywhere in the scope of the program. The situation is significantly different for the case of a syntactically extensible language. This capability is provided by means of a set of functions, properly imbedded in the language, which acts to change the grammar. Provided that the user is cognizant of these functions and their grammatical domain, he then has the option of effecting perhaps quite substantial modifications to the syntax of that language *during the course of writing a program in that language;* this is in parallel with whatever semantic extensions he might introduce. In effect, the grammar itself becomes subject to dynamic variation, and the actual syntax of the language becomes dependent on the program being processed.

The syntactic macro mechanism: A model

The basis of most existing proposals for achieving syntactic extensibility is what has come to be called the *syntactic macro mechanism.* A model of this mechanism is introduced at this point in order to illustrate the possibilities of syntactic extension. The model is based on the assumption that the syntactic component of the base language, and by induction any subsequent extended language, can be effectively defined by a *context-free grammar* (or the equivalent BNF representation). This relatively simple formalism is adopted as the underlying definitional system despite an obvious contradiction which is present: a grammar which is subject to dynamic redefinition by constructs in the language whose syntax it defines is certainly not "context-free" in the strict sense. Therefore, it is only the instantaneous syntactic definition of the language which is considered within the context-free framework.

The most essential element of the syntactic macro mechanism is the function which establishes the definition of a syntactic macro. It must be a legitimate linguistic construct of the base language proper, and its format would likely resemble any other declaration in that language. The following representation will be used to model a call on this function:

macro ϕ *where* π *means* ρ;

The respective components are:

ϕ, the *production*;

π, the *predicate*; and

ρ, the *replacement.*

The *macro* clause would be written in the form

macro C→'phrase'

where C is the name of a category (non-terminal) symbol in the grammar, and the phrase is an ordered sequence, $S_1 \ldots S_n$, such that each constituent is the name of a category or terminal symbol. Thus the production in a *macro* clause corresponds directly to a context-free production. The *where* and *means* clauses are optional components of the definition, and will be discussed below.

A syntactic macro definition differs from an ordinary declaration in the base language in the sense that it is a function operating directly on the grammar, and takes effect immediately. In essence, it incorporates the specified production into the grammar. Subsequent to the occurrence of such a definition in a program, syntactic configurations conforming to the structure of the phrase are acceptable wherever the corresponding category is syntactically valid. This will apply until such time as that definition is, in some way, disabled. As an example, one might include a syntactic macro definition starting with

macro FACT→'PRIM !'

for the purpose of introducing the factorial notation into the syntax for arithmetic expressions. Within the scope of that definition, the effect would be the same as if the syntactic definition of the language (represented in BNF) incorporated an additional alternative

⟨factor⟩::= ... | ⟨primary⟩!

Thus, in principle, a statement of the form

c := n!/((n−m)! * m!);

might become syntactically valid according to the active set of definitions.

The production

The role of the production is to establish both the context and the format in which "calls" on that macro may be written. The category name on the left controls where, within the syntactic framework, such calls are permitted. One may potentially designate any category which is referenced by the active set of productions. The phrase indicates the exact syntactic configuration which is to be interpreted as a call on that particular macro. In general, one may specify any arbitrary sequence (possibly empty) of symbol names. The constituents may be existing symbols, terminals which were not previously present, or categories to be defined in other macros. This is of course, subject to the constraint that the grammar as a whole must remain both

well-formed and non-ambiguous, if it is to fulfill its intended function.

In addition, the *macro* clause serves to declare a set of formal parameters, which may be referenced elsewhere in the definition. Each separate call on that macro can be thought of as establishing a local syntactic context, defined with respect to the complete syntax tree which structurally describes the program. This context would be relative to the position of the node corresponding to the specified category, and would include the immediate descendants of that node, corresponding to the constituents of the phrase. At a call, the symbol names appearing in the production are associated with the actual nodes occurring in that context. Thus, the terminal names represent an individual instance of that terminal, and the category names represent some complete syntactic sub-tree belonging to that category. In order to distinguish between different formal parameters having the same name, the convention of subscripting the names will be adopted here; this notation could readily be replaced by declaration of unique identifiers.

The replacement

The *means* clause specifies the syntactic structure which constitutes the replacement for a call on that particular macro. It is written in the form

means 'string'

where the string is an ordered sequence, composed of either formal parameters or terminal symbol names. An instance of this string is generated in place of every call on that macro, within which the actual structure represented by a formal parameter is substituted for every occurrence of its name. If the complete syntactic macro definition for the factorial operator had been

macro $FACT_0 \rightarrow$'$PRIM_1$!'

means 'factorial ($PRIM_1$)';

then each call on this macro would simply be replaced by a call on the procedure named "factorial", assumed to be defined elsewhere.

When present, the means clause establishes the semantic interpretation to be associated with the corresponding production; if absent, then presumably the construct is only an intermediate form, whose interpretation is subsumed in some larger context. The "meaning," however, is given as the expansion of that construct into a "logically lower level language". While the replacement may be expressed in terms of calls on other syntactic macros, these will also be expanded. In effect, the meaning of every new construct introduced into the language is defined by specifying its systematic expansion into the base language. Accordingly, one might consider syntactic extension merely as a means for permitting a set of "systematic abbreviations" to be defined "on top of" the base language.

An important consequence of the fact that a syntactic macro definition is itself a valid element of the base language is that such definitions may occur in the context of a replacement. This is illustrated by the following example, showing how a declaration for "push-down stack" might be introduced:

```
macro DECL0→'TYPE1 stack [EXPR1] IDEN1;'
    means     'TYPE1 array [1:EXPR1] IDEN1;
              integer level_IDEN1 initial 0;
              macro PRIM0→'depth_IDEN1'
                  means 'res (EXPR1)';
              macro PRIM1→'IDEN1'
                  means '(if level_IDEN1 >0
                      then
                      (IDEN1 [level_IDEN1],
                      level_IDEN1:=
                        level_IDEN1−1;)
                      else
                      error ("overdraw
                        IDEN1"))';
              macro REFR0→'IDEN1'
                  means '(if level_IDEN1 <
                      depth_IDEN1
                      then
                      (level_IDEN1:=
                        level_IDEN1+1;
                      IDEN1 [level_IDEN1])
                      else
                      error ("overflow
                        IDEN1"))';';
```

Thus a declaration of the form

integer stack [K] S;

would generate not only the necessary array for holding the contents of the stack, but also several other declarations, including:

1. An integer variable, named level_S, corresponding to the level counter of the stack. It is initialized to zero on declaration.
2. A literal, written "depth_S," for representing the depth of that stack. Its expansion is given in terms of the operator *res*, which is taken to mean the result of a previously evaluated ex-

pression, and presumed to be defined accordingly.

3. A macro definition ($PRIM_1$) which establishes, by means of a compound expression, the interpretation of the stack name in "fetch-context". This allows one to write "N: =S;" for removing the value from the top of the stack S and assigning it to the integer variable N.
4. A macro definition ($REFR_0$) which establishes the corresponding "store context" operation. One can then write "S: =5;" to push a new value into the stack.

The predicate

The *where* clause provides a way of specifying additional conditions which must be satisfied in order that the configuration defined by the phrase constitute a call on that particular macro. Its absence implies that the syntactic structure of the phrase is sufficient to identify a call. When present, it serves to introduce additional selectivity into the definition, which enhances the effect of conditional expansion. It is also a vehicle for enlarging the local syntactic context established at each call on the macro, thereby expanding the set of formal parameters declared within the definition.

As construed here, a predicate would be written as a sequence of calls on specialized logical functions, separated by the usual operators of predicate calculus (conjunction, disjunction, implication, etc.) and grouped by parentheses. The list which follows is indicative of the kind of functions which might be appropriate:

1. $S_i = S_j$
   which decides whether the syntactic configurations associated with the two previously declared formal parameters, S_i and S_j, are structurally equivalent. S_j may instead be the symbol ϵ, which is used to decide whether S_i represents a construct corresponding to the empty phrase.
2. $S_i \neq S_j$
   which is the opposite of function (1). The following definition

   macro $BLOC_0 \rightarrow$'$LABL_1$: *begin* $STATLIST_1$ *end* $LABL_2$;'
   where $LABL_2 \neq \epsilon \supset LABL_2 = LABL_1$
   means . . .

   illustrates the use of these functions in a predicate.
3. $S_i \rightarrow$'phrase'
   where S_i is a previously declared parameter representing a category, and the phrase is written analogously to that of the production in a *macro* clause. It verifies whether the immediate substructure of the specified parameter corresponds to the indicated configuration. The constituents of the phrase are also declared as formal parameters. An interesting example is suggested by a peculiarity in PL/I, wherein the relation "7<6<5" is found to be true. A possible remedy might be formulated as follows:

   macro $REL_0 \rightarrow$'$REL_1 < EXPR_3$'
   where $REL_1 \rightarrow$'$EXPR_1 < EXPR_2$'
   means '$REL_1 \wedge res(EXPR_2) < EXPR_3$';

   The production in the where clause is assumed to be included in the base language, and "REL ∧ REL" is taken to be syntactically defined elsewhere.
4. $S_i \Rightarrow$'phrase'
   which is analogous to function (3), except that it verifies the sub-structure at an arbitrary depth, even to the terminal string. An example of its use might be:

   macro $ASGN_0 \rightarrow$'$REFR_1$: =$EXPR_1$'
   where $REFR_1 \rightarrow$'$IDEN_1$'
   ∧ $EXPR_1 \Rightarrow$'$IDEN_2$+1'
   ∧ $IDEN_1 = IDEN_2$
   means . . .

   These functions are readily generalizable into an extremely powerful pattern-matching mechanism.
5. $\exists S_i \rightarrow S_j$
   which determines, in the local syntactic context of the previously declared parameter, S_j, whether the immediate antecedent of S_j corresponds to the category specified by S_i. Also, S_i is declared as a formal parameter representing the "father" of S_j.
6. $\exists S_i \Rightarrow S_j$
   which is simply a generalization of function (5), establishing S_i as the (nearest) direct antecedent of S_j, regardless of the distance, which belongs to the category named by S_i. For example, to access the name ($IDEN_1$) of the procedure in which a particular statement ($STAT_1$) is imbedded, one might write a *where* clause of the

following form:

where ∃ $PROC_1 \Rightarrow STAT_1$
∧ $PROC_1 \rightarrow$ '$HEAD_1$ . . . '
∧ $HEAD_1 \rightarrow$ '$IDEN_1$:*proc* . . . '

The ellipsis notation is introduced with the framework of functions (3) and (4) to indicate that the structure of the corresponding constituents is irrelevant [and indeed, it may not even be knowable in the contexts that can be established by functions (5) and (6)].

7. ∃ $S_i \leftarrow$'string'
which is successful on the condition that the string (generated analogously to the replacement string) is directly reducible to the category specified by S_i, which is also declared as a formal parameter to represent the completed sub-tree reflecting the analysis.

8. ∃ $S_i \Leftarrow$'string'
which is analogous to function (7), but the condition is generalized to verify whether the string is reducible (regardless of the depth of the structure) to the specified category. The definition of the "maximum" function, which requires two syntactic macros, provides an interesting example:

macro $PRIM_0 \rightarrow$'*max* ($EXPRLIST_1$)'
where $EXPRLIST_1 \rightarrow$'$EXPR_1$'
means '($EXPR_1$)';
macro $PRIM_1 \rightarrow$'*max* ($EXPRLIST_1$)'
where $EXPRLIST_1 \rightarrow$'$EXPRLIST_2$, $EXPR_2$'
∧ ∃ $PRIM_2 \Leftarrow$'*max* ($EXPRLIST_2$)'
means '(*if* $PRIM_2 > EXPR_2$
then res $PRIM_2$
else res ($EXPR_2$))'.

9. P (arguments)
where P is the name of a semantic predicate, and the arguments may be either formal parameters or terminal symbols. Such conditions constitute a means of imposing non-syntactic constraints on the definition of a syntactic macro. They are especially applicable in those situations where it is necessary to establish the mode of a particular entity. For example, one might rewrite the factorial definition as follows:

macro $FACT_0 \rightarrow$'$PRIM_1$!'
where is_integer ($PRIM_1$)
means 'factorial ($PRIM_1$)';

In this form the definition also has the effect of allowing different meanings to be associated with the factorial operator, dependent on the mode of the primary.

10. ∃ S_i: F (arguments)
Where F is the name of a semantic function which conditionally returns a syntactic result. S_i is also declared as a formal parameter to represent this result. The semantic functions and predicates establish an interface whereby it is possible to introduce syntactic and semantic interdependencies. A likely application of semantic functions would be definitions involving identifiers:

where ∃ $LABL_1$: newlabel ($IDEN_1$) . . .

A particularly intriguing possibility is to provide a semantic function which evaluates an arbitrary expression:

where ∃ $CONST_1$: evaluate ($EXPR_1$) . . .

Obviously, this concept could be expanded to encompass the execution of entire programs, if desired.

It is evident that the role of the *where* clause in a syntactic macro definition is to provide a framework for specifying those properties which effectively cannot be expressed within the context-free constraints. The fashion in which they are isolated allows these aspects to be incorporated without sacrificing all of the practical advantages which come from adopting a relatively simple syntactic formalism as the point of departure. With respect to the model presented here, however, it is nonetheless clear that the definitional power of the syntactic macro mechanism is determined by the power of the predicates.

Operationally, the syntactic macro mechanism can be characterized by three distinct phases, each of which is briefly considered below.

Definition phase

The definition phase encompasses the different functions incorporated within the base language which act to insert, delete or modify a syntactic macro definition. Together, they constitute a facility for explicitly editing the given grammar, and are employed to form what might be called the *active syntactic definition*. This consists of the productions of the currently active syntac-

tic macros (with their associated predicates and replacements), plus the original productions of the base language. An interesting generalization would be to provide a means of selectively eliminating base language productions from the active syntactic definition, thereby excluding those constructions from the source language; they would still remain part of the base language definition, however, and continue to be considered valid in the context of a replacement. In this fashion, the syntax of an extended language could be essentially independent of the base language syntax, thus further enhancing the definitional power of the syntactic macro mechanism.

Interpretation phase

The interpretation phase includes the processing of syntactic macro calls. It consists of three separate operations: (1), recognition of the production; (2), verification of the predicate; and (3), generation of the replacement. Obviously, these operations must proceed concurrently with the process of syntactic analysis, since syntactic macro expansion is incontestably a "compile-time facility". Given the present formulation of the syntactic macro mechanism, some form of what is called "syntax directed analysis" suggests itself initially as the appropriate approach for the analyzer. It must be observed that the character of the analysis procedure is constrained to a certain extent by the nature of the predicates contained within the active syntactic definition. Furthermore, the presence of semantic predicates and functions precludes realization of the analyzer/generator as a pure preprocessor.

In general, there will be the inevitable trade-off to be made between power of definition and efficiency of operation. It is pointless to pretend that this trade-off can be completely neglected in the process of formulating the syntactic definition of a particular extended language. However, deliberate emphasis has been given here to power of definition, with the intention of providing a very general language development framework, one which furnishes an operational compiler at every stage. It is argued that the problem of obtaining an efficient compiler properly belongs to a subsequent phase.

Restriction phase

The restriction phase, as construed here, would be a separate operation, corresponding to the automatic consolidation of some active syntactic definition in order to provide a specialized syntactic analyzer for that particular dialect. The degree to which this analyzer can be optimized is determined both by the syntactic complexity of the given extended language, and by the specific constraints on further syntactic extension which are imposed at that point. If subsequent extensions are to be permitted, they might be confined within extremely narrow limits in order to improve the performance of the analyzer; they may be excluded entirely by suppressing the syntactic definition functions in the base language. In either case, various well-defined sub-sets of context-free grammars, for which explicit identification and efficient analysis algorithms are known to exist, constitute a basis for establishing the restrictions. This represents the greatest practical advantage of having formulated the syntactic definition essentially within the context-free framework.

In conclusion, it is to be remarked that syntactic extensibility is especially amenable to realization by means of an extremely powerful extension mechanism in conjunction with a proportionally powerful restrictions mechanism. This approach provides the essential definitional flexibility, which is a prerequisite for an effective language development tool, without sacrificing the possibility of an efficient compiler. In the end, however, the properties of a particular extended language dictate the efficiency of its processor, rather than the converse. This is consistent with the broadened interpretation of extensible languages discussed in this paper.

BIBLIOGRAPHY

1 T E CHEATHAM JR
The introduction of definitional facilities into higher level programming languages
Proceedings of AFIPS 1966 Fall Joint Computer Conference Second edition Vol 29 pp 623-637 November 1966

2 T E CHEATHAM JR A FISCHER Ph JORRAND
On the basis for ELF—an extensible language facility
Proceedings of AFIPS 1968 Fall Joint Computer Conference Vol 33 Part 2 pp 937-948 November 1968

3 C CHRISTENSEN C J SHAW Editors
Proceedings of the extensible languages symposium
Boston Massachusetts May 1969 SIGPLAN Notices Vol 4 Number 8 August 1969

4 B A GALLER A J PERLIS
A proposal for definitions in ALGOL
Communications of the ACM Vol 10 Number 4 pp 204-299 April 1967

5 J V GARWICK
GPL, a truly general purpose language
Communications of the ACM Vol 11 Number 9 pp 634-639 September 1968

6 E T IRONS
Experience with an extensible language
Communications of the ACM Vol 13 Number 1 pp 31-40 January 1970

7 Ph JORRAND
Some aspects of BASEL, the base language for an extensible language facility
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 14-17 August 1969
8 B M LEAVENWORTH
Syntax macros and extended translation
Communications of the ACM Vol 9 Number 11 pp 790-793 November 1966
9 M D McILROY
Macro instruction extensions to compiler languages
Communications of the ACM Vol 3 Number 4 pp 214-220 April 1960
10 A J PERLIS
The synthesis of algorithmic systems
First ACM Turing Lecture Journal of the ACM Vol 14 pp 1-9 January 1967
11 T A STANDISH
Some features of PPL, a polymorphic programming language
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 20-26 August 1969
12 T A STANDISH
Some compiler-compiler techniques for use in extensible languages
Proceedings of the Extensible Languages Symposium
SIGPLAN Notices Vol 4 Number 8 pp 55-62 August 1969
13 A VAN WIJNGAARDEN B J MAILLOUX
J E L PECK C H A KOSTER
Report on the algorithmic language ALGOL 68
MR 101 Mathematisch Centrum Amsterdam October 1969
14 B WEGBREIT
A data type definition facility
Harvard University Division of Engineering and Applied Physics unpublished 1969

Vulcan—A string handling language with dynamic storage control*

by E. F. STORM

Syracuse University
Syracuse, New York

and

R. H. VAUGHAN

National Resource Analysis Center
Charlottesville, Virginia

INTRODUCTION

The implementation of the man-machine interface for question-answering systems, fact-retrieval systems and others in the area of information management frequently involves a concern with non-numeric programming techniques. In addition, theorem proving algorithms and more sophisticated procedures for processing natural language text require a capability to manipulate representations of non-numeric data with some ease, and to pose complex structural questions about such data.

This paper describes a symbol manipulation facility which attempts to provide the principal capabilities required by the applications mentioned above. In order to reach this goal we have identified the following important and desirable characteristics for a set of non-numeric programming capabilities.

1. Conditional Expressions: Since the formal representations of non-numeric information are ordinarily defined inductively, it is to be expected that algorithms to operate on such representations will also be specified inductively, by cases. A conditional language structure seems appropriate for a "by-cases" style of programming.

2. Storage Maintenance: Assemblers and other higher-level languages eliminate many of the troublesome aspects of the storage allocation problem for the user. Very little use has been made, however, of more sophisticated storage maintenance functions. Non-numeric computation is provisional in the sense that one ordinarily wants to transform a piece of data only if that datum (or some other) has certain properties. For example, a certain kind of English sentence with a verb in the passive, may want to be transformed into a corresponding sentence with an active verb. Or, in a theorem proving context, two formal expressions may have joint structural properties which permit a certain conclusion to be drawn. In practice, however, it is the rule rather than the exception that a datum will fail to have the required property, and in such a case one wishes that certain assignments of values had never taken place. In order to accommodate these very common situations the semantics of *Vulcan* are defined and implemented so that changes to the work space are provisional. While this policy requires some complex machinery to maintain internal storage in the presence of global/local distinctions and of formal/actual usage, these maintenance features give *Vulcan* much of its power and flexibility.

3. Suppression of Bookkeeping Detail: A programmer should never need to concern himself with storage allocation matters. Nor should there be troublesome side effects of the storage maintenance conventions. Specifically it should be possible to call a set of parameters by name in invoking a procedure or subroutine so that changes to the values of actual parameters may easily be propagated back to the calling context. In such a case no special action should be required from the programmer. In addition the details of the scan of a symbol string to locate an infix substring should never intrude on the programmer's convenience. And the use of local/global distinctions and formal/actual

* This work was supported by the National Resource Analysis Center in the Office of Emergency Preparedness.

usage should require no special action in a recursive situation.

4. Numeric Capabilities: It should be possible to perform routine counting and indexing operations in the same language contexts that are appropriate for processing symbol strings. At the same time, more complex numerical operations should be available, at least by means of linkage to a conventional numerical language.

5. Input/Output: Comprehensive file declaration and item handling facilities should be included if the non-numeric features are to be useful in realistic applications. Simple formatting conventions should be available to establish a correspondence between the fields of a record and a suitable set of symbol strings.

6. Interpretive Execution: There is little penalty associated with the interpretive execution of non-numeric algorithms, since the bulk of the system's resources are concerned with accommodating a sequential, fixed-field system to a recursive, variable-field process. In addition, interpretive execution is easier to modify on a trial basis, and permits some freedom in the modification of source language syntax, provided there is an intermediary program to convert from source code to the internally stored form, suitable for interpretive execution.

While there are other desirable features for a very general programming language, these were accepted as a minimum set for exploratory purposes. An overall goal was to attain a reasonably efficient utilization of machine resources. In this implementation study it was felt desirable to achieve running speed at the expense of storage utilization if a choice were required. Since most non-numeric computing processes are thought to be slow in execution, it was decided to emphasize speed whenever possible in the *Vulcan* system.

List processing certainly plays a central role in the applications contemplated here. But the *Vulcan* language was initially intended to be experimental and to provide an exploration tool, and the implementation was therefore restricted to string manipulation, elementary arithmetic and file handling.

OVERVIEW

The *Vulcan* language has been successfully implemented on a Univac 1108 system running under EXEC-8, and a comprehensive programmer's reference manual is available.[1] The emphasis in the implementation of *Vulcan* has been on providing a powerful storage maintenance structure in place of complex and general elementary operations. From experience with applications this has been a satisfactory compromise. Extravagant elementary operations have not been so commonly needed, and when needed they are easily supplied as specially tailored *Vulcan* procedures. Storage maintenance for a recursive situation, on the other hand, would be much more difficult to supply in terms of more conventional programming language structures.

Vulcan is an imperative rather than a functional language. Since every call on a *Vulcan* procedure may be treated both as an imperative assertion and as a Boolean expression there are places in the language design where the notion of truth value assignment has a character not predictable from more conventional usage. The conventions adopted to cover these cases may be seen to be justified by their use in applications.

Since *Vulcan* is a conditional language there are no labels and no GOTO statements. In a word, the logical structure of an algorithm must be expressed in purely inductive terms.

For the numerical calculations associated with a string manipulation algorithm there are rudimentary arithmetic operations and conversions between alphanumeric and binary, and there is a comprehensive range test. All of these operations are defined only for binary integers. More complex numerical processing may be invoked by coding a Fortran program with parameters passed to it from *Vulcan*. While there are restrictions on this facility it has been found to be more than adequate for the situations encountered so far.

A complete package of file processing functions is available as an integral part of the *Vulcan* system. Individual records can be read or written, files opened or closed, temporary or permanent, on tape or mass storage. By specifying a format in terms of lengths of constituent substrings, a record can be directly decomposed into its fields by a single call on the item handling facility. Calls on the item handler are compatible with the Boolean character of a *Vulcan* procedure.

There is an initial syntax scanner which puts the *Vulcan* constructs into a standard form suitable for interpretive execution. There are several constructs which are admitted by the syntax scanner for which there are no interpretable internal codes, and the scanner is used to supply equivalent interpretable internal codes for these situations. The ability to deal with quoted material in any context appropriate to an identifier is a case in point.

The scanner has been implemented so that a *Vulcan* program may be punched on cards in free-field style. There are no restrictions on the position of *Vulcan* constructs on the program cards except that a dollar

sign (signalling a comment card) may not appear in column 1, and columns 73–80 are not read.

The more common run-time errors are recognized by the interpreter and there are appropriate error messages. As with any non-numeric facility, restraint and judgment are required to avoid situations where excessive time or storage can be used in accomplishing very little.

The entire *Vulcan* scanner/interpreter occupies approximately 3900 words of core. A small amount of storage is initially allocated for symbol tables and string storage. When this storage is exhausted additional 5000 word blocks of storage are obtained from the executive. Routine data processing computations seem to make modest demands on storage, while a theorem-prover may consume as much storage as is given it.

A system written in *Vulcan* consists of a set of *Vulcan* procedures. A procedure is a sequence of statements, and a statement is a sequence of clauses. A clause is conditional in character and consists of a series of basic symbol manipulation functions, Input/Output operations, a limited set of arithmetic facilities, and procedure calls. The language is recursive in execution so that a call on a procedure is executed in a context which depends on the data available at the time the call is made. The distinctions between local and global identifiers and between formal and actual parameters that are common to Algol are explicitly utilized in *Vulcan*.

LANGUAGE DEFINITION

Symbol strings

A string is a sequence of zero or more occurrences of characters from the UNIVAC 1108 character set. In particular, the empty string, with zero character occurrences, is an acceptable string. A string is normally referenced with an identifier and an identifier to which no string has been assigned is said to be improper. (One common run-time error results from an attempt to use an improper identifier in an incorrect way.) A symbol string may also be quoted directly in the context of an instruction. Except for the left-hand side of an infix or assignment operation, anywhere that a string identifier may be used, a quoted literal string may be used in place of that identifier. For example, both

(1) WRITE ('ABC')

and

(2) X = 'ABC', WRITE (X),

cause the string 'ABC' to be printed.

A facility exists to assign a literal string to an identifier:

(1) X = 'ABC'

(2) Y = (assigns the empty string to Y)

Quoted strings may be associated together from left to right. Suppose one wishes to assign the following literal string:

RECORDS CONTAINING 'ABC' ARE LOST.

The following literal assignment will create and store the above string:

X = 'RECORDS CONTAINING' " ' 'ABC' " '

'ARE LOST.'

Spaces outside quote marks are ignored by the translator. Note that five sub-strings are quoted in the above literal assignment:

RECORDS CONTAINING

'

ABC

'

ARE LOST.

The string value of an identifier is called the referent of that identifier and it may be changed as a result of an operation. Note that the quote mark itself is always quoted in isolation.

Language structure

The instructions in *Vulcan* are embedded in expressions which, like instructions, come out true or false. A clause is an expression which has an antecedent and a consequent part, separated by a colon, and bounded by parentheses. The instructions are coded in the antecedent and consequent parts and are separated with commas. For example,

$$(\phi_1, \phi_2, \ldots, \phi_n{:}P_1, P_2, \ldots, P_m),$$

where the ϕ_i and P_i are *Vulcan* instructions.

A clause comes out true if all the instructions in the antecedent part, executed from left to right, come out ture. In this case, all the operations in the consequent

part are executed, from left to right. For example, the clause

$$(\phi_1, -\phi_2{:}P_1)$$

will come out true and P_1 will be executed just in case instruction ϕ_1 comes out true and instruction ϕ_2 comes out false (its negation making it true).

A clause with no antecedent part always comes out true:

$$(:P_1, P_2)$$

The consequent part of a clause may also be empty:

$$(\phi_1, \phi_2:)$$

A clause with neither an antecedent nor a consequent part comes out true and performs no computation.

$$(:)$$

A statement is a series of clauses enclosed in square brackets:

$$[(\phi_1{:}P_1)\ (\phi_2{:}P_2)\ \cdots\ (:)]$$

The consequent part of at most one clause will be executed within a statement. Starting with the left-most clause, if a particular clause comes out true (as the result of all the tests in its antecedent part coming out true), then, as soon as execution of all the operations in the clause is finished, the remaining clauses are skipped and execution begins in the next statement. If a particular clause comes out false (as the result of some test in its antecedent part coming out false), then, execution begins on the next clause. If any clause comes out true in a statement, then, the statement is considered to come out true. If all clauses in a statement come out false, then, the statement is considered to come out false.

A procedure body is a sequence of statements bounded by angle brackets, ⟨ and ⟩. Each statement in a procedure body is executed once in turn regardless of the truth-value of the individual statements.

A procedure consists of the word PROCEDURE, a procedure name, a list of formal parameters, the necessary local and global identifier declarations, and a procedure body. The truth-value of a procedure is set to be the same as the truth-value of the last statement in the procedure body. For example, the form of a procedure might be:

```
PROCEDURE T4(X);
LOCAL X1, X2; GLOBAL Y1, Y2;
    ⟨ [(φ1:P1)
      (φ2:P2)]
      [(φ3:P3)] ⟩
```

A program is a set of procedures with a period (.) terminating the last procedure. The initial procedure is executed first and acts as a driver or main program for the system. All other procedures are executed only by means of calls to them. The completion of this initial procedure terminates the run.

String manipulation operations

There are two basic string manipulation instructions, the concatenate operation and the infix test.

Concatenation

Concatenation is used to build strings out of other strings and the operation always comes out true. The operation has the following format:

$$X = X1\ .\ X2\ .\ \ldots\ .XN,$$

where X, X1, through XN are identifiers. For example:

(1) X = Y1.X1.Y2.Z

(2) X = T1.'A QUOTED VALUE'.USE

The strings referred to by the identifiers and literal strings on the right-hand side of the assignment symbol (=) are concatenated and the resulting string is assigned to the identifier on the left. In the operation, each of the identifiers and literal strings to the right of the equal sign must be separated with a period.

The identifier on the left may appear several times to the right of the equal sign. In this case, it retains its original referent until the entire concatenation is completed. The resulting concatenated string is then assigned to the identifier to the left of the equal sign. For example, if X is 'AB' and Y is 'C', then the operation X = Y.X.Y.X.Y results in setting X to 'CABCABC'. The identifier on the left may be either proper or improper, but each identifier on the right must be proper.

Infix test

The basic test on a symbol string is the infix test, and is written

(1) X/*.Y.*

(2) Z/*.'ABC'.*

In the first example, the test comes out true if the referent of Y occurs as an infix in the referent of X,

and comes out false otherwise. In the second case, if the literal string 'ABC' occurs in the referent of Z, then the test comes out true, and false otherwise. The asterisks play the role of dummy identifiers to take up the residue prefix and residue suffix substrings defined by the occurrence of Y in X.

There are three variants of this test, typically written X/Y.*, X/*.Y and X/Y, asking whether (the referent of) Y begins X, or ends X, or if they are equal, respectively. Finally, EMPTY(X) asks of the referent of X is the empty string.

In each of the above operations, all identifiers must be proper.

A generalization of the infix test occurs when one wants to retain, for later processing, the residue prefix and suffix substrings that are defined as a consequence of the infix test coming out true. For these purposes the test may be written as follows:

X/U.'ABC'.V

or

X/U.Y.V

where U and V are improper and X and Y are proper. If there is more than one occurrence of the referent of Y in X, *Vulcan* identifies the left-most occurrence. For example, if the referent of X is 'ABCDEFGDEK', one could ask if X contains the string 'DE':

X/U.'DE'.V

The test will come out true and assign the string 'ABC' to U and the string 'FGDEK' to V. If an infix test fails, then the identifiers U and V remain improper.

Procedure calls and local/global distinctions

Procedure calls and truth-value control

Vulcan procedures are subroutines and the procedure call is the mechanism for calling a subroutine. Execution of *Vulcan* procedures is strictly recursive so that locally declared identifiers are recreated at each new level of recursion. Parameters to the procedure are also reconstituted at each level of execution. Distinct versions of identifiers and parameters are preserved at each level so that when execution returns to a given level of recursion, the versions generated by subordinate calls are lost, higher-level versions are still retained in a push-down store, and only the current level versions are available to the procedure.

A procedure is given to the *Vulcan* system with a (possibly empty) set of formal parameters as follows:

(1) SUBR

(2) T(A, B, C)

The procedure in (2), for example, may be called with any identifiers in place of A, B, C, respectively. Thus, the procedure call may be written:

T(X, Y, Z)

At execution time, the *Vulcan* interpreter makes copies of the strings associated with X, Y, Z and assigns them temporarily to A, B, and C, respectively. The procedure is then executed on its formal parameters: A, B, C. When execution is completed, the referents, if any, of A, B, and C are copied back to X, Y, and Z, respectively, subject to the condition that the procedure call comes out true. If the procedure call comes out false, the strings are not copied back and X, Y, and Z remain as they were prior to the procedure execution.

The procedure call is the sole mechanism for calling a subroutine in *Vulcan*; and local identifiers, global identifiers, and formal parameters, as well as quoted literal strings, may be included in a list of actual parameters.

A procedure can be caused to be repeated until the call comes out true (or false). This allows an iterative facility which sometimes uses less storage than an equivalent recursion.

An asterisk preceding a procedure call implies that the procedure is to be executed repeatedly until the procedure call comes out true.

*RECORD (X, Y, Z)

An asterisk with a minus sign implies that the procedure is to be repeated until the negation of the procedure call comes out true (i.e., "repeat until false").

* — RECORD (X, Y, Z)

In a conventional programming language the logical flow of execution is controlled by branch instructions and unconditional transfers. Since there are no labels in *Vulcan*, there are no transfer instructions. The flow through a *Vulcan* program is controlled by causing certain clauses, statements or procedures to come out true in some cases and false in others. The conditional nature of a clause allows the programmer to choose between alternative paths, and the procedure call allows either path to be as complex as may be required. The beginning user of *Vulcan* should make special note of the fact that control is handled in terms of certain con-

structs coming out true or false, and this is a definite departure from conventional programming practice. With this convention, it is sometimes useful to be able to force the last statement of a procedure (and hence the procedure) to be false although some clause in it has come out true. This is done by the FALSE command. The FALSE operation, if executed in any clause, will cause control to skip to the end of the statement and will set the truth-value of the statement to be false. A generalization of the FALSE command causes control to pass immediately to the end of a procedure. The command RETURN causes exit of the current procedure with true, and the command RETURNF causes exit of the current procedure with truth-value false.

Local, global distinctions, and clause workspace

A procedure S is in the scope of a procedure T if T contains a call on S, or else if T calls some other procedure which in turn has a call on S, or else, etc.

Suppose it is desirable that an identifier be declared in the procedure T and be available to the procedure S without including it as an actual parameter in a call on S. Such an identifier will be available throughout the scope of T if it is declared in T to be GLOBAL. In other words, that identifier will be available in T, in all the procedures which are called by T, in all the procedures which are in turn called by those, and so on. An identifier which is not declared global is then, by contrast, a local identifier, and in all cases it must be so declared with the reserved word LOCAL. Such an identifier is available only within the procedure in which it is declared and its referent is not available in any of the procedures which are called by it, except as an actual parameter. Each identifier, then, must be declared as either local or global, or implicitly declared by inclusion in a formal parameter list.

Each time a procedure is called, its declared identifiers are constituted afresh so that if a procedure calls itself recursively its declared identifiers have separate and distinct copies at each level of recursion.

The local, global distinction of identifiers is further used in that, within the execution of a clause, changes to the referents of local identifiers and formal parameter identifiers are contingent upon the final truth-value of the clause. However, changes to the referents of global identifiers are not subject to this condition. If a clause comes out true, changes to all identifiers are permanent. If a clause comes out false, changes to global identifiers are permanent but changes to local identifiers and formal parameters are obliterated. In effect, the execution of a clause is carried out in a temporary workspace so that changes made to any particular identifier are made permanent just in case one of the following conditions holds: the clause containing the assignment came out true or the identifier in question was global.

Storage is managed by dividing it into two regions—one for identifiers and their properties, and another for strings themselves. An implementation of the storage requirements of *Vulcan* is inevitably complex. We note here that experience in using *Vulcan* led to the adoption of a garbage collection policy that is very similar to that described by Kain.[2]

Arithmetic operations

The arithmetic facilities provided in *Vulcan*, while not complex in structure, allow for most counting, averaging or testing that is needed. No floating point instructions are provided, only integer arithmetic. Normally, the representation of a binary integer is a string which is six characters (i.e., 36 bits) in length.

A binary integer string may be converted to its Field Data equivalent with the command ALPHA (X), where X is the identifier for the binary integer string. As a result of the operation, X is the identifier for a string of numeric characters which is the value equivalent of the binary integer. If the referent of X is a negative integer, a minus sign (−) is prefixed to the converted string; however, no sign is attached for a positive integer. Although a *Vulcan* binary integer is a string six characters in length, any string $\leq$ six characters may be interpreted as a binary integer. Hence, the ALPHA (X) instruction allows the resulting referent of X to have a length of from one to six characters. For example, if the twelve-bit (two-character) field extracted from a data file is binary, then it may be converted to its alphanumeric equivalent with the ALPHA (X) command. No other arithmetic operation, however, allows this special representation of a binary integer. An error results if X is improper, empty, or is more than six characters in length. ALPHA (X) always comes out true.

Inverse to the ALPHA (X) operation, a string of numeric characters can be converted to a binary integer with the BINARY(X) operation. The legal form for a numeric string to be converted to binary is a plus (+) or minus (−) sign or neither, followed by a purely numeric field of at least one and at most 11 characters. Blank characters may precede the sign, if any, may intervene between the sign and the numeric field, and may trail the numeric field. For example, the following

are well-formed numeric strings (the 'Δ' symbol being interpreted as the blank character).

(1) '123'

(2) 'ΔΔ9ΔΔ'

(3) 'Δ+Δ68Δ'

(4) ' –0'

The following are not well-formed numeric strings.

(1) '12 3'

(2) 'ΔΔ9.000'

(3) 'Δ$Δ19.24'

(4) '+ –86'

If the string to be converted is not well-formed, then BINARY(X) comes out false. If it is well-formed, then the command comes out true and the referent of X is the converted binary integer string, six characters in length. If X is improper, error termination occurs.

Arithmetic operations are listed below.

(1) ADD (X, Y, Z) means X = Y + Z

(2) SUB (X, Y, Z) means X = Y – Z

(3) MPY (X, Y, Z) means X = Y * Z

(4) DIV (X, Y, Z) means X = Y / Z

(5) SETZRO (X) means X = 0

where the identifiers Y and Z must have referents that are binary integers, six characters in length. Each operation always comes out true. The operation DIV (X, Y, Z) yields the integral quotient in X and discards the remainder.

There is one numeric test:

RANGE (X, Y, Z),

where the identifiers X, Y, and Z must be binary integers, six characters in length. RANGE(X, Y, Z) comes out true just in case $X \leq Y \leq Z$ and comes out false otherwise.

The following *Vulcan* program illustrates the basic operations and the language structure presented thus far.

In this example, as part of a fact retrieval query scheme, the task is to simplify an English language sentence by replacing all occurrences of the string 'GREATER THAN' by the string '>', and preserve the original.

Procedure INITIAL sets the values for the identifiers W, X, and Y and then calls procedure REP, passing in the actual parameters W, X, Y, and Z to formal parameters A, B, C, and D. (Note that W, X, and Y are proper and that Z is improper when the call is made.) Procedure REP then replaces all occurrences of string B in string A with string C and calls the new string D. Notice that if no occurrence of B is found in A, then D is simply set to the referent of A. Called with the input given in procedure INITIAL, REP will set the referent of Z to 'LIST ITEMS IF AGE > 24 AND WEIGHT > 150'.

```
PROCEDURE INITIAL;

LOCAL W, X, Y, Z;

⟨[(: W = 'LIST ITEMS IF AGE GREATER THAN
        24 AND WEIGHT GREATER THAN 150',

    X = 'GREATER  THAN', Y = '>', REP(W,
        X, Y, Z) )]⟩

PROCEDURE REP(A, B, C, D);

LOCAL X1, X2;

⟨[(A/X1.B.X2 : REP(X2,  B,  C,  D), D = X1.C.D)
  (  :D = A)]⟩
```

INPUT OUTPUT OPERATIONS

The Input-Output operations in *Vulcan* fall into two categories: (1) card reading and line printing operations, and (2) file handling operations (for tapes, Fastrand files, etc.).

Card read and line print

There are standard operations to read a string from a card and to write a string on the line printer. The instructions are as follows:

(1) WRITE (X1, . . . , XN)

(2) PRINT (X1, . . . , XN)

(3) READ (X1, . . . , XN)

WRITE causes the referents of the strings for each identifier in the list to be printed on successive lines. PRINT, for each identifier in the list, writes out the

identifier, followed by an '=' sign, followed by the string. If a string is longer than the number of print positions on the line, remaining characters of the string are printed on subsequent lines.

For each identifier in the list, READ reads the next card and assigns the string of characters on the card to the next identifier. Trailing blanks on a card are deleted before assigning the string to the identifier. If a blank card is read, the empty string is assigned to the identifier.

The WRITE and PRINT operations always come out true. READ comes out false if any EXEC 8 control card is read, but comes out true otherwise.

There is a modified version of READ available for use with remote terminal applications which avoids unwanted line feeds and carriage returns.

File handling operations

The traditional concept of reading and writing items (logical records) and blocks (physical records) is extended in *Vulcan* to provide for the handling of individual fields within items. An item in a file is thought of as a single string which may be decoded into various substrings, or fields. Alternatively, a set of substrings, or fields, may be grouped together to form an item which is then put into a file. These two functions are accomplished by the ITMREAD and ITMWRITE operations, respectively. Supplied on each ITMREAD or ITMWRITE request is the name of the file to be processed, a format which is a definition of the fields within the item, and a list of identifiers. The specific relation between the format and the list of identifiers in each particular request is the subject of Part B of this section. The general sequence of commands for manipulating data files in *Vulcan* is as follows. Prior to executing the *Vulcan* program, buffer storage requirements must be supplied with the FILE statement. Each file to be processed must be assigned, either externally or dynamically, through the CSF instruction (described later). The file must be opened before reading or writing and then closed after processing. A file may be reopened after it is closed, and it need not be reassigned unless it has been freed. The *Vulcan* file handling capability employs the Input-Output Interface for FORTRAN V under EXEC 8 described in the National Resource Analysis Center's Guidebook, REG-104. The user is advised to read this manual before using the *Vulcan* file handling commands. The instructions for file handling and their calling sequences follow.

1. OPEN: opens a file for processing.
   CALL: OPEN(FN, MODE, TYPE, LRS, PRS, LAF, LAB), where
   FN = File name (1–12 characters)
   MODE = Mode of operation ($1 \leq MODE \leq 7$)
   TYPE = Type of blocks ($1 \leq TYPE \leq 5$)
   LRS = Logical record size, for fixed size records. ($1 \leq LRS \leq PRS$). If LRS = '0' then variable size records are indicated.
   PRS = Physical record size ($1 \leq PRS \leq N$, where N is buffer size stated on the FILE declaration).
   LAF = Look-ahead factor (is ignored if LAF = ⟨empty⟩)
   LAB = Label (is ignored if LAB = ⟨empty⟩).

Only the first five arguments are necessary for opening a file. The label field (LAB), or the label (LAB) and look-ahead factor (LAF) fields may be omitted in the call. The OPEN request comes out true if the operation is completed normally and comes out false otherwise. I/O status information may be retrieved with the STATUS instruction, described later in this section. For example, the *Vulcan* instruction

OPEN('TEACH', '2', '2', '28', '28')

will open an output file named 'TEACH' (with fixed size blocks with no block counts and no sum checks) of 28-word records each and 28-word items (i.e., one item per physical record).

2. CLOSE: closes a file that has been opened.
   CALL: CLOSE (FN, TYPE), where
   FN = File name (1–12 characters)
   TYPE = Close type ($1 < TYPE < 5$).

CLOSE always comes out true. For example, the instruction

CLOSE('TEACH', '4')

will close file 'TEACH', without rewind if 'TEACH' is a tape file, and with rewind if 'TEACH' is on a mass storage device.

3. REWIND—rewinds a file.
   CALL: REWIND (FN, where
   FN = File name (1–12 characters).

The REWIND instruction rewinds a tape or drum

file that has been opened with the OPEN statement. It always comes out true.

4. ITMREAD and RREAD—serve to input a new record from a file.
   CALL: ITMREAD (FN, F, List), where
   FN = File name (1–12 characters)
   F = Format
   List = List of identifiers.
   CALL: RREAD (FN, IN, F, List), where
   FN = File name (1–12 characters)
   IN = Item number (binary integer)
   F = Format
   List = List of identifiers.

ITMREAD reads the next available item from the file while RREAD reads the item in item position IN. IN = 1 causes the first item in the file to be read while IN = 0 is equivalent to an ITMREAD command (i.e., a sequential read). The ITMREAD and RREAD commands come out true if an item is read and come out false if an end-of-file or an abnormal status is encountered.

5. ITMWRITE and RWRITE—serve to output a record to a file.
   CALL: RWRITE (FN, IN, F, List), where
   FN = File name (1–12 characters)
   IN = Item number (binary integer)
   F = Format
   List = List of identifiers

ITMWRITE writes into the next available item position of the file while RWRITE writes into the item position specified by IN. IN = 1 specifies the first item and IN = 0 causes RWRITE to function like ITMWRITE (i.e., write into the next available item position). ITMWRITE and RWRITE come out true if the task is normally completed and come out false otherwise.

6. STATUS—returns the status of an operation.
   CALL: STATUS(X), where
   X = Identifier having the returned status as its referent.

The STATUS instruction is used only in connection with the OPEN, ITMREAD (RREAD), ITMWRITE (RWRITE), and CSF operations. Each of these instructions is a link to the executive system where a completion status is returned. If the particular task called upon terminates normally, then, the *Vulcan* instruction is set true, indicating normal completion. If the task invoked is not completed normally (e.g., an end-of-file is encountered when reading), then, the *Vulcan* instruction is set false, indicating an abnormal status returned from the EXEC. The particular value of the status may be retrieved with the STATUS (X) instruction. The identifier X will have as referent a six-character binary integer string, which is the status code of the last operation performed. The STATUS instruction itself always comes out true.

7. FILE—is a declaration of file facilities needed.
   Format: FILE n, m; where
   n = the number of files in use simultaneously in a *Vulcan* program
   m = the number of machine words of the maximum physical record size of any file.

The FILE statement for any *Vulcan* program may appear in the declarations of any *Vulcan* procedure, since it is really a meta-command to the *Vulcan* processor and not an executable *Vulcan* instruction. If more than one file statement is encountered, then, the last one processed by the translator is entered and earlier ones ignored. If no FILE statement is encountered, then, no buffer space is allocated and no files may be opened.

Item and field processing

The main result of *Vulcan's* ITMREAD (RREAD) is to assign the characters in specified fields of an item, sequentially, to a list of identifiers. The ITMWRITE (RWRITE), alternatively, constructs an item from the strings of a list of identifiers, again, according to a specification of field sizes.

The Format identifier refers to a string whose legal construction is specified as follows:

⟨format⟩::=(⟨phrase list⟩)
⟨phrase list⟩::=⟨phrase⟩|⟨phrase list⟩,⟨phrase⟩
⟨phrase⟩::=⟨X-phrase⟩|⟨S-phrase⟩|⟨U-phrase⟩
⟨X-phrase⟩::=X⟨integer⟩
⟨S-phrase⟩::=S⟨integer⟩
⟨U-phrase⟩::=U⟨integer⟩

Two typical legal formats are '(S24, U24, X19, S4)' and '(X1, S2451, U18)'.

The format acts as a specification of field sizes to read or write an item. On output of an item, the ⟨X-phrase⟩, X19, means to skip over the next 19 characters in the item. (Effectively, this outputs 19 @ signs

to the item.) The ⟨S-phrase⟩, S240, causes the referent of the next list identifier to be written into the next 240 character positions of the item. If the string is shorter than the field size (i.e., <240 characters), binary zeros (@ signs) are filled out to the right. If the string is longer than the field size specified by the ⟨S-phrase⟩, an error termination occurs. On output, the ⟨U-phrase⟩ is exactly like the ⟨S-phrase⟩.

On input of an item, the ⟨X-phrase⟩, X19, means to skip the next 19 characters in the item. The ⟨S-phrase⟩, S240, means to assign the next 240 characters in the item to the next list identifier, but with any trailing binary zero characters (@ signs) deleted. The ⟨U-phrase⟩, U240, is like the ⟨S-phrase⟩ but causes unconditional acceptance of all of the next 240 characters (specifically including @ signs).

The ⟨S-phrases⟩ and ⟨U-phrases⟩ of a format string match up in a one-to-one correspondence with the list of identifiers in the ITMREAD or ITMWRITE request. It need not be, however, that the number of list identifiers equals the number of ⟨S-phrases⟩ plus ⟨U-phrases⟩. The shorter of the two lists terminates the I/O request on that item.

To illustrate the ITMREAD, suppose we define the following item: (Note that the 'Δ' symbol is used for the blank character.)

Word

1	Δ	A	A	C	1	9
2	Δ	N	R	A	C	/
3	M	C	L	@	Δ	Δ
4	O	F	F	I	C	E
5	Δ	O	F	Δ	E	M
6	E	R	G	E	N	C
7	Y	Δ	P	R	E	P
8	A	R	E	D	N	E
9	S	S	@	@	@	@

After execution of the following clause,

(T1 = 'FILENAME',

T2 = '(X4, S2, X1, U9, X2, S36)',

ITMREAD (T1, T2, A, B, C, D):)

the strings A, B, C, and D will be defined as follows:

A: 19
B: NRAC/MCL@
C: OFFICEΔOFΔEMERGENCYΔPREPAREDNESS
D: (holds its previous value).

The following *Vulcan* program illustrates more generally the file processing techniques presented. The task is simply to catalog a Fastrand file, copy a tape file into it, and terminate; or, if the Fastrand file is already cataloged, simply terminate.

The first clause attempts to assign file 'FAST'; and, if it is accepted, then the file is already catalogued and the program will terminate. If the file is not already catalogued, then another assign is made, this time requesting that' FAST' be catalogued when it is freed. If it cannot be assigned again, a message is printed and the program terminates. In the second statement the file 'TAPE' is assigned, calling for the mounting of tape U1234. If this assignment cannot be made, then the run terminates with a message indicating the tape file could not be assigned. In the third statement both files are opened successfully, or else a message is printed that an open request failed and the run is terminated. Once the files are opened, procedure COPY is called until the negation of the procedure call comes out true. That is, as long as the ITMREAD command comes out true, then an item of data is written into file 'FAST' and procedure COPY comes out true, making its negation come out false. Hence, COPY is to be called again. When ITMREAD finally comes out false, the status of the last I/O operation is checked to determine if an EOF was detected or an error occurred.

```
PROCEDURE MAIN;
FILE 2,1200; GLOBAL FN1, FN2, FORMAT, B2;
⟨[(CSF ('@ASG, A FAST, F2') :
WRITE ('ALREADY ASSIGNED'), RETURN)
(CSF('@ASG, PC FAST, F2/ /TRK') : )
(:WRITE ('CANNOT ASSIGN FASTRAND FILE'),
  RETURN)]
[(CSF('@ASG, T TAPE, T, U1234') :
FN1='TAPE',FN2='FAST', B2='2', BINARY(B2))
(:WRITE('CANNOT ASSIGN TAPE FILE'), RE-
  TURN)]
[(OPEN(FN1,'4','4','100','1200'),
FORMAT='(U600)',*—COPY, CLOSE (FN1, '6'),
  (FN2, '6'))
(:WRITE('CANNOT OPEN') )]⟩
PROCEDURE COPY; LOCAL X;
⟨[(ITMREAD (FN1, FORMAT, X):ITMWRITE
  (FN2, FORMAT, X) )
(STATUS(X), RANGE (B2, X, B2):
WRITE('EOF DETECTED'),FALSE)
(:WRITE('ERROR IN READ'), FALSE)]⟩.
```

ADDITIONAL VULCAN COMMANDS

In addition to the string and file handling and arithmetic facilities already described, *Vulcan* includes

several other convenient instructions which we mention here. There is a command to decompose a string based on character position rather than character value, a command to compute the length of a string, and a command to obtain elapsed CPU time in 200 microsecond increments.

There is an instruction to send a message directly to the EXEC 8 executive request function, a page eject command and a command to obtain a Teletype break key contingency interrupt status. There is a trace facility for observing the sequence of procedure calls obeyed in execution and their resulting truth values. A standard Fortran subroutine may be called in *Vulcan*, which in turn may, under programmer direction, call other Fortran programs. The legal parameters to this standard program must be strings whose lengths are non-zero multiples of six in length.

Finally, there is a DEFINE facility in which the user may define a macro string with parameters. And a respectable set of syntax and run-time errors is included providing suitable messages.

APPLICATIONS

Vulcan has been successfully applied to several problems. A theorem proving system based on Robinson's resolution principle[3] has been implemented,[4] and a trial system for the translation of a simplified subset of German has been studied.[5] There is a general File Management System[6] which allows remote terminal communication in restricted English for the purposes of changing and interrogating any files which can be read by the 1108 executive. Fact retrieval systems tailored to special applications have been constructed, and a large scale command and control activity has used *Vulcan* procedures for the man-machine interface.

SUMARY OF INSTRUCTIONS AND SYNTAX

1. Infix test:	X/A.Y.B
	X/*.'ABC'.*
	X/*.Z1
	X/Z1.*
	X/Y
2. Empty test:	EMPTY(X)
3. Assignment:	X='ABC'
	X=''
4. Concatenation:	X=Y1.Y2.Y3.Y4
	X=
5. Procedure call:	T(X,Y,Z)
	*GO
	*—FETCH
	FLINK(X1,X2)
6. Card read:	READ(X,Y,Z)
	TREAD(A,B)
7. Line print:	WRITE(I,J,K)
	PRINT(X,Y,Z)
8. File Handling:	OPEN (FN,MODE, TYPE,LRS,PRS)
	CLOSE (FN, TYPE)
	REWIND (FN)
	ITMREAD (FN,F, A,B,C)
	RREAD (FN, IN, F,A,B,C)
	ITMWRITE (FN, F,X,Y,Z)
	RWRITE (FN, IN, F,X,Y,Z)
	STATUS(X)
9. Arithmetic operations:	ADD(X,Y,Z)
	SUB(X,Y,Z)
	MPY(X,Y,Z)
	DIV(X,Y,Z)
	SETZRO(X)
	BINARY(X)
	ALPHA(X)
10. Arithmetic test:	RANGE(X,Y,Z)
	—RANGE(X,X,Y)
11. Truth-value control:	FALSE
	RETURN
	RETURNF
12. Miscellaneous commands:	DECODE (X,F,A, B,C)
	CSF ('@ASG, T, F, T, U1234')
	LENGTH(X,Y)
	TIME(X)
	ITMSET(PAGE)
	ITMSET (CON-TIN,X)
	TRACEN (MAIN, TGO,PR)
	TRACEP (MAIN, TGO,PR)
	TRACEF
13. Identifiers:	LOCAL X,Y,Z;
	GLOBAL G1,TRU;
	LOCAL K;
14. File Facilities:	FILE 3,1792;
15. Procedure:	PROCEDURE T (K,K2);
	LOCAL B1,B2;
	GLOBAL 74; GLO BAL ONE;

```
FILE 1,500;
⟨[(. . . : . . .)
(. . . : . . .)]
[(. . . : . . .)]⟩
```

ACKNOWLEDGMENTS

Conditional expressions are familiar from LISP,[7] and many of the features of Vulcan originally appeared in CHAMP.[8] The infix test and its variants are elementary versions of the pattern-matching facilities in SNOBOL[9] but are much less comprehensive. File handling facilities are direct calls on the NRAC Input-Output package,[10] and its manual is available for use with *Vulcan*.

The authors are indebted to Edgar M. Cagley and to Richard A. Stanley for their participation in the development of *Vulcan*. Mr. Cagley designed many trial applications to prove the usefulness of the language and Mr. Stanley prepared the syntax scanner and other significant features. The management of the National Resource Analysis Center supported the project enthusiastically from its inception.

REFERENCES

1 R H VAUGHAN
Vulcan-programmer's reference manual
NRAC Executive Office of the President Office of Emergency Preparedness Washington D C TM-209 April 1970

2 R Y KAIN
Block structures, indirect addressing, and garbage collection
CACM 12 July 1969 395-398

3 J A ROBINSON
A machine-oriented logic based on the resolution principle
J ACM 12 1 Jam. 1965

4 T G HAMRICK
First-order logic on the machine
Master's thesis University of Virginia Charlottesville Va June 1969

5 G P HILL
Syntactic analysis of simple German sentences using the Vulcan programming language
School of Engineering and Applied Science University of Virginia Charlottesville Va May 1970

6 N J RAY
Information management system
NRAC Executive Office of the President Office of Emergency Preparedness Washington D C TM-208 May 1970

7 J McCARTHY et al
LISP 1.5 programmer's manual
The MIT Press Cambridge Mass 1962

8 E F STORM
CHAMP-character manipulation procedures in Algol
Comm ACM 11 Aug. 1968

9 D J FARBER et al
SNOBOL, a string manipulation language
J ACM 11 2 Jan. 1964

10 R FEDDER
Input-output interface for Fortran V under EXEC-8
NRAC Executive Office of the President Office of Emergency Preparedness Washington D C REG-104 Sept 1969

On memory system design

by ROBERT M. MEADE

Cogar Corporation
Wappingers Falls, New York

INTRODUCTION

A hierarchy of information accessibility exists in every system. Even simple calculators employ a two-level hierarchy consisting of internal registers and external key-entered data. In a typical computer system we find a multilevel hierarchy extending from working registers through random-access main-memory, to direct access devices, to sequential access devices, and on outward to off-line archives.

System design always consists primarily in specification of the hierarchy of information, of information media, and of the controls for their interconnection. Thereafter, it remains only to define the information and the operations upon it to completely specify a system.

The value of information is reflected in the frequency with which it is referenced.[1] The system design problem is to match the relative values of all information to the relative access times of storage media. This must be a dynamic matching because the value of the information in the system changes rapidly with particular activity. The hierarchy control hardware and much of the operating system software exist to accomplish the matching.

An "optimum" system can be *either* one which will perform a given application in a given amount of time at the lowest cost, *or* one which will perform the given application at a minimum cost/time product. Because of the number of alternatives for each element of the system there will be a set of nearly optimum configurations. Thus there is no one best system design. The optimum hierarchy will be determined by the application and available elements rather than by a general theory. However, the concepts discussed below can aid in choosing the best elements for a given application.

There are two natural boundaries in the hierarchy of information. The first occurs at the man/machine interface; on one side exists the information that can be electronically called by a program; on the other side resides information that can only be supplied by a human. This may be machine-readable such as a library reel of tape, or it may be first-hand input from a terminal. The second boundary occurs between *internal* information that is directly addressed and *external* information that must be transferred from another storage mechanism into the directly addressed memory. This is commonly the boundary between main memory and electro-mechanical storage. The distinguishing characteristic is that program instructions and data can be executed only from internal storage. External information must be moved into the internal memory by an explicit input/output command, generally executed by a logically independent transfer channel.

The definition of this internal-external boundary is particularly critical. The addressing architecture of the system determines the range of possible internal storage. The channel command structure determines the flexibility, autonomy and concurrency of the external storage system.

From a user's point of view, this internal-external distinction is arbitrary and awkward. For him it is an additional nuisance in the already cumbersome practice of referring to information by its system's location. It is more natural for him to refer to information by name in an inverse hierarchy of file, record, field, which are logical data entities. This hierarchic information structure is fundamental to the design of the storage hierarchy.

Systems simulating memory of a single level have been designed in order to provide a more natural programming situation. These are called virtual memory systems because they allow the programmer to operate as though physically external memory were the internal program space. The physically internal memory is made invisible or transparent to him by the software used to control the flow of information. Hardware implemented or assisted systems have been proposed,

generally based upon the use of associative memory techniques. Thus a discussion of memory hierarchy is a discussion of virtual memory systems, and conversely.

The multiple level hierarchy exists because of the cost of storage. If the storage device with the fastest access time were also the least expensive, a system would employ a single level of memory. Levels are added only as the effective performance/cost ratio of the system is improved thereby. Thus, direct access devices are used in addition to magnetic tapes because their shorter access time enhances system performance. Recently two-level implementation has similarly enhanced cost-effectiveness of main memories, as in the IBM System 360, Model 85.[2,3]

History

Basic concepts for memory hierarchy date from around 1960. The first system to employ transparent two-level memory was the Atlas.[4,5] This system used random-access core memory as a buffer backed by a large amount of drum storage. The programming space was implemented in the latter; the information was transferred in and out under a combination of hardware-software control in 512-word blocks or pages. The performance was limited by the long access time of the drum combined with the small capacity of the core.

From 1961 to 1963 high performance systems employing various buffering schemes were developed. Based upon operational/experience with LARK and STRETCH, the CDC 6600 and the IBM 360/91–95 were designed. Both of these employ a register stack to buffer the flow of information from and to main memory. This comprises a form of virtual memory which is controlled by logic making instantaneous analysis of the microstructure of the executed program. The buffer algorithm is machine-design dependent. In the model 90's,[6] this approach provided a rather general virtual memory for the instruction stream through the incorporation of the ability to do full inner loops of instructions from the buffer. However, no such loop exists for data.

Meanwhile, by 1962, the Atlas structure had been analyzed by Bloom, et al.,[7] with respect to high-performance system potential, with the result that they proposed the use of a relatively small, but very fast, main memory buffer as "look aside". This buffer was to use an associative algorithm to map blocks of the main memory for general instruction and data residence. Conceptually, this was very close to the configuration later implemented by the IBM Model 85.

In parallel with this evolution, software-controlled virtual memory systems have been developed particularly to service multiple remote terminals. These systems sequentially overlay in main memory data from many users who cannot be aware of the instantaneous allocation of memory. The user programs as though he has adequate addressable memory while the software maintains the data on external storage and calls it into memory in pages as required.

In order thus to transform physically external memory into logically internal memory, transfer and control algorithms had to be developed for the software. These were subsequently refined for internal memory hierarchy control.

In 1967 Gibson published[8] his definitive analysis of performance considerations in two-level internal memory hierarchy. This work, which led toward development of the IBM 360/85, will be used extensively below.

Prior to that time there had been much debate on how best to employ high-speed local storage. Intuitively, transparent mapping buffers seemed inefficient as compared to explicit use by a knowledgeable programmer. It turned out that this view overlooked inefficiences caused by programming overhead in direct control and the fact that the hardware-implemented control could perform more functions concurrently.

In order to prove efficiency of the transparent buffer it was necessary both to develop adequate simulation models for the system and to exercise these models over an adequately large data base of representative programs. The published results showed that programs have substantial clustering of activity (i.e., some information has relatively high value) so that subsets of the data can be collected for processing.

DESIGN

This paper's thesis is that an adequate theoretical and statistical basis for the design of a memory hierarchy appropriate to a given application now exists. This basis will be developed below.

The hierarchy exists to enhance the cost-effectiveness of the system by integrating the characteristics of dissimilar memories, trading access time and cost. In order for this to be effective there must be a substantial cost reduction accompanying substantially increased access time. This implies that different levels in the hierarchy must have a significant technical difference. One cannot achieve a sufficiently high cost/performance differential between a small and a large memory when both employ ferrite cores. Thus it was the advent of high-speed monolithic memory at the same time as the

development of hierarchy theory that made the hierarchy profitable.

Systems constructed to date have used at most one level of transparent memory. Conceptually, it is natural to extend the structure to additional levels. The discussion below is intended to apply to multiple-level systems as defined by Figure 1. In these, the inner-level buffer speed is closely matched to the processor's operating cycle.

Access time and cost remain the basic memory parameters; they are intrinsic while all others are subject to system design. The additional parameters for hierarchy design are block-size, buffer capacities, control algorithms and information transfer rates. Before discussing these in detail, a brief review of the terminology and operation of transparent hierarchy may be helpful.

The objective is to gain processing speed by causing most operational information to come from the fast buffer. Because of the clustering of information, it is probable that when data is used, it or neighboring data will be subsequently used. The logic, therefore, determines for each effective address the presence of the data in the buffer. If it is not present, the address requires an access from the main memory. The system logic then moves a contiguous data set called a block (containing the addressed data) from the outer (main) memory to the inner (buffer) memory according to a rule called the mapping algorithm. If the buffer is full, the logic must first employ a replacement algorithm to eliminate a block. The frequency of thus accessing from outside the buffer, called the miss-rate, is the prime determinant of the resulting performance.

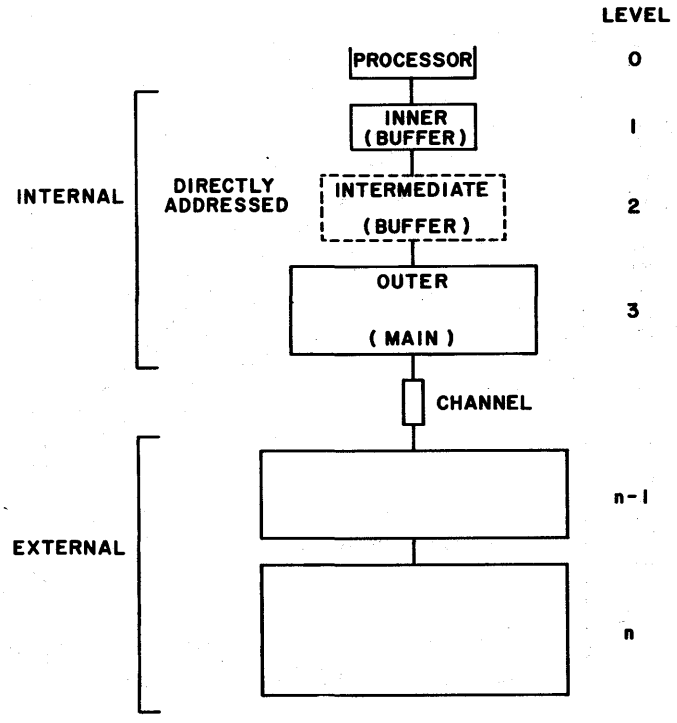


Figure 1—Hierarchy of memory

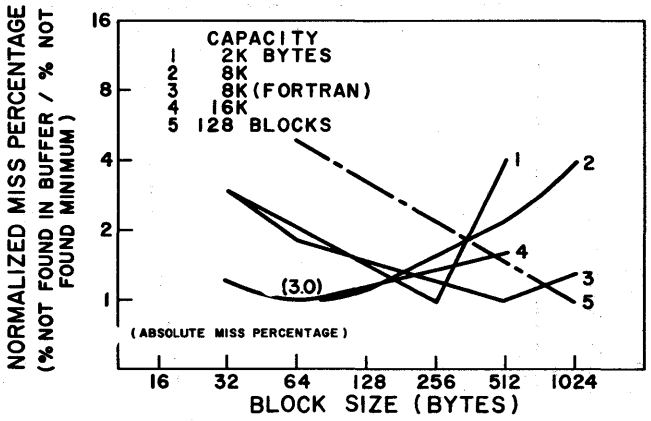


Figure 2—Effect of block size upon buffer miss rates

Block size

The size of the block (group of bytes) to be handled at the inner memory level is the first design choice. Figure 2 shows the effect of varying the block size upon the number of references not found in the buffer.[8,9] In order to suppress differences in magnitude caused by other variables, the data is normalized to the minimum miss rate for each study. For a fixed buffer capacity, the miss rate tends to first decrease as the block size increases and then increase sharply after the minimum is reached. The minimum occurs for blocks in the range of 16 to 256 bytes. The increase results from the block becoming so large that too few are contained by the buffer. If the buffer capacity increases to always contain the same number of blocks, the miss rate continues to decrease.

More significant, therefore, is the traffic between the buffer and the next level that results from the miss rate. Since an entire block must be moved for each reference not found in the buffer, this is the product of miss rate and block size, as shown in Figure 3. This is a function that always increases, and increases rapidly as the block size goes above 64 bytes.

Thus, larger blocks imply the need for larger buffers to maintain an adequate number of blocks, longer total block transfer time, and greater backing store bandwidth. Small blocks imply a larger expenditure in hierarchy control because of their number.

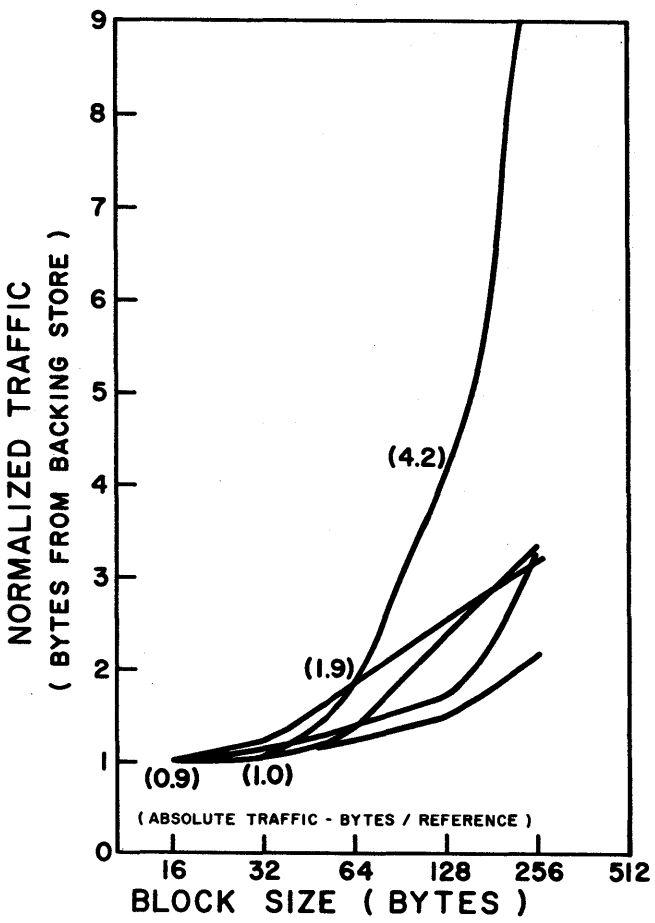


Figure 3—Effect of block size upon information required from backing store

Combining these data one selects an inner level block size of 32, 64 or 128 bytes for machines having a binary number of bytes per word. Results to data indicate that a 64 byte block is optimum.

In extending the structure to additional levels, the same considerations apply. Because an intermediate buffer is larger than an inner level, one prefers to use large blocks in it to minimize the cost of control. Since the miss rate is lower, the block size can be larger for the same traffic level. In addition, larger blocks tend to compensate for the longer access times. The use of pages of up to 4,096 bytes is common in time-sharing systems for disk or drum transfer blocks. This is much too large, however, for a three-level internal memory hierarchy. By analysis like that above, the block size for a third-level should be from one to eight second-level blocks. Preliminary results indicate that a 4:1 ratio (256 bytes at the third level) is best. With disk access times reduced by head-per-track designs and effective request queuing, the reduction of the external page size to 1024 bytes appears advantageous.

Capacity

Given a transfer block size, the next (and most important) design parameter is the buffer capacity. The percentage of memory references not found in the buffer is primarily determined by that capacity as shown in Figure 4. The various plots represent specific program traces; the limits include the effects of different algorithms. For a given capacity, block size and algorithm, there is a distribution of the references-not-found over a set of programs as shown in Figure 5. As the size of the buffer is increased, this distribution shifts to a lower average miss rate, and also becomes less program sensitive, i.e., sharper.

The choice of buffer capacity must be based upon performance and cost/performance analysis as discussed below. Based upon miss-rate alone, one would be unlikely to use a buffer of less than 8,192 bytes.

Since these data show that the buffer size does not

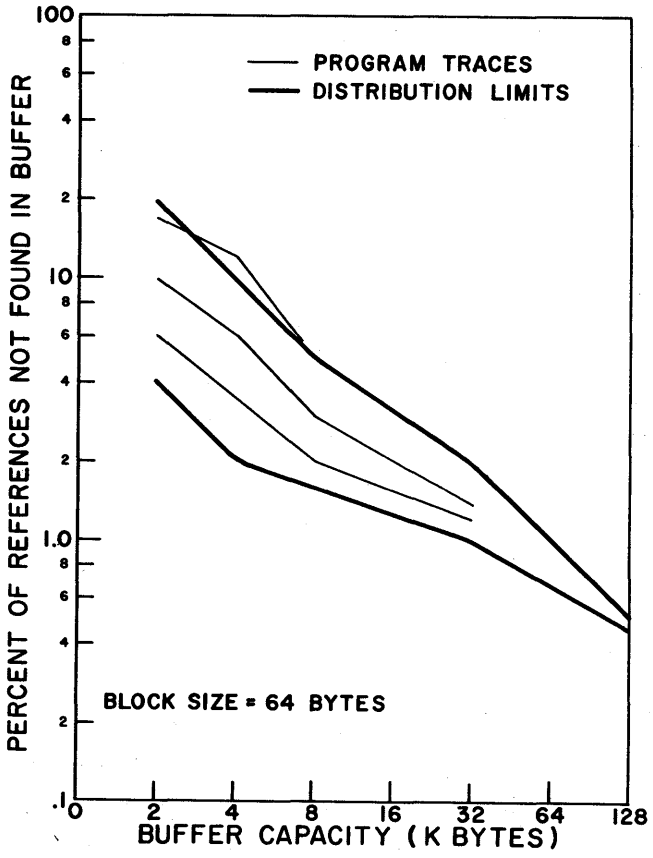


Figure 4—Effect of buffer capacity upon buffer miss rates

depend upon the total memory of a system, the advantage of hierarchy increases as that capacity increases. More powerful processors solve larger problems and execute more small programs concurrently; they require larger memories. The relationship between computational speed and typical memory capacity is shown in Figure 6 using the IBM 360 series as an example. Thus, the more powerful the system, the more effectively it can employ a hierarchy. As noted by Conti, in highest performance systems a hierarchy must be used to achieve the performance. Otherwise, the physical size of the massive memories causes long cable lengths and access time limiting system performance.

Conversely, in designing lesser systems, one eventually needs so little memory that system cost/performance cannot be improved by a hierarchy. Clearly, as the needed capacity approaches the buffer size, use of two levels is uneconomical.

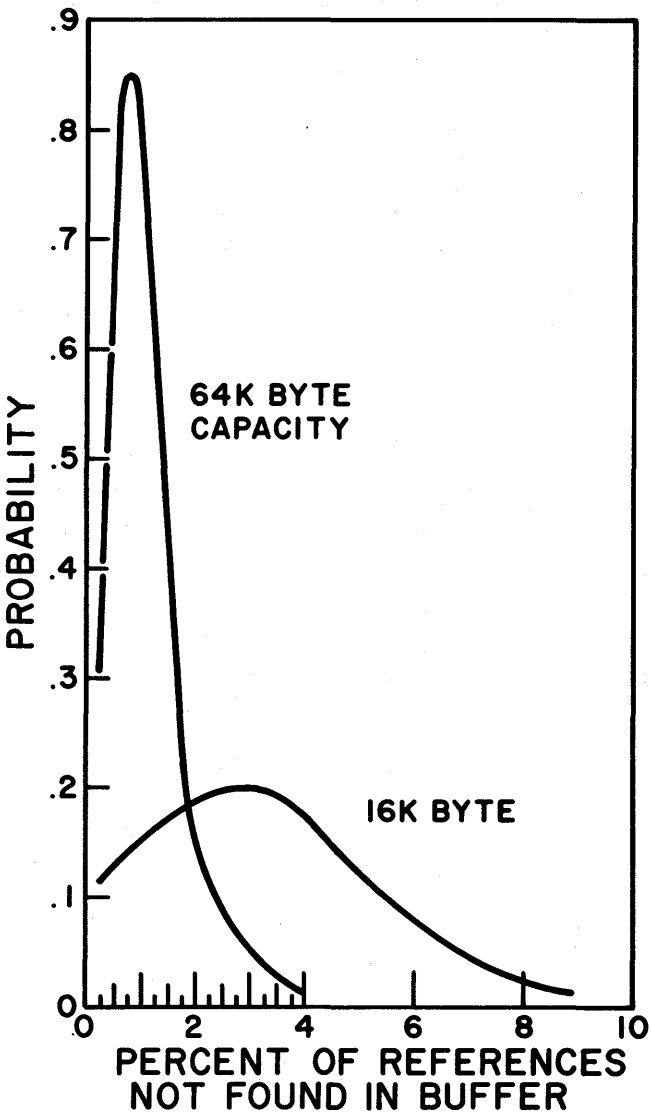


Figure 5—Distribution of references outside buffer over many programs

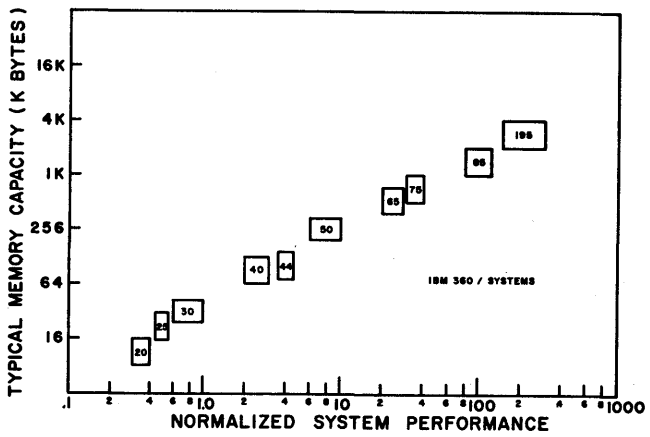


Figure 6—Main memory requirements as a function of processor power

In extending the memory hierarchy structure to multiple levels, the statistics of Figure 4 continue to apply. They must be corrected for the block size used, however. At each successive level the capacity must increase as the access time increases. The number of references not found in an intermediate level will be approximately the same as if that level were itself the inner level of memory in the system.

Algorithms

Each level in the hierarchy requires control algorithms to map the larger memory onto the smaller and to determine the area of the smaller that is to be overlaid. Mapping algorithms are all derived from two basic approaches.

The first is associative mapping in which a section of buffer is linked to any section of backing store by maintaining with the data the address of the backing-store block currently residing in the buffer block. The address storage for all blocks comprises a directory memory, all of which must be interrogated in order for any reference to be located. Unless relatively expensive high-speed parallel-search associative memory contains the directory, considerable time must be spent searching the memory. The associative function can also be implemented as a random access array of a size equal to the total number of blocks that can be addressed, as in the experimental 7044X system described by Brawn.[10] This is an expensive method, however.

The second basic algorithm is congruence mapping,

in which the binary address of the main store block is directly related to the corresponding buffer block address, by truncation. It results in a loss of performance due to swapping blocks between memory levels. This occurs because congruence is a simple homomorphism between the many members in a subset of all blocks in the backing store and a single block in the buffer. Consequently, different program entities, e.g., instructions and data, frequently conflict at a buffer location and must be exchanged.

The most useful techniques combine associativity and congruence. The 360/85, for example, associates "sectors" of 16 blocks each in the buffer and backing store. Blocks within the sector correspond uniquely, by congruence. The "set associative" algorithm described by Conti is a kind of inverse, in which the sets in the backing store are congruent to sets in the buffer but blocks within the sets are associatively linked.

When a data block not contained by a full buffer is needed, an algorithm must determine the area of the buffer that is to be overlaid. For a pure congruence mapping algorithm, this replacement algorithm is implicit. When any measure of associativity exists, however, a choice must be made. The basic alternatives include random selection and activity-weighting. A simple form of the latter is replacement of the least recently used block. Refining this approach to include measuring total block usage as well as how recently blocks were used assigns a higher value to information such as a supervisory routine which, although not recently used, has been executed many times in the program. These algorithms attempt to measure the value of each block and to displace the least valuable at any instant.

An ideal replacement algorithm can be defined as one that always replaces that block which will be used most distantly in the future. The practical methods discussed above are close to this ideal. Extreme sophistication is not profitable; even the random displacement method is not significantly inferior.

Semiconductor memory is important in implementing these controls as well as in creating the hierarchy. Associative arrays effectively perform the address mapping. A shift register set can implement least-recently-used replacement, which corresponds to a push-down list. Indeed, the entire paging control can be designed as a shifting associative tag memory.

Particular consideration must be given to the storing of processor-generated information into memory. Since the master data exists only at the outer level, this level must be updated. This can be done in parallel with writing into the buffer from the processor. This technique (store-thru) is effective when the backing memory is accessed by word, as in the 360/85, but can consume excessive time in a block oriented backing memory. Alternatively, a tag can be set for the buffered block to indicate that it has been modified and to control its transfer back to the master location when displaced. This method can present a problem in a multi-processor configuration or in the case of malfunction, in that the master copy does not immediately show the true status of the program. On the other hand, the unmodified base data can be employed in a retry or recovery procedure.

Similar considerations apply in block fetching. Since requested data is needed immediately, it is always profitable to fetch information from its current level into the inner level and simultaneously into processor registers. The fetching sequence can be ordered so that the word needed is the first member of the block to be fetched. However, the block being replaced must first be written into the higher level if it has been updated and store-thru is not used.

If the same block size were used at all levels one would never copy from higher levels into intermediate levels; rather the intermediate levels would fill by displacement from the inner level. This implies that the effective capacity of an intermediate level would be greater than its physical size. However, when larger

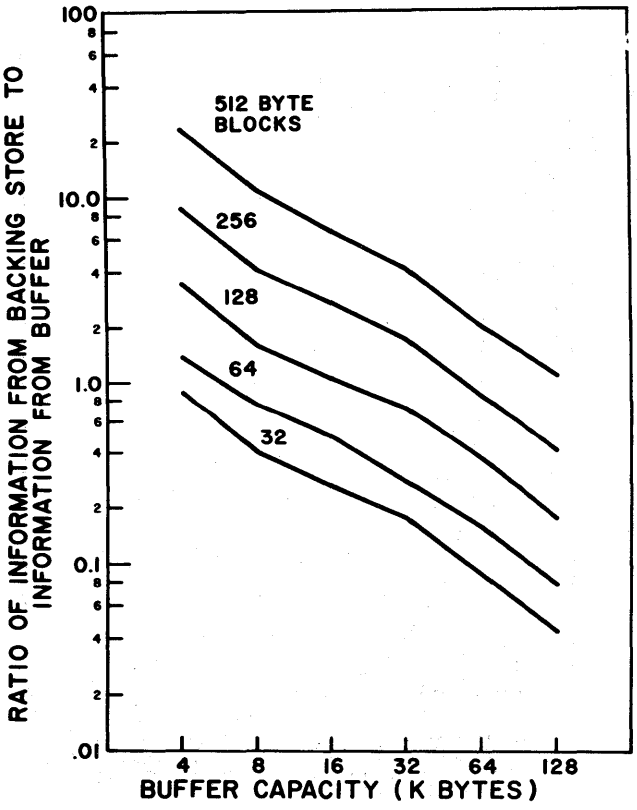


Figure 7—Information from backing store as a function of buffer capacity

blocks are used at outer levels it is better to transfer them into intermediate levels to avoid reaccessing the outer level for a subsequent working block contained within the specified transfer block.

Information transfer

Hierarchy reduces the amount of data required from the slower memories. The designer must, however, provide sufficient bandwidth at each level to insure that the access time rather than the time to transfer a block, determines performance. The average data requirements from a backing store are shown in Figure 7 as a ratio to those from the buffer. These are a function of the miss rate and hence, of the buffer capacity. The data rate from outer levels must also be adequate for access bursts, interference from input/output and program startup.

The more demanding requirement is that a block transfer be complete before a second block call is

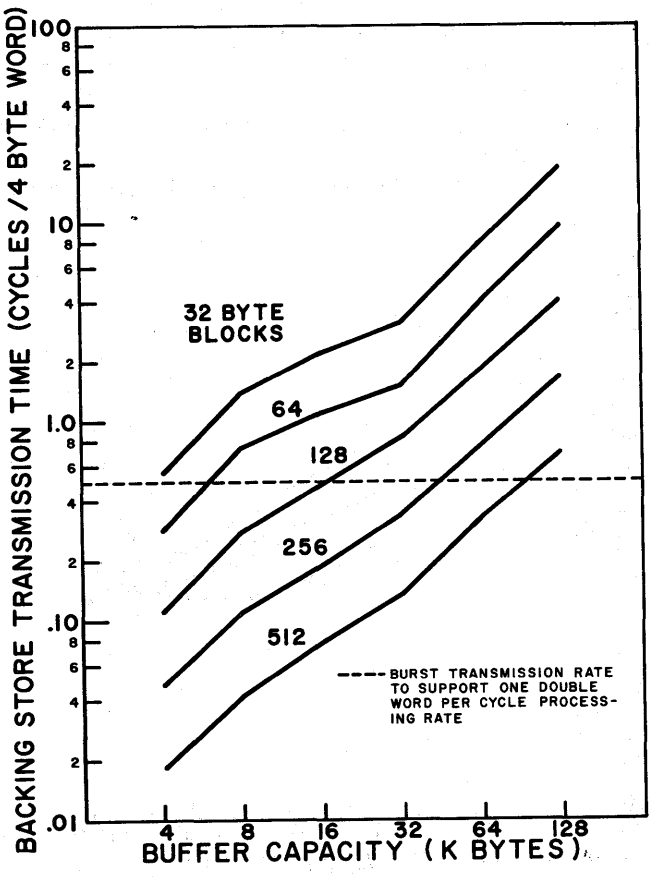


Figure 8—Allowable backing store transmission times

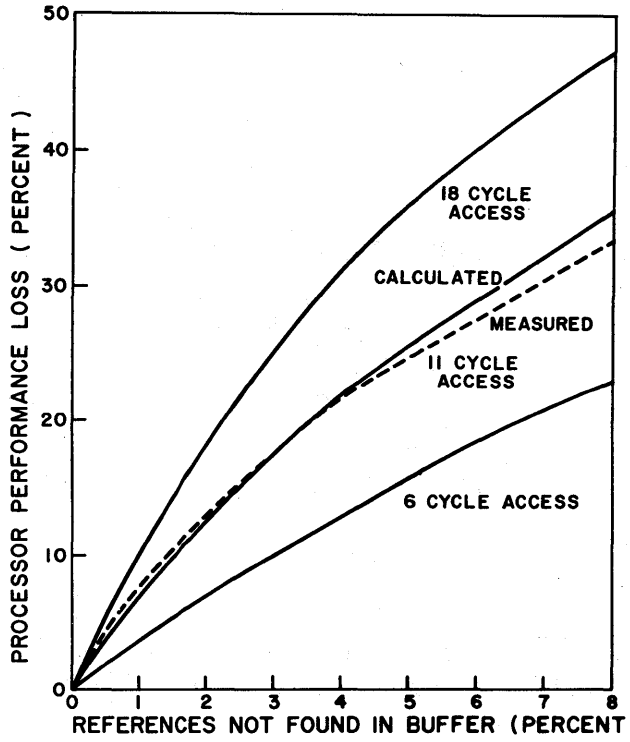


Figure 9—Effect of buffer miss rate upon processor performance

statistically expected. The time between block calls is a random variable, Poisson distributed, with a mean that is proportional to the average miss rate for the given buffer capacity, as shown in Figure 4. The allowable transfer times (in buffer cycles) are shown in Figure 8 as a function of buffer capacity and block size.

At each successive level, the basic information turnover rate as determined by the memory cycle is lower. The designer can compensate for this either be increasing the number of bits per memory cycle (word length) or by activating a larger number of memory units on each block reference (interleaving).

Performance

Rigorous performance prediction for a proposed system can be accomplished only by exercising its design over a representative program set by simulation. Sufficient data has been published, however, to permit reasonable estimates of performance to be made from the given design parameters.

If one knows the average number of memory references that fall outside the buffer, he can readily compute the corresponding time penalty, given the backing store access time. If the percentage of processor cycles that can generate memory references is also known, a complete estimate of processing time can be made. The total time T = T (processing) + T (buffer) + T (backing store). In making relative perfor-

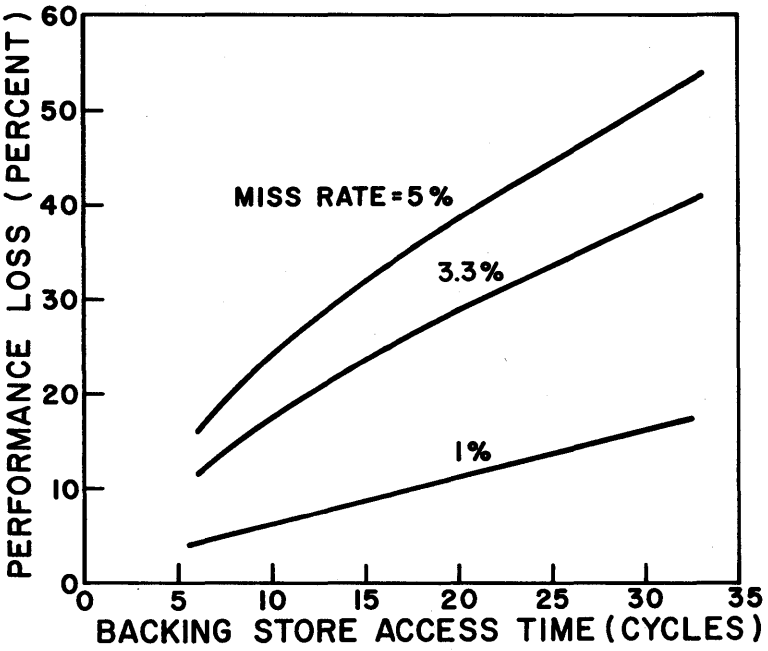


Figure 10—Effect of backing store cycle upon processor performance

mance estimates, the entire expression can be handled as a summation of the fractions of the total time spent in each activity.

Figure 9 uses this technique to show system performance in terms of degradation from what the system would give if the buffer provided the total memory capacity, as a function of the percentage of references not found in the buffer. The data given by Liptay for the 360/85 are shown for comparison. In the case of main memory access time of eleven cycles, it was assumed that for every one-hundred memory references there were 42 processor-only cycles. The mean-time between calls to the backing store was taken as greater than the block transfer time. These data can also show the effect upon performance of varying the backing store access time (expressed as a multiple of the processor cycles) for various miss rates, as in Figure 10.

This approach is particularly useful in comparing memory system alternatives. Using a reference system it can be translated into absolute performance if desired. However, it relates entirely to processor computational power, not system throughput.

In predicting performance when another level is added, we note that an intermediate level acts as a backing store to an inner level and as a buffer to an outer level. Based upon the number of references not found in that level, a time premium due to accessing the next level can be added. For a given configuration and set of access times, total running time can be calculated as indicated above, using the data found in Figure 4.

No experimental data has yet been published to validate such estimates of multiple-level memory hierarchy performance; no such systems are known to have been constructed. Predictions based upon simulation require accumulation of a data base of representative programs; in order to be convincing, these must include system programs and others large enough to use the available address space and to overflow intermediate members of the hierarchy. Purely theoretical methods of performance prediction have been incomplete because, as yet, no one has adequately characterized the parameters of the program process.

Cost/performance

Unless marketing needs force a specific cost or performance target, the designer's objective is to minimize the cost/performance ratio for the system within a general area of performance. In the case of designing the memory system he may have to compare sets of one, two and three level designs. The basis for cost/performance comparisons must be the performance estimates discussed above. The costs must include all of the memory, processor and control costs—not merely raw memory cost—to arrive at a properly balanced design.

For the two-level hierarchy, Figure 11 illustrates the

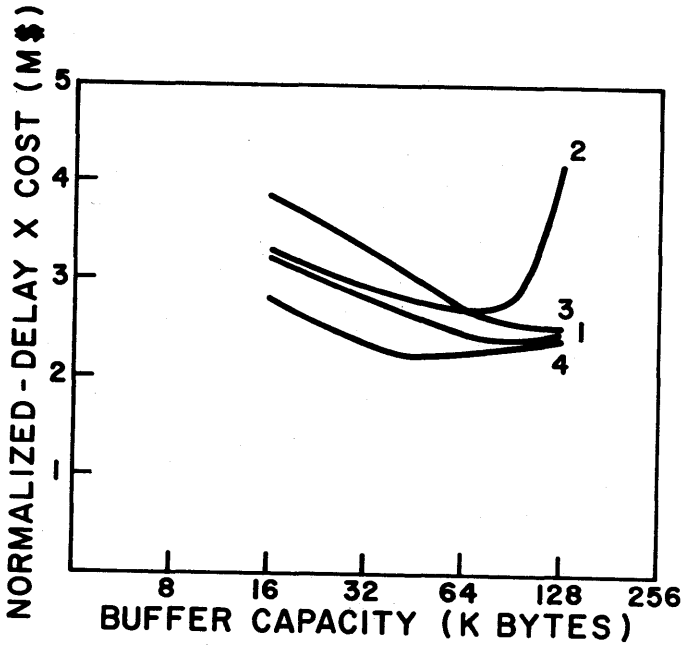


Figure 11—Cost/performance analyses of two-level hierarchy examples

analysis to be made. Using appropriate costs for the system elements, we plot the relative delay-cost product as a function of the buffer capacity. This is proper because the backing store capacity is fixed by the application. The most effective system is that in which the delay-cost product is least. Curve (1) of Figure 11 is plotted for arbitrary assumptions including: buffer cost = \$.25 per bit, two million byte main memory cost = \$45,000 with access equal to 33 cycles, high-performance processor cost = \$900,000, processor cycles = 32 percent. For these assumptions a buffer capacity of 96K bytes is most effective. It is large because of the long main-memory access time.

In order to illustrate the effect of varying these assumptions, the following curves are also shown:

2. first level buffer costs twice as high (\$.50)
3. main memory access longer (50 cycles)
4. miss ratio improved (lowered) by a factor of two for each capacity.

Some qualitative rules for optimizing memory system cost/performance are apparent from these analyses:

1. as buffer memory is relatively more expensive less should be used;
2. as main memory is relatively slower more buffer should be used;
3. as algorithms yield a lower miss rate less buffer should be used.

The converses also apply.

In order to assess the utility of a three-level hierarchy one must first evaluate the two-level alternatives. To find the most favorable three-level configuration we must consider a range of capacities for each buffer level. Figure 12 shows how cost-performance values for the three-level alternatives can be displayed as a function of first-level buffer capacity for comparison with the two-level situation.

Conditions that are favorable to the use of a three-level configuration include:

1. expensive first level technology
2. steep cost/performance curve for main memory technology
3. relatively high miss ratios
4. large total memory requirements

An optimum three-level configuration will use less first-level and more second-level buffer memory than the equivalent two-level case. The two-level configuration is more generally applicable, until a lower cost bulk memory is available.

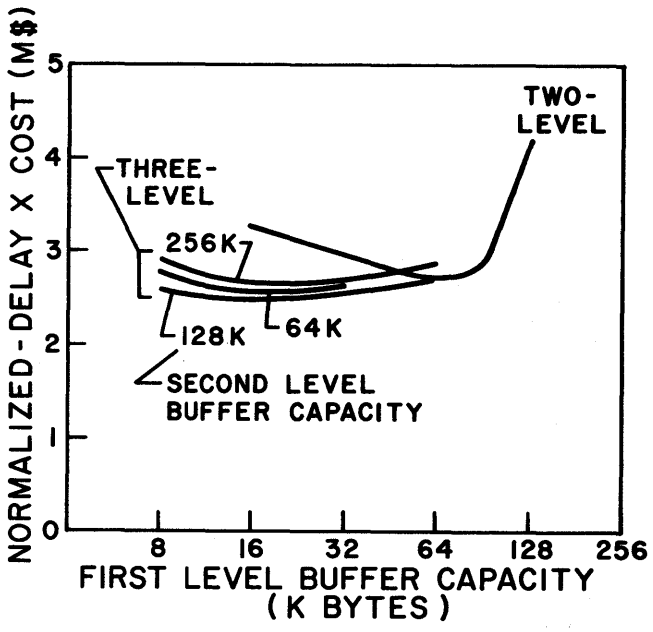


Figure 12—Cost/performance analyses of three-level hierarchy examples

DISCUSSION

A properly designed memory hierarchy magnifies the apparent performance/cost ratio of the memory system. For example, the first case assumed in Figure 11 shows a cost/performance advantage of five times that of a plausible single-level memory system with a three-cycle access costing \$.15 per bit. The combination achieves the capacity of the outer level at a performance only slightly less than that of the inner level. Because of the substantial difference in the capacities, the total cost is not greatly more than that of the outer level alone.

The early memory hierarchy designs attempted to integrate the speed/cost characteristics of electronic and electromechanical storage. Now the large performance loss could be predicted from the relatively enormous access time of the rotating device. For example, degradation of more than 100 times over operation from entirely within two-microsecond memory would occur with addresses generated every two microseconds, 64K byte buffer (core) capacity, 512 word block size, and 8ms average drum access time. To compensate for such disparity in access time, the inner memory must contain nearly complete programs.

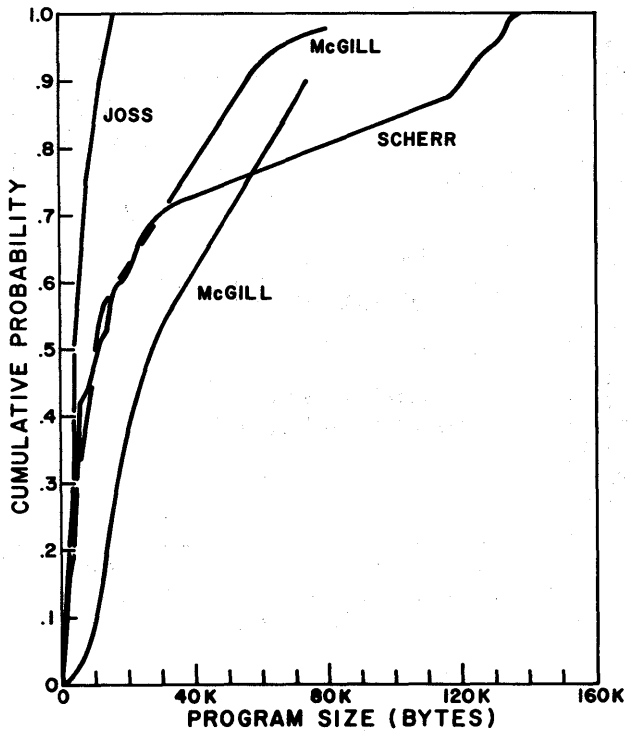


Figure 13—Distribution of program size

Successful time-sharing systems essentially do this. Figure 13 shows the results of several studies[13,14,15] of the distribution of their program size.

These time-sharing systems also indicate direction toward the use of multiple level internal memory. In particular, they show the need for low-cost medium-access bulk memory. They are caught between inadequate response time achieved paging from drum or disk and prohibitive cost in providing enough of present large capacity core memories (LCM). However, designers such as Freeman[11] and MacDougall[12] have stated that only by investment in such LCM can systems as powerful as the 360/75 have page accessibility adequate to balance system cost/performance. Freeman's design associates the LCM with a backing disk, as a pseudo-disk system.

Transparent hierarchy can make it easier to connect systems into multiprocessing configurations, with only the outer level common. This minimizes interference at the common memory, and delays due to cable length and switching. It has no direct effect on associated software problems.

To date, hierarchy has been used only in the main (program) memory system. The concept is also powerful in control memories used for microprogram storage. There it provides the advantages of writeable control memory, while allowing the microprograms to reside in inexpensive read-only or non-volatile read-write memory.

A primary business reason for using hierarchy is to permit continued use of ferrite memories in large systems. With a buffer to improve system performance, ferrites can be used in a lowest cost design. It is unnecessary to develop ferrite or other magnetic memories at costly, high performance levels.

The use of multiple levels also removes the need to develop memories with delicately balanced cost/performance goals. Rather, independent efforts can aim toward fast buffer memories and inexpensive large capacity memories. This permits effective use of resources and implies higher probability of success.

Systems research in the near future should concentrate upon better characterization of existing systems and programs. There is still little published data that describes systems in terms of their significant statistical characteristics. This is particularly true with respect to the patterns of information scanning that are now buried under the channel operations required to exchange internal and external data. Only from analysis and publication of program statistics and accompanying machine performance data will we gain the insight needed to improve system structure significantly.

REFERENCES

1 C J CONTI
Concepts for buffer storage
IEEE Computer Group News Vol 2 No 8 March 1969

2 C J CONTI D H GIBSON S H PITKOWSKY
Structural aspects of the system/360—Model 85, I.—General organization
IBM Systems Journal 7 1 1968

3 J S LIPTAY
Structural aspects of the system 360 Model 85, II—The cache
IBM Systems Journal 7 1 1968

4 T KILBURN
Electronic Digital Computing Machine
Patent 3,248,702

5 T KILBURN D B G EDWARDS M J LANIGAN F H SUMMER
One-level storage system
IRE Transactions on Electronic Computers Vol 11 No 2 1962 pp 223–235

6 D W ANDERSON F J SPARACIO R M TOMASULO
The IBM System/360 Model 91: Machine philosophy and instruction handling
IBM Journal Vol 11 No 8 1967

7 L BLOOM M COHEN S PORTER
Considerations in the design of a computer with a high logic-to-memory speed ratio
Proc of Sessions on Gigacycle Computing Systems AIEE Winter General Meeting January 1962

8 D H GIBSON
Considerations in block oriented systems design
AFIPS Proceedings Vol 30 SJCC 1967 pp 75-80
9 S S SISSON M J FLYNN
Addressing patterns and memory handling algorithms
AFIPS Proceedings Vol 33 FJCC 1968 pp 957-967
10 B S BRAWN F G GUSTAVSEN
Program behavior in a paging environment
AFIPS Proceedings Vol 33 FJCC 1968 pp 1019-1032
11 D N FREEMAN
A storage hierarchy system for batch processing
AFIPS Proceedings Vol 32 SJCC 1968 p 229
12 M H MacDOUGALL
Simulation of an ECS-based operating system
AFIPS Proceedings Vol 30 SJCC 1967 p 735
13 A L SCHERR
Time-sharing measurement
Datamation Vol 12 No 4 April 1966 pp 22–26
14 I F FREIBERGS
The dynamic behavior of programs
AFIPS Proceedings Vol 33 FJCC 1968 pp 1163-1167
15 G E BRYANT
JOSS—A statistical summary
AFIPS Proceedings Vol 31 FJCC 1967 pp 769-777

Design of a very large storage system*

by SAMUEL J. PENNY, ROBERT FINK, and MARGARET ALSTON-GARNJOST

University of California
Berkeley, California

INTRODUCTION

The Mass Storage System (MSS) is a data-management system for the on-line storage and retrieval of very large amounts of permanent data. The MSS uses an IBM 1360 photo-digital storage system (called the chipstore) with an on-line capacity of 3×10^{11} bits as its data storage and retrieval equipment. It also uses a CDC 854 disk pack for the storage of control tables and indices. Both these devices are attached to a CDC 6600 digital computer at the Lawrence Radiation Laboratory—Berkeley.

Plans for the MSS began in 1963 with a search for an alternative to magnetic tape as data storage for analyses in the field of high energy physics. A contract was signed with IBM in 1965 for the chipstore, and it was delivered in March of 1968. The associated software on the 6600 was designed, produced, and tested by LRL personnel, and the Mass Storage System was made available as a production facility in July of 1969.

This paper is concerned with the design effort that was made in developing the Mass Storage System. The important design decisions, and some of the reasons behind those decisions, are discussed. Brief descriptions of the hardware and software illustrate the final result of this effort.

CHOICE OF THE HARDWARE

By 1963 the analysis of nuclear particle interactions had become a very large application on the digital computers at LRL—Berkeley. More than half the available time on the IBM 7094 computer was being used for this analysis, and the effort was expanding. Much of the problem was purely data manipulation—sorting, merging, scanning, and indexing large tape files—and single experiments produced tape libraries of hundreds of reels each.

The problems of handling large tape libraries had become well known to the experimenters. Tapes were lost; they developed bad spots; the wrong tapes were used; keeping track of what data were on what tape became a major effort. All these problems degraded the quality of the data and made the experiments more expensive. A definite need existed for a new approach.

The study of the problem began with establishment of a set of criteria for a large-capacity on-line storage device, and members of the LRL staff started investigating commerically available equipment. The basic criteria used were:

a. The storage device should be on-line to the central computing facility.
b. It should have an on-line capacity of at least 2.5×10^{11} bits (equivalent to 2000 reels of tape).
c. Access time to data in the storage device should be no more than a few seconds.
d. The data-reading transfer rate should be at least as fast as magnetic tape.
e. The device should have random-access capability.
f. The storage medium of the device should be of archival quality, lasting 5 years at least.
g. The storage medium need not be rewritable.
h. The frequency of unrecoverable read errors should be much lower than on magnetic tape.
i. Data should be easily movable between the on-line storage device and shelf storage.
j. The device hardware should be reliable and not subject to excessive failures and down time.
k. Finally, the storage device should be economically worthwhile and within our budget.

Several devices were proposed to the Laboratory by various vendors. After careful study, including computer simulation of the hardware and scientific evaluations of

* Work done under auspices of the U.S. Atomic Energy Commission.

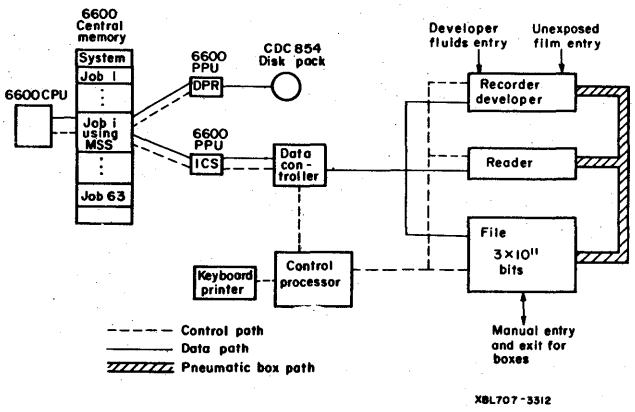


Figure 1—General MSS architecture

the technologies, the decision was made to enter into a contract with IBM for delivery, in fiscal year 1968, of the 1360 photo-digital storage system. This contract was signed in June of 1965. The major application contemplated at that time is described in Ref. 1.

It was clear that one of the major problems in the design of the associated software would be the storage and maintenance of control tables and indices to the data. Unless indexing was handled automatically by the software, the storage system would quickly become more of a problem than it was worth. Protection of the indices was seen to be equally important, for the system would be dependent on them to physically locate the data. It was decided that a magnetic disk pack drive, with its removable pack, was the most suitable device for the storage of the MSS tables and indices.

A CDC 854 disk pack drive was purchased for this purpose.

DESCRIPTION OF THE HARDWARE

1360 Photo-digital storage system

The IBM 1360 chipstore is an input-output device composed of a storage file containing 2250 boxes of silver halide film chips, a chip recorder-developer, and a chip reader. Figure 1 shows the general arrangement of the chipstore hardware and its relation to the CDC 6600 computer. References 2 through 5 describe the hardware in detail. A brief summary is given below.

A chip is 35 by 70 mm in size and holds 4.7 million bits of data as well as addressing and error-correction or error-detection codes. Data from the 6600 computer are recorded on the chip in a vacuum with an electron beam, taking about 18 sec per chip. The automatic film developer unit completes the processing of a chip within 2.5 min; it overlaps the developing of eight chips so that its processing rate is comparable to that of the recorder.

Up to 32 chips are stored together in a plastic box. Figure 2 shows a recorded film chip and the box in which it is kept. These boxes are transported between the recorder-developer, the box storage file, and the chip reader station by means of an air blower system. Transport times between modules on the Berkeley system average around 3 sec.

Under the command of the 6600 computer the chipstore transports a box from the storage file to the reader, picks out a chip, and positions it for reading. The chip is read with a spot of light generated by a cathode-ray tube and detected by a photomultiplier tube at an effective data rate of 2 million bits per second. The error correction-detection codes are checked for validity as the data are read, and if the data are incorrect, an extensive reread and error-correction scheme is used to try to reproduce the correct data. The data are then sent to the 6600 across a high-speed data channel. Chip pick and store times are less than 0.5 sec.

The box storage file on the Berkeley 1360 system has a capacity of 2250 boxes. This represents an on-line data capacity of 2750 full reels of magnetic tape (at 800 BPI); 1360 systems at other sites have additional file modules, giving them an on-line capacity three or more times as great as at Berkeley.

A manual entry station on the chipstore allows boxes of chips to be taken out of the system or to be reinserted. By keeping the currently unused data in off-line storage and retaining only the active data in the file, the potential size of the data base that can be built in the MSS is equivalent to tens of thousands of magnetic tapes.

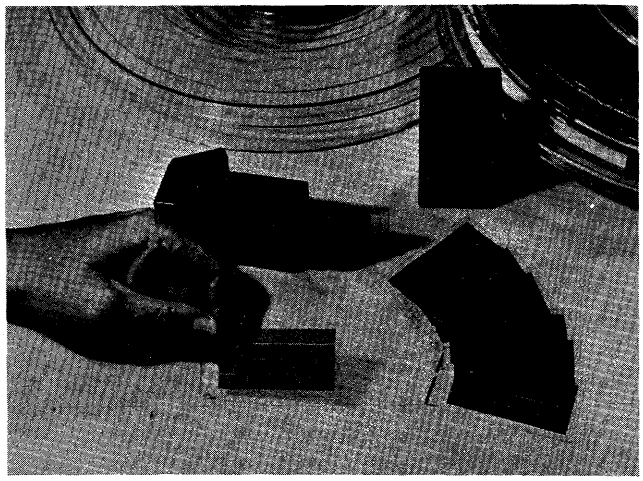


Figure 2—Recorded film chips and storage box

A process control computer is built into the chipstore hardware. This small computer is responsible for controlling all hardware actions as well as diagnosing malfunctions. It also does the detailed scheduling of events on the device. Communication between the chipstore and the host computer goes through this processor. This relieves the host of the responsibility of commanding the hardware in detail, and offers a great deal of flexibility.

854 Disk pack drive

The CDC 854 disk pack drive holds a removable 10-surface disk pack. The pack has a typical access time of 90 msec, and a data transfer rate of about 1 million bits per sec. Its storage capacity is 48 million bits.

MSS uses this pack for the storage of all its tables and indices to the data that have been written into the 1360 chipstore. A disk pack was chosen for this function to insure the integrity of the MSS tables. The 854 has a proven record of hardware and data reliability. Also, since the pack is removable, the drive can be repaired and serviced without threat to the tables.

6600 Computer complex

The chipstore is connected to one of the CDC 6600 computers at LRL through a high-speed data channel. The 6600 computer has 131072 words of 60-bit central core memory (CM), a central processor unit (CPU) operating at a 100-nsec cycle rate, and 10 peripheral processor units (PPU). Each PPU contains 4096 words of 12-bit core memory and operates at a 1-μsec cycle rate. The PPUs control the data channel connections to the external input-output equipment and act as the interface between jobs residing in CM and the external world.

The operating system on the 6600 is multiprogrammed to allow several jobs to reside in CM at once and share the use of the CPU. Two of the PPUs act as the system monitor and operator interface for the system, and those remaining are available to process task requests from the monitor and execute jobs. The MSS, composed of both CPU and PPU programs, has been built as a subsystem to this operating system.

CHOICE OF THE MASS STORAGE SYSTEM SOFTWARE

Design objectives

Having made the commitment on hardware, the Laboratory was faced with designing and implementing the associated software. The basic problem was to produce a software system on the CDC 6600 computer that, using the IBM 1360 chipstore, would lead to the greatest increase in the productive capacity of scientists at the Laboratory. In addition, it was necessary that the system be one that the scientists would accept and use, and to which they would be willing to entrust their data. It would be required to be of modular design and "open-ended," allowing expansion and adjustment to new techniques that the scientists might develop for their data analysis.

Overall study of the problem yielded three primary objectives. Most important was to increase the reliability of the data storage, both by reducing the number of data-read errors and by protecting the data from being lost or destroyed; much time and effort could be saved if this objective were met. The second objective was to increase the utilization of the whole computer complex. The third was to provide facilities for new, more efficient approaches to data analysis in the future.

The problem was divided into three technical design areas: the interaction between the software and the hardware, the interaction between the user and the software, and the structure of the stored data.

In the area of software-hardware interaction, the design objectives were to maximize protection of the user data, interleave the actions for several jobs on the hardware, reduce the need for operator intervention, and realize maximum utilization of the hardware. This was the approximate order of importance.

Objectives in the area of user interaction with the MSS included making that interaction easy for the user, offering him a flexible data-read capability, and supplying him with a protected environment for his data. Ease of data manipulation was of high value, but not at the expense of data protection. A flexible read mechanism was necessary, since if the users could not read their data from the MSS, they would seek other devices. This flexibility was to include reading data from the chipstore at rates up to its hardware limit, having random access to the data under user control, possibly intermixing data from the chipstore, magnetic tapes, and system disk files, and being able to read volumes of data ranging in size from a single word to the equivalent of many reels of tape.

The problem of data structures for the MSS was primarily one of finding a framework into which existing data could be formatted and which met the requirements of system and user interaction. This included the ability to handle variable-length data records and files and to access these data in a random fashion. It was decided that a provision to let the user reference his data by name and to let the system dynamically allocate storage space was very important. It was also important

to have flexible on-line-off-line data-transfer facility so that inactive data could be moved out of the way.

Software design decisions

Several important design decisions were made that have had a strong effect on the nature of the final system. Some of these decisions are listed here.

Each box used for data storage is given a unique identification number, and this number appears on a label attached to the box. A film chip containing data is given a unique home address, consisting of the identification number of the box in which it is to reside and the slot in that box where it is to be kept. Control words written at the beginning of the chip and at various places throughout the data contain this address (along with the location of the control word on the chip), and this information can be checked by the system to guarantee correct positioning for retrieval of the data. It is also used to aid in recovery procedures for identifying boxes and chips. This control information can be used to help reconstruct the MSS tables if they are destroyed.

The control words are written in context with the data to define the record and file structure of the data on the chips. The user is allowed to give the address of any control word (such as the one at the beginning of a record) to specify what data are to be read. This scheme meets the design objective of allowing random access to data in the chipstore.

Data to be written into the chipstore are effectively staged. The user must have prepared the data he wishes to be recorded in the record and file structure he desires in some prior operation. He then initiates the execution of a system function that puts the source data into chip format, causes its recording on film chips, waits for the chips to be developed, does a read check of the data, and then updates the MSS tables.

Data read from the chipstore are normally sent directly to the user's program, though system utility functions are provided for copying data from the chipstore to tape or disk. If the user desires, he may include a system read subroutine with his object program that will take data directly from the chipstore and supply them to his executing program. This method was chosen to meet the objectives of high data-transfer rates and to provide the ability to read gigantic files of data.

To aid the user in the access and management of his data in the MSS, it was decided to create a data-management control language oriented to applications on the chipstore. A user can label his data with names of his own choosing and reference the data by those names. A two-level hierarchy of identification is used, that of data set and subset. The data set is a collection of named subsets, in which each subset is some structure of user data. The control language is not limited to manipulating only data from the chipstore; it can also be used to work with magnetic tape or system disk files.

Two more decisions have greatly simplified the overall problem of data management in the MSS. The first was to allocate most of the on-line storage space on the chipstore in blocks to the scientists engaged in data analysis of current experiments, and give them the responsibility of choosing which of their data are to reside on-line within their block and which are to be moved off-line. The second decision was to treat all as permanent. Once successfully written, film chips are never physically destroyed. At most, the user may delete his reference to the data, and the chips are moved off-line.

DESCRIPTION OF THE MSS SOFTWARE

The system in use on the 6600 computer for utilizing the chipstore results both from design effort at the beginning of the project and from experience gained during the implementation and initial production phases. Its essential features are listed below.

Indexing and control of the data stored in the chipstore are handled through five tables kept on the disk pack, as follows.

The box group allocation table controls the allocation of on-line storage space to the various scientists or experiments at the Laboratory. Any attempt by a user to expand the amount of on-line space in use by his box group above its allowable limit will cause his job to be aborted.

The box identification table contains an entry for each uniquely numbered box containing user data chips. An entry tells which box group owns the box, where that box is stored (on-line or off-line), which chip slots are used in the box, and the date of its last use.

The file position table describes the current contents of the 1360 file module, defines the use of each pocket in the file, and gives the identification number of the box stored in it.

The data set table contains an entry for each of the named collections of data stored in the chipstore. Status and accounting information is kept with each data-set table entry. Each active entry also points to the list of subsets collected under that data set.

The subset list table contains the lists of named subsets belonging to the entries in the data set table. A subset entry in a list gives the name of the subset, the address of the data making up that subset, and status information about the subset.

These tables are accessed through a special PPU task processor program called DPR. This processor reads or writes the entries in the tables as directed. However, if the tables are to be written, special checks and procedures are used to aid in their protection. Twice daily the entire contents of the MSS disk pack are copied onto magnetic tape. This is backup in case the data on the pack are lost.

All communication to the chipstore across the data channel link is handled through another PPU task processor program called 1CS; 1CS is multiprogrammed so that it can be servicing more than one job at a time. Part of its responsibility is to schedule the requests of the various user jobs to make most effective use of the system. For instance, jobs requiring a small amount of data are allowed to interrupt long read jobs. Algorithms for overlapping box moving, chip reading, and chip writing are also used to make more effective use of the hardware.

1CS and DPR act as task processors for jobs residing in the central memory of the 6600. The jobs use the MSSREAD subroutine (to read from the chipstore) or the COPYMSS system utility to interface to these task processors. These central memory codes are described below.

The reading of data from the chipstore to a job in central memory is handled by a system subroutine called MSSREAD. The addresses of the data to be read and how the data are to be transmitted are given to MSSREAD in a data-definition file. This file is prepared prior to the use of MSSREAD by the COPYMSS program described later. MSSREAD handles the reading of data from magnetic tape, from disk files, or from the chipstore. If the data address is the name of a tape or disk file, MSSREAD requests a PPU to perform the input of the data from the device a record at a time. If the address is for data recorded in the chipstore, it connects to 1CS, and working with that PPU code, takes data from the chipstore, decodes the in-context structure, and supplies the data to the calling program.

A system program called COPYMSS is responsible for supplying the user with four of the more common functions in MSS. It processes the MSS data-management control language to construct the data-definition file for MSSREAD. It performs simple operations of copying data from the chipstore to tape or disk files. It prepares reports for a user, listing the status of his data sets and subsets. Finally, COPYMSS is the program that writes the data onto film chips in the chipstore.

To write data to the chipstore, the user must prepare his data in the record and file structure he desires. He then uses the MSS control language to tell COPYMSS what the data set and subset names of the data are to be and where the data can be found. COPYMSS inserts the required control words as the data are sent through 1CS to the chipstore to be recorded on film chips. After the chips have been developed, 1CS rereads the data to verify that each chip is good. If a chip is not recorded properly, it is discarded and the same data are written onto a new chip. When all data have been successfully recorded and the chips are stored in the home positions, COPYMSS uses DPR to update the disk pack tables, noting the existence of the new data set—subset.

The remaining parts of the MSS software include accounting procedures, recovery programs, and programs to control the transfer of data between on-line and off-line storage. These programs, used by the computer operations group, are not available to the general user.

TABLE I—Distribution of MSS Implementation Effort.

Operation	Man-years
Procurement and Evaluation	1.0
System design	2.8
Software coding	1.7
Software checkout	0.8
Maintenance, documentation, etc.	1.2

TABLE II—MSS Usage Per Week.

Number of read jobs	250
Number of write jobs	100
Chips read	11500
Bits read	5.4×10^{10}
Unrecoverable read errors	15
Chips written	1900
Percentage down time	8.5

RESULTS AND CONCLUSIONS

Effort

A total of about 7.5 man-years of work was invested in the Mass Storage System at LRL—Berkeley. The staff on the project was composed of the authors with some help from other programmers in the Mathematics and Computing Department. The breakdown of this effort is shown in Table I.

Operating experience

The Mass Storage System has been in production status since June 1969. Initial reaction of most of the

TABLE III—Comparison of Storage Devices at LRL—Berkeley

	MSS	CDC 607 tape drive	CDC 854 disk pack	IBM 2311 data cell	CDC 6603 system disk
On-line capacity (bits/device)	3.3×10^{11}	1.2×10^{8}	4.8×10^{7}	3.0×10^{9}	4.5×10^{8}
Equivalent reels of tape	2750	1	0.4	25	3.75
Cost of removable unit	\$13/box	\$20/reel	\$500/pack	\$500/cell	—
Storage medium cost (¢/10^3 bits)	0.008	0.017	1.0	0.17	—
Average random access (sec)	3	(minutes)	0.075	0.6	0.125
Maximum transfer rate (kilobits/sec)	2000	720	1330	450	3750
Effective transfer rate[a]	1100	500	—	200	400
Approximate capital costs (thousands of dollars)	1000	100	35	220	220
Mean error-free burst length (bits)	1.6×10^{9}	2.5×10^{7}	$>10^{10}$	10^{9}	$>10^{10}$

[a] Based on usage at LRL—Berkeley; the rates given include device-positioning time.

users was guarded, and many potential users were slow in converting to its use. As a result, usage was only about 2 hours a day for the first 3 months. Soon after, this level started to increase, and at the end of one year of production usage a typical week (in the month of June 1970) showed the usage given in Table II.

Most of the reading from the chipstore is of a serial nature, though the use of the random-access capability is increasing. Proportionally more random access activity is expected in the future as users become more aware of its possibilities.

A comparison of the MSS with other data-storage systems at the Laboratory, shown in Table III, points out the reasons for the increased usage. For large volumes of data, the closest competitor is magnetic tape (assumed here to be full 2400-foot reels, seven-track, recorded at 800 BPI).

The values shown in Table III are based on the following assumptions: on-line capacities are based on having a single unit (e.g., a single tape drive); capital costs are not included in the storage medium costs; effective transfer rates are based on usage at LRL, and are very low for the system disk because all jobs are competing for its use; and all costs given are only approximate.

The average data-transfer rate on long read jobs (involving many chips and many boxes) is more than one million bits per second. This is decidedly better than magnetic tape. Short reads go much faster than from tape once the 3-sec access time is complete.

The biggest selling point for the Mass Storage System has been the extremely low data-error rate on reads. This rate is less than 1/60 of the error rate on magnetic tape. The second most important point has been the potential size of the data files stored in the chipstore. Several data bases of from 20 to 200 boxes of data have been constructed. Users find that having all their data on-line to the computer and not having to rely on the operators to hang tapes is a great advantage. Their jobs run faster and there is less chance that they will not run correctly.

The cost of storing data on the chipstore has proven to be competitive with magnetic tape, especially for short files or for files that will be read a number of times. Users are beginning to find it profitable to store their high-use temporary files on the chipstore.

The system has not been without its difficulties. Hardware reliability has at times been an agonizing problem, but as usage increases and the engineers gain more experience on the hardware, the down time for the system has decreased significantly. We now feel that 5 percent down time would be acceptable, though less would be preferable. Fortunately, lack of hardware reliability has not affected the data reliability.

CONCLUSIONS

Though intended primarily as a replacement for magnetic tape in certain applications, the MSS has shown other benefits and capabilities. Data reliability is many times better than for magnetic tape. Some applications requiring error-free storage of large amounts of data simply are not practical with magnetic tape, but they become practical on the chipstore. The nominal read rate is faster than that of magnetic tape for long serial files. In addition, any portion of a file is randomly accessible in a time ranging from a few milliseconds to 5 seconds.

The MSS is not without its limitations and problems. The 1360 is a limited-production device: only five have been built. It uses technologies within the state of the art but not thoroughly tested by long experience.

Keeping the system down time below reasonable limits is a continuing and exacting effort. Development of both hardware and software has been expensive. The software was a problem because the chipstore was a new device and people had no experience with such large storage systems.

The Mass Storage System has met its purpose of increasing the productive capacity of scientists at the Laboratory. It has also brought with it a new set of problems, as well as a new set of possibilities. The biggest problem is how to live with a system of such large capacity, for as more and more data are entrusted to the chipstore, the potential loss in case of total failure increases rapidly. The MSS offers its users important facilities not previously available to them. More important, the age of the very large Mass Store has been entered. In the future, the MSS will become an important tool in the computing industry.

REFERENCES

1 M H ALSTON S J PENNY
The use of a large photodigital mass store for bubble chamber analysis
IEEE Trans Nucl Sci Volume NS-12 4 pp 160-163 1965

2 J D KUEHLER H R KERBY
A photo-digital mass storage system
AFIPS Conference Proceedings of the Fall Joint Computer Conference Volume 29 pp 735-742 1966

3 L B OLDHAM R T CHIEN D T TANG
Error detection and correction in a photo-digital storage system
IBM J Res Develop Volume 12 6 pp 422-430 1968

4 D P GUSTLIN D D PRENTICE
Dynamic recovery techniques guarantee system reliability
AFIPS Conference Proceedings of the Fall Joint Computer Conference Part II Volume 33 pp 1389-1397 1968

5 R M FURMAN
IBM 1360 photo-digital storage system
IBM Technical Report TR 02.427 May 15 1968

Design of a megabit semiconductor memory system

by D. LUND, C. A. ALLEN, S. R. ANDERSEN and G. K. TU

Cogar Corporation
Wappingers Falls, New York

INTRODUCTION

This paper describes a 32,768 word by 36 bit word Read/Write Memory System with an access time of 250ns, and a cycle time of 400ns.

The memory system is based on MOS technology for the storage array and bipolar technology for the interface electronics. A functionally designed storage array chip with internal decoding minimizes the number of external connections, thereby maximizing overall system reliability. The average power dissipation of the overall system is maintained at about 0.4mw per bit including all support circuitry dissipation. This is based on a card configuration of 102 modules with a maximum module dissipation of 600mw.

System status

At present test sites containing individual storage array chip circuits and single bit cross sections have been processed and are being evaluated. Although initial test results are favorable sufficient data has not been accumulated to verify all design criteria. Source-drain storage array chip production masks are in line with other levels nearing completion. Layouts of the bipolar support chips are complete and ready for generation of production masks.

System description

An isometric view of the complete 32,384 word by 36 bit memory system is shown in Figure 1. The total volume occupied by the system is 0.6 cu. ft., resulting in a packing density of approximately 2 million bits/cu. ft. A mechanical housing is provided for the eight multi-layer printed circuit cards that contain the memory storage elements and peripheral circuits. To facilitate insertion and extraction of cards a mechanical assembly is also included. The card connectors are mounted on a printed circuit interconnection board. All necessary system wiring is done on the outside surfaces of this

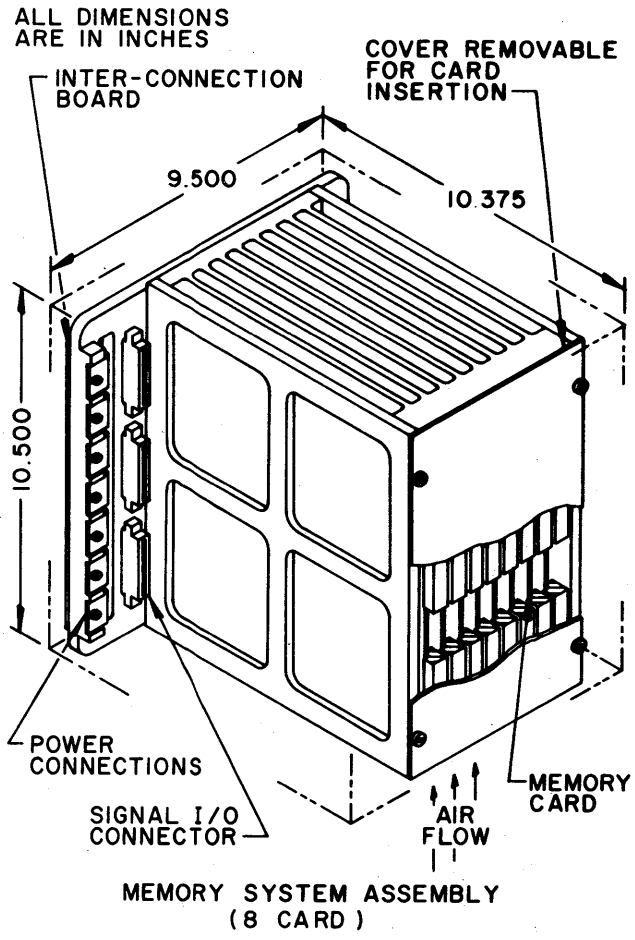


Figure 1—Memory system assembly

board with voltage distribution accomplished by the internal planes. Additional edge connectors are mounted in this board to accommodate I/O signal cabling via plug-in paddle cards. Power connections are provided at the outermost edge of the board.

Since the purpose of this design was to provide a large, fast, low-cost system for use as a computer main frame memory the following design constraints were observed:

Capacity

A one megabit capacity was chosen to be representative of the size of memory that is applicable to a fairly large, high-speed processor. It was decided that the system should be built from modular elements so that memory size and organization could be easily varied. An additional advantage of ease of servicing and stocking accrued from this approach.

Speed

A balance between manufacturability and system requirements was established in setting the performance objectives. This tradeoff resulted in a goal of 250ns. access time and 400ns cycle time.

Density

The density of memory cells should be maximized in order to create minimum cost per cell. An objective of 1024 bits of information was chosen as a reasonable goal using present LSI technology on a .125 in. × .125 in. chip. In order to keep the I/O signal count within reasonable bounds it was decided that address complementing and decoding should be included within the chip. The chip was structured 1024 words by one bit.

Memory card

A drawing of the basic modular unit, the memory card, is shown in Figure 2. The card is a multilayer printed circuit unit with two external planes for signal wiring and two internal planes for distribution of the three required voltages and ground. Ninety-eight double sided connecting tabs are situated along one edge of the card on a .150 in. pitch. These tabs provide for a mating connection with the edge connectors mounted on the interconnection board, and serve to electrically connect all supply voltages and signal wiring

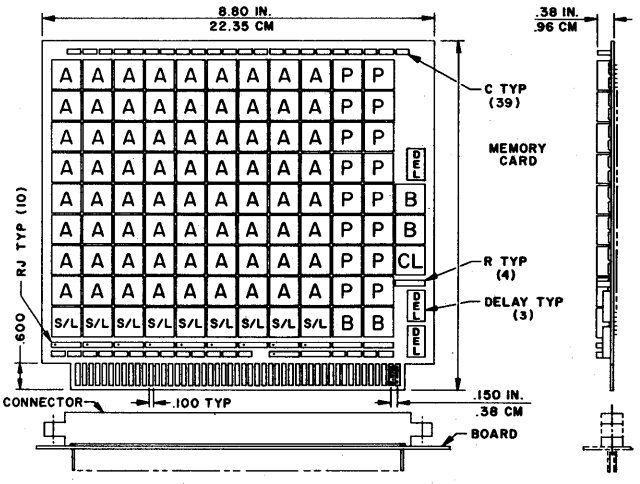


Figure 2—Memory card

to the card. The modules mounted on the card contain one or two chips each, solder reflow bonded to a wiring pattern on a ceramic substrate. Each module occupies a 0.7 in. square area. The 72 modules marked "A" contain the storage array with two chips of 1024 bits each included in each module. The "B" modules provide the primary stages of bipolar buffering while the "P" modules contain the secondary bipolar buffering and decoding. Modules "CL" and "DEL" provide for timing generation while the remaining "S/L" modules perform the sense amplification and latching functions.

Logic design

Memory system logic design was based on the modular card concept to provide easy upward and downward variation of total memory capacity. This card contains all necessary input buffering circuitry, timing circuits, storage elements, sensing circuits, and output registers. The card is structured so that smaller organizations can be obtained by depopulating modules. TTL compatible open collector outputs are provided to allow "wired-or" expansion in multiple card systems such as the 32K word by 36 bit system discussed here. Unit TTL compatible input loads help alleviate the problems of driving a multiple card system.

Card logic flow

A signal flow logic diagram for the 8192 word by 18 bit memory card is shown in Figure 3. Thirteen single rail address lines are required to uniquely determine one

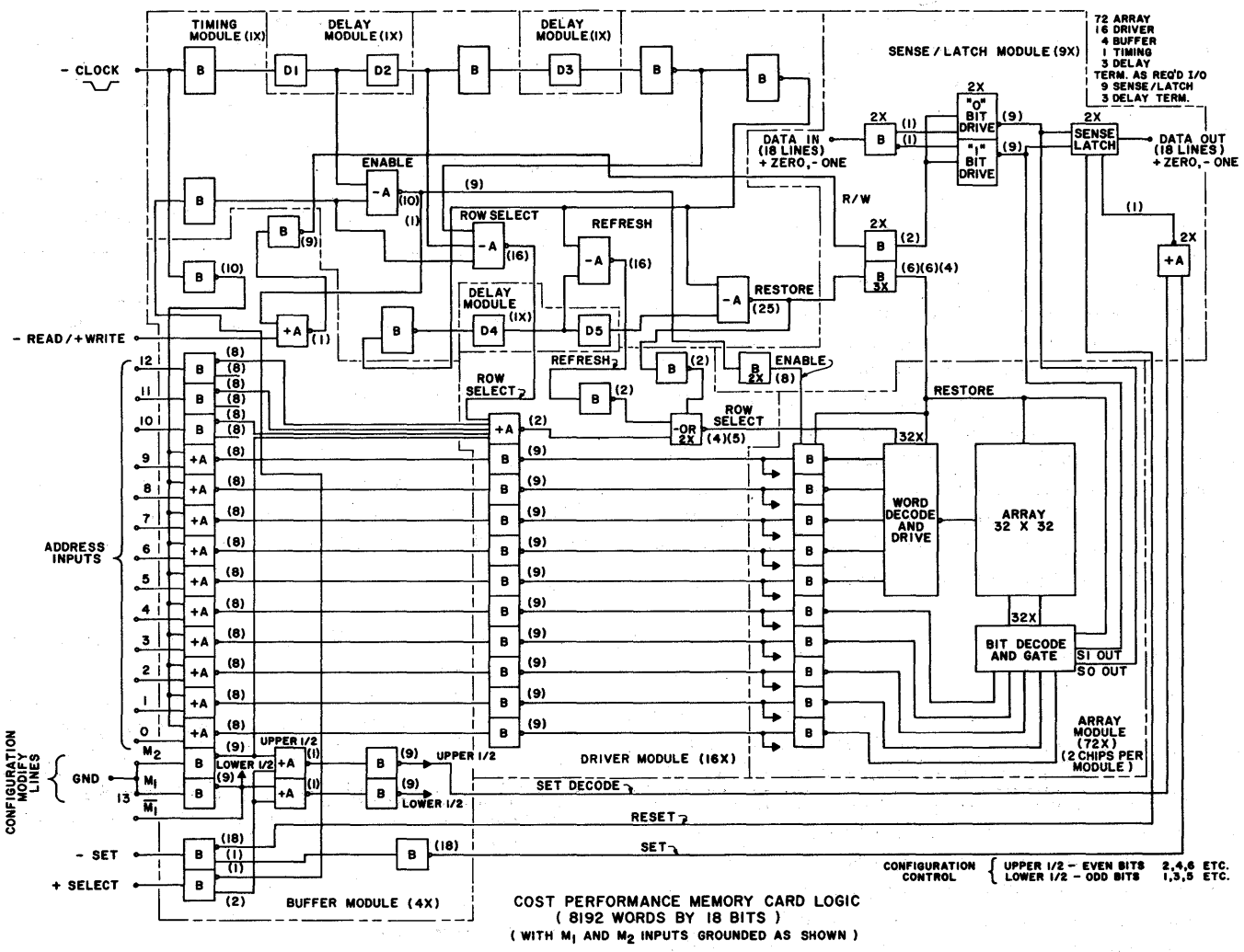


Figure 3—Cost performance memory card logic

of 8192 words. Four control lines are required as follows:

Select—causes selection of entire card.
Read/Write—determine the mode of operation to be performed.
Set—provides timing for the output data register.
Clock—generates timing for read and write operations as well as timing for cyclic data refreshing.
Thirty-six more lines are used for data-in and data-out.

Read operation signal flow

All input lines are buffered immediately upon entering the memory card. A second stage of address buffering is included on the card to allow fan out to all 144 storage array chips. Ten address lines (0–9) drive all storage array chips on the card in parallel, decoding to one of the 1024 bits stored on each chip. The remaining address lines (10–12) are decoded and combined with the timed Select pulse to create two Row Select signals which energize two of the sixteen rows of array chips on the card (two rows of chips per row of modules). Since there are nine array chips in each row, a total of eighteen bits are read out in each operation. The eighteen bits are transmitted to eighteen combination differential sense amplifier and latch circuits which are, in turn, wired to the card connector interface.

Write operation signal flow

Cell selection is performed in the same fashion during a write cycle as in a read cycle. However, instead of

sensing the differential pairs associated with each bit position as in a read operation, the lines are pulsed by one of a pair of bit driver circuits. The magnitude of this excursion is sufficient to force the selected cell to the desired state as indicated by the condition of the data-in line.

Storage array chip logic organization

The storage array chip is organized in a 32 by 32 matrix of storage cells. Five input address lines are complemented upon entering the chip and then selectively wired to the word decoder drivers to provide a one-of-32 selection. These word drivers are also gated by Row Select so that only storage cells on a selected chip are energized. The remaining one-of-32 decoding function is performed on the cell outputs using the remaining five input address lines. The 32 outputs of this final gating stage are wire-ored together to the single differential pair of output bit lines.

Timing structure

Because the array chip is operated in a dynamic fashion, it is necessary to provide several timed lines for periodic refreshing of data and for restoration of the array chip selection circuits after a read or write operation. To minimize the number of lines required at the system interface, the timing generation circuits and delay lines are included on each memory card. These functions are implemented with nonsaturating current switch circuits for minimum skew between timed pulses. Tapped delay lines are used to chop and delay the input clock pulse. A total of four timing pulses are generated as described below:

Row Select: This line is used to turn on the array chip word and bit selection circuits during a read or write operation.

Refresh: This line is timed to follow the Row Select line and energizes all word selection circuits to refresh the array data.

Enable: The address inverters on the array chip are enabled by this line during a normal read or write operation. During the refresh portion of the cycle the absence of this pulse disables the address inverters so that all word selection circuits are simultaneously energized. This permits refreshing of data in all storage cells.

Restore: This line gates on load devices in all array chip selection circuits during the refresh portion of the cycle. These devices provide a recharging path for all the selection circuit node capacitances that were discharged during the immediately preceding operation, and for the node capacitances of the storage cells themselves.

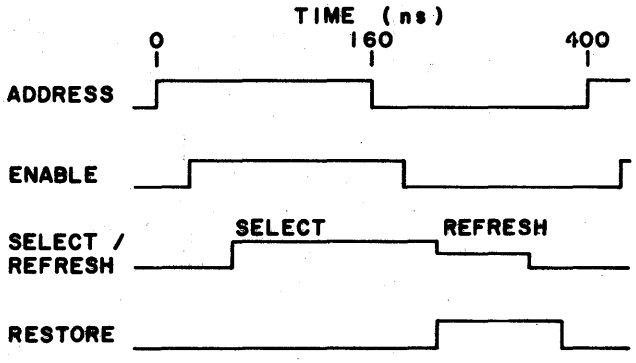


Figure 4—Storage array chip input timing

A diagram showing the relative timings of array chip input lines is shown in Figure 4.

A timing chart for the memory system interface is shown in Figure 5. It can be seen that two timed lines are required at this interface. The first is the Clock line from which all the aforementioned timings are derived. The second is the Set line which latches array data into the output register.

System operation

A block diagram for the complete 32K word by 36 bit memory system is shown in Figure 6. Eight memory

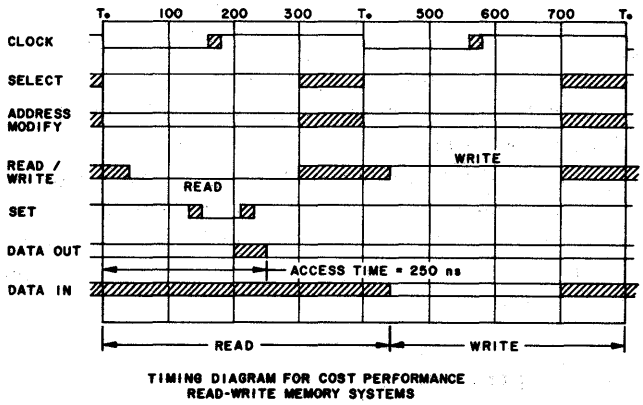


Figure 5—Timing diagram for cost performance read-write memory systems

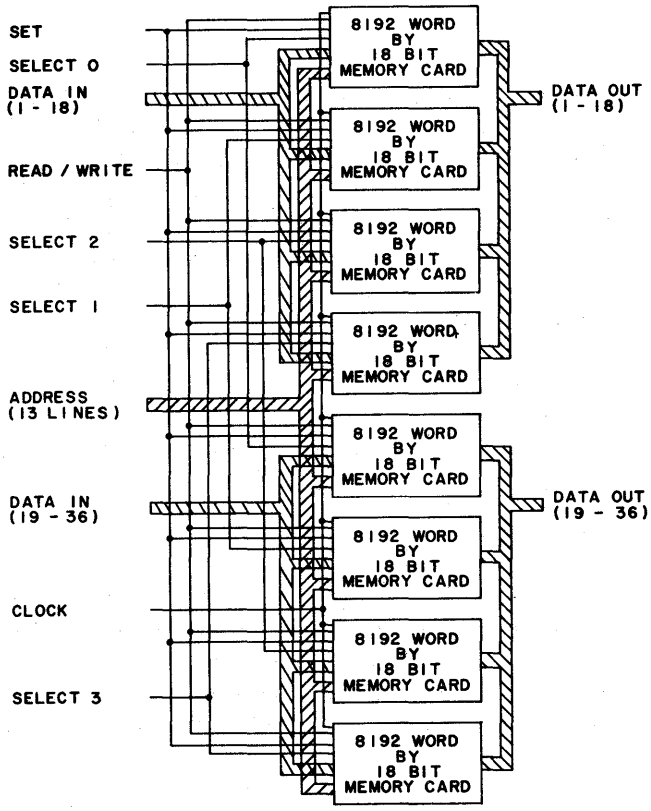


Figure 6—Memory system block diagram

cards, each containing 8192 words by eighteen bits are interconnected as shown to form the total system. All cards are addressed in parallel with four mutually exclusive Select lines energizing one pair of memory cards each cycle. Each card output is "wire-ored" with three other card outputs to expand word depth from 8192 words to 32,768 words.

Maximum access time is 250ns as measured from the +1.6 volt level of the input Clock leading edge transition. Minimum allowable cycle time is 400ns. and is measured in a similar manner from one leading edge Clock transition to the next. Since the Clock line provides refreshing of data, it is also necessary that a *maximum* Clock repetition time of 1.2μs be maintained to avoid loss of information.

Circuit design

In the design of LSI memories the most important costs to be minimized are as follows:

Unmounted chip cost per bit
Chip carrier cost per bit
Printed circuit card cost per bit
Support costs per bit

The chip cost per bit is largely a function of the area of processed silicon required per bit of storage, the process complexity as measured by the number of masking or diffusion steps, and the chip yield. All of these factors strongly favor a MOS-FET chip process over bipolar process. For a given chip size the chip carrier costs, the printed circuit cost and the support costs are all inversely proportional to the number of bits per chip, thus the advantage of high-density MOS-FET array circuitry is overwhelming.

The chief drawback to MOS-FET circuits for semiconductor memories is their low gain-bandwidth compared with bipolar circuits using equivalent geometric tolerances. This shortcoming can be minimized by using bipolar circuits to provide the high-current drives to the MOS-FET array circuits, and by using bipolar amplifier circuits to detect the low MOS-FET sense currents. If the circuits are partitioned so that all the devices on a given chip are either bipolar or MOS-FET, no additional processing complexity is added by mixing the two device types within the same system. The use of bipolar support circuits also allows easy interfacing with standard bipolar logic signals, thus the interface circuits can match exactly standard interface driving and loading conditions.

Given an MOS-FET array chip, the two most important remaining choices involve the polarity of the MOS-FET device (n-channel or p-channel) and the gate oxide thickness. It is well known that the transconductance of n-channel devices is approximately three times that of equivalent p-channel device and thus the current available to charge and discharge capacitance is substantially greater. Since the substrate is backbiased by several volts in an n-channel device, the source-to-substrate and drain-to-substrate capacitances are also slightly lower, with the net result that n-channel circuits are a factor of two to three faster than equivalent p-channel circuits. This speed difference is critically important if address decoding and bit/sense line gating are to be included on the MOS-FET chip. Because the transconductance of a MOS-FET device, and consequently its ability to rapidly charge and discharge a capacitance, is inversely proportional to the gate oxide thickness, it is advisable to use the minimum thickness that the state of the art will allow; in this case 500 Angstroms was chosen as the minimum that would give a good yield of pinhole free oxide with adequate breakdown voltage. Other key device parameters are tabulated below:

V_t = 1.25V nominal with substrate bias
ρsub = 2Ωcm P type
γm = 33.5μa/v nominal
ρd = 7Ω/square N type

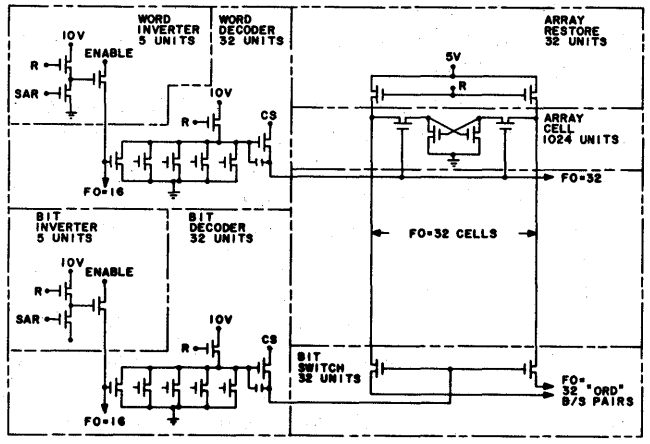


Figure 7—Array chip cross-section

Chip partitioning

Since it was desired that the same chip set be used to configure memory systems of different sizes, different word lengths, and different system speeds, many of the chip partitioning choices are obvious. The timing circuits, which are used only once per system, are contained on a separate chip. The sensing and bit/drive circuits are combined on one chip to allow easy expandability in the bit dimension. The array drivers are contained on a third chip type to allow easy expansion in the memory size, while general buffering and gating logic make up the fourth chip type. The most important chip-partitioning choice involves the dividing line between bipolar and MOS-FET circuits at the array chip interface. By including the array word-line decoding and the array bit/sense line gating on the array chip, the number of connections to the array chip can be greatly reduced, allowing the chip carrier wiring to be less dense and the chip pad size and spacing to be relaxed. The complexity of the bipolar support circuitry was reduced still further by including the address inverters on the array chip, with a small penalty in delay. If a MOS-FET sense amplifier/bit driver were included on the array chip, however, the increase in delay would be excessive, owing to the poor response time of MOS-FET high-gain amplifiers. In the design shown here, the cell sense current is gated to a bipolar sense amplifier for amplification and discrimination, and the cell nodes are driven through the same MOS-FET gating circuits to the desired state during the write operation. This arrangement requires that approximately 35 percent of the available array chip area be used for decoding and gating circuits, with the remaining 65 percent used for storage cells. Figure 7 shows a cross-section circuit schematic of the array chip. Included below are nominal chip parameters:

Address input capacitance . . . (including gate protective device) 4pf
Enable input capacitance . . . (depending on address) 2.75 pf or 20pf
Restore input capacitance . . . (including gate protective device) 57pf
Sense line input capacitance . . . 5.5pf
Select input capacitance . . . 8pf
Word line capacitance . . . 7.5 pf
Bit line capacitance . . . 2pf
Sense current . . . 150μa
Maximum gate protective device input 3400V

Storage cell

Typical MOS-FET storage cells are shown in Figure 8. In cell 8(a), T_1 and T_2 form the cross-coupled pair, while T_3 and T_4 gate the external circuitry to the cell nodes, either to sense the state of the cell by detecting the imbalance in the current through T_3 and

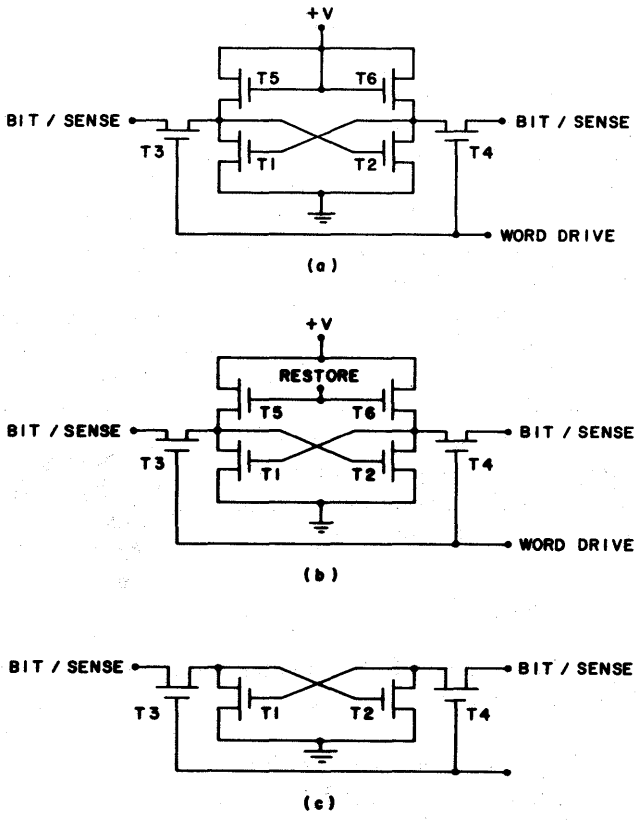


Figure 8—Storage cell configurations

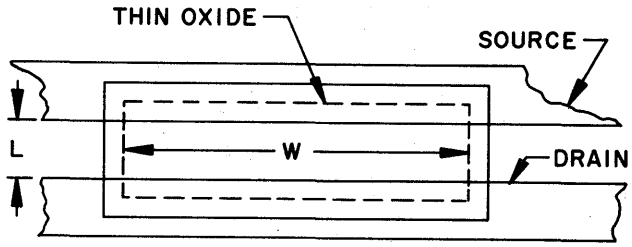


Figure 9—W/L ratio

T_4 or to write into the cell by pulling one node to ground while simultaneously driving the other cell node positive. The load devices, T_5 and T_6, replace the leakage current from the more positive node during stand-by. Since one of the load devices has full voltage across it at all times, the standby power dissipation of the cell will be quite high in comparison to the cell sense current unless the W/L ratio, Figure 9, of the load device (T_5, T_6) is made very small compared to the W/L ratio of the cross-coupled device (T_1, T_2). This, in turn, requires that either the load devices or the active devices or both occupy a large chip area. In addition, the standby load current flowing through the on-biased active device provides a voltage drop across that device, tending to unlatch the cell. This effect can be compensated for by increasing the value of all device thresholds, however, this will require a higher supply voltage to maintain the same standby current thereby increasing the power dissipation.

In cell 8(b), the standby power is reduced by pulsing the "restore" input at a clock rate sufficiently fast to replace leakage current from the cell node capacitance, while maintaining a low average power drain. The chief drawback to this cell is the five connections must be made to the cell, with a resulting increase in cell complexity over (a) above.

Cell 8(c) shows the configuration chosen for this memory. In this cell, both the word selection and the restore functions are performed through the same devices and array lines, by time sharing the word-select and restore line. During read-out, the cell operation is similar to 8(b) above. At the end of each memory cycle, however, all word lines are raised to the "restore" level for a period sufficient to recharge the cell node capacitances, then all word lines are dropped and the next memory cycle can begin. Selection of the "restore" level is dependent on the speed at which the cell node capacitance is to be charged and the sense line voltage support level required during restore. Too high a "restore" level creates a large current flow thru the restore devices lowering the sense line voltage used to charge the cell; too low a voltage prevents the cell node capacitance from reaching the required voltage for data retention. This cell employs fewer devices and less complex array wiring than either of the cells above, and thus requires substantially less silicon area. The disadvantage of this approach is that the restore function must be synchronized with the normal read/write function since they share the same circuitry. The average power cannot be made as low as in (b) above, since the restore current and the sense current are both determined by a common device, and the restore frequency is determined by the memory cycle time; however, the average power can be made significantly lower than with the static cell 8(a) above.

MOS-FET support circuits

The MOS-FET support circuits employed on the array chip are shown in Figure 10. A better understanding of the circuit operation will be gained by first considering the MOS-FET inverter circuit (Figure 10). At the end of a read/write cycle, the input address level is held down, the E level is down, and the R line is pulsed positive, charging node A to approximately +7 volts. When the R pulse has terminated, node A remains positive awaiting the next input. At the start of the read/write cycle, the address input is applied to T_1; if the address line is positive, node A quickly discharges through T_1, and when E is applied to T_3,

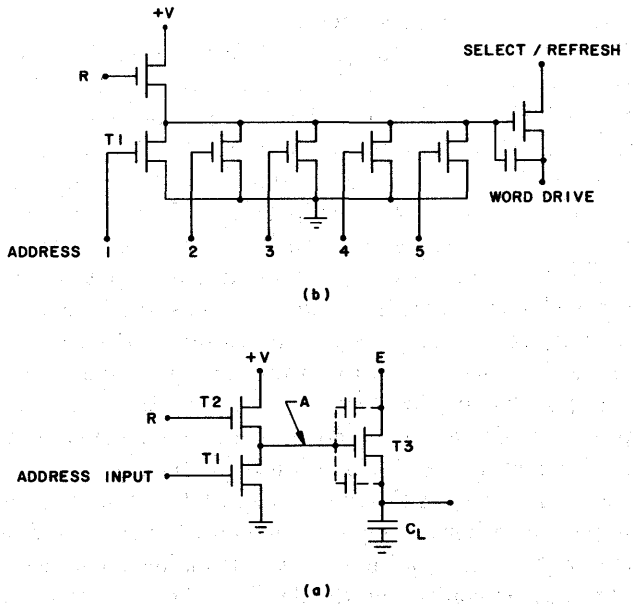


Figure 10—Array chip inverter-decoder circuits

T_3 remains non-conducting and the address inverter output remains at ground potential. If, however, the address input line is a down level, then node A remains charged to +7 volts, and both T_1 and T_2 are cut-off, while T_3 is biased on. When a positive E pulse is applied to T_3, current is capacitively coupled into node A from both the E node and from the output node, with the result that node A is driven more positive than either; thus T_3 remains strongly biased on, charging the output node capacitance to the level of E. When the positive E pulse is terminated, the same action quickly discharges the output to ground through the E line. At the end of the address pulse, a positive R pulse is again applied to T_2, restoring node A to +7 volts. This regenerative inverter has several advantages over a conventional source follower circuit; (a) the output up level is set by the level of the E input, and does not vary with the device threshold voltage; (b) the output rise time is nearly linear, since the gate-to-source bias on T_3 remains well above the threshold voltage throughout the transition, and (c) this same high conductance output device can be used to both charge and discharge the load capacitance. Since the leakage current from node A during a cycle is negligible, the final potential of node A, and thus the output drive current, is determined by the capacitor-divider action of the gate-to-source, gate-to-drain, and gate-to-substrate capacitances associated with device T_3. Any of these capacitances can be artificially increased to optimize the circuit operation. The operation of the decoder circuit (Figure 10b) is similar to the inverter just described, with the bi-level chip select/refresh line replacing the E input discussed previously. Thus, a single word line is selected to the higher (Select) level during the Read/Write portion of the cycle, while *all* word lines are selected to the lower (Refresh) level during the Restore portion of the cycle. Thus the cell input/output devices are biased to a low impedance to provide maximum sense current during readout, and to a higher impedance to reduce the power dissipation and maintain the necessary sense line voltage during the restore operation.

Protective devices are used on all gate inputs to the array chip to reduce the yield loss from static electricity during processing, testing, and assembly. Because the array chip uses a P-epitaxy grown on a $P+$ substrate it was possible in this system to replace the usual RC protective device with a more favorable zener type. This device is an $N+$ diffusion diffused at the same time as the source-drain diffusions and exhibits a low internal impedance when its depletion region intersects the $P+$ substrate. The required reverse breakdown voltage is obtained by controlling the depth of the $N+$ diffusion. When driven with an impedance equivalent to a human body, approximately 1000 ohms, gate protection is

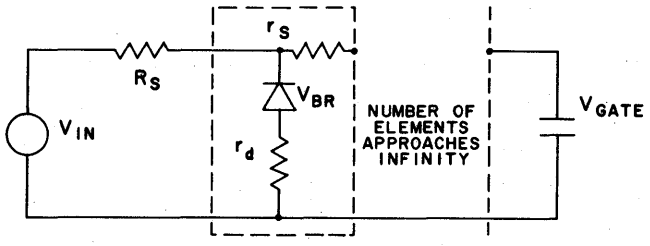


Figure 11—Gate protective device

provided for input levels up to 3400 volts. Figure 11 and equation 1 represent the characteristics and operation of this type protective device as presented during the IEEE, International Electron Devices Meeting, October 29 thru 31, 1969.[1] For analysis the device is arranged as a series of distributed elements; each element containing sub-elements r_s, r_d, and V_{BR}.

$$V_{\text{gate}} = V_{BR} + \{(V_{in} - V_{BR})(r_s r_d)^{1/2}[\cosh(r_s\gamma/r_d)^{1/2}]^{-1}\}/ R_s + (r_s r_d)^{1/2} \quad (1)$$

In this design γ, the number of elements, was set at nine with the following sub-element values:

$$r_s = 4.27 \text{ ohms}$$

$$r = 61.2 \text{ ohms}$$

$$V_{BR} = 30 \text{ volts}$$

The maximum capacitance before breakdown is 1.25pf.

Bipolar support circuits

Because of the critical timing relationships required among the Select/Refresh, Enable, Address, and Restore pulses to the array chip, all timing pulses are generated *on each card* by a special timing chip and three tapped delay lines. This arrangement allows each card to be fully tested with the timing circuits that will drive it, and minimizes any interaction between cards in a multi-card system.

The TTL compatible buffer chip allows interfacing conveniently with the TTL compatible logic of the using system, and minimizes the loading which the memory presents to the system.

A schematic cross-section of the Drive and Sense circuits is shown in Figure 12. The Driver module, when addressed, selects a row of nine Array Chips from a low impedance TTL output. The ten address inputs to the

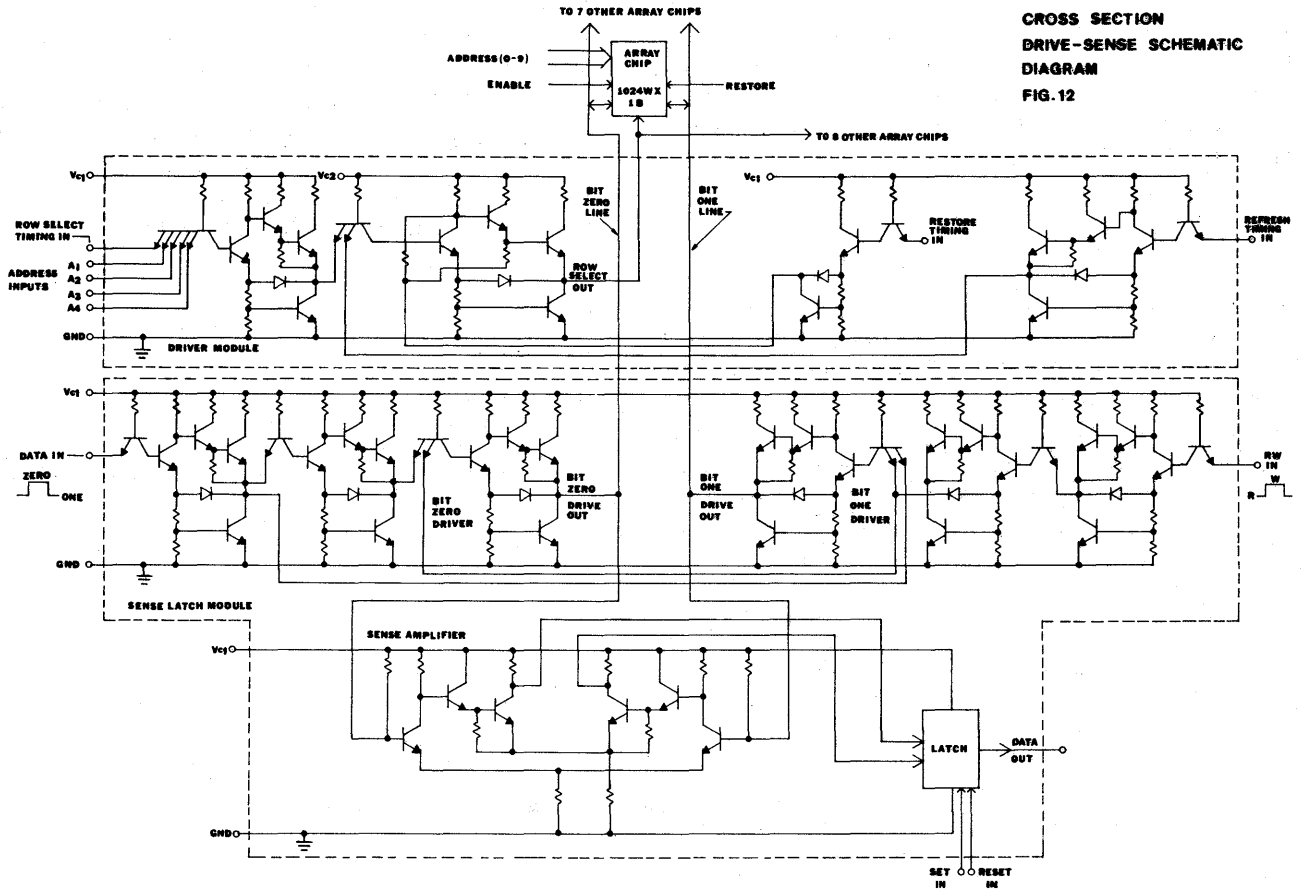


Figure 12—Cross-section drive-sense schematic diagram

Array Chip serve to select one bit of the 1024 bits/chip. The write pulse permits the data-in to be loaded differentially into the single bit which has been addressed. The removal of the write pulse turns off both the "one" and "zero" bit drives with the low impedance active pull ups rapidly charging the capacitance of the bit lines to a voltage level required for the read mode of operation.

The sense amplifier requires a minimum differential signal of 50 microamps to detect a "one" or "zero" stored in the addressed bit. This information is transferred to a set-reset latch which is included to increase the "data-good" time to the using system.

During a portion of every cycle not used for read/write operation the timing chip provides refresh and restore timing pulses which turn on all the driver modules on the memory card to a lower voltage level, and perform the refresh operation previously discussed.

All four of the bipolar support chips are packaged one-chip per chip carrier, to allow flexibility in configuring various size memory systems. In all cases, the power density is limited to 600 mw per chip carrier,

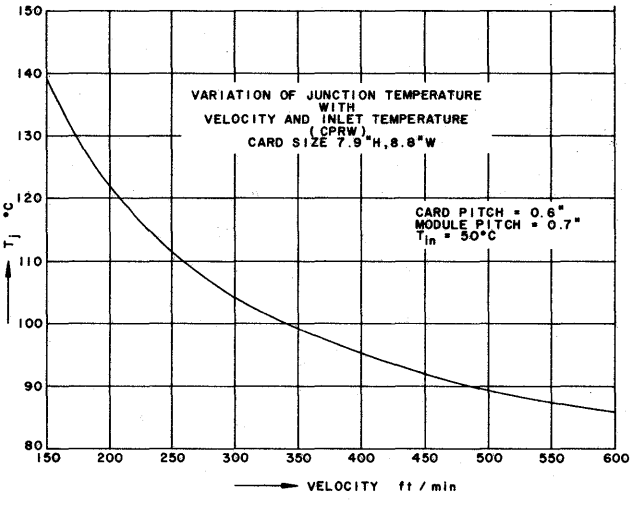


Figure 13—Variation of junction temperature with velocity and inlet temperature

a level which allows for convenient forced-air cooling. Because the limiting heat factor is the junction temperature of the bipolar support circuits all cooling considerations are in respect to this parameter. Figure 13 illustrates the junction temperature as a function of air flow thru the system.

CONCLUSION

The memory system described here is but one of many possible sizes and organizations that can be created using the same modular approach. If desired, several smaller organizations can be used within the same system without significant cost penalties. The system approach to memory design has created an optimum condition wherein each individual component is matched to the other components with which it must interact. This approach also yields a memory with a simple, effective, easily usable set of interface requirements. It is anticipated that increasing yields will allow prices competitive with magnetic storage for high-performance main memories. This low cost, coupled with high performance and density, makes a powerful combination for use in future system designs.

REFERENCE

1 M LENZLINGER
Gate protection of MIS devices
Presented at International Electron Devices Meeting
Washington D C 1969

Optimum test patterns for parity networks

by D. C. BOSSEN, D. L. OSTAPKO and A. M. PATEL

IBM Laboratories
Poughkeepsie, New York

INTRODUCTION

The logic related to the error detecting and/or correcting circuitry of digital computers often contains portions which calculate the parity of a collection of bits. A tree structure composed of Exclusive-OR gates is used to perform this calculation. Similar to any other circuitry, the operation of this parity tree is subject to malfunctions. A procedure for testing malfunctions in a parity tree is presented in this report.

Two important assumptions are maintained throughout the paper. First, it is assumed that the parity tree is realized as an interconnection of Exclusive-OR gates whose internal structure is unknown or may differ. This requires that each gate in the network receive a complete functional test. Second, it is assumed that detection of single gate failures is desired.

Since each gate must be functionally tested, an m-input Exclusive-OR gate must receive 2^m input patterns. It will be shown that 2^m test patterns are also sufficient to test the network of any size, if m is the maximum number of input lines to any Exclusive-OR gate. Hence, the procedure yields the minimum number of test patterns necessary to completely test the network for any single Exclusive-OR gate failure. It will also be shown, by example, that the procedure is fast and easy to apply, even for parity trees having a large number of inputs.

GATE AND NETWORK TESTABILITY

Since the approach is to test the network by testing every gate in the network, it is primarily necessary to discuss what constitutes a test for an individual Exclusive–OR gate. Although it is assumed that the parity trees are realized as a network of Exclusive-OR gates, no internal realization is assumed for the Exclusive-OR gates. Hence, it will be presumed that all 2^k input patterns are necessary to diagnose a single k-input Exclusive-OR gate. Each gate, therefore, is given a complete functional test so that single error detection means that any error in one Exclusive-OR gate can be detected. The following is the definition of a gate test.

Definition 1:

A test for a k-input Exclusive-OR gate is the set of 2^k distinct input patterns of length k. Figure 1 shows a three input Exclusive-OR gate, the $2^3=8$ input test patterns, and the output sequence which must result if a complete functional test is to be performed.

If the output sequence and the sequences applied to each input are considered separately, each will be a vector of length 2^k. Thus, the Exclusive-OR gate can be considered to operate on input vectors while producing an output vector. Figure 2 shows a three input Exclusive-OR gate when it is considered as a vector processor. In terms of vectors, a test is defined as follows.

Definition 2:

A test for a k-input Exclusive-OR gate is a set of k vectors of length 2^k which, when considered as k sequences of length 2^k, presents all 2^k distinct test patterns to the gate inputs.

Theorem 1:

If K is a test for a k-input Exclusive-OR gate, then any set M, $M \subset K$, having m, $2 \leq m \leq k-1$, elements forms 2^{k-m} tests for an m-input Exclusive-OR gate.

Proof:

Consider the k vectors in K as sequences. Arrange the sequences as a k by 2^k matrix in which the last m

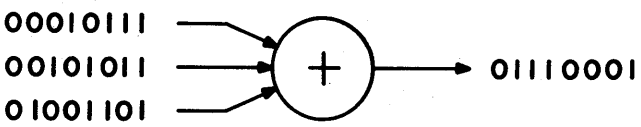


Figure 1—Three input Exclusive-OR gate with test patterns

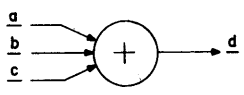
rows are the sequences in M. Code each column as a binary number with the highest order bit at the top. Since the columns are all distinct according to definition 1, each of the numbers 0 through 2^k-1 must appear exactly once. Considering just the bottom m rows, it follows that each of the binary numbers 0 through 2^m-1 must appear exactly 2^{k-m} times. Since each of the possible sequences of m bits appears 2^{k-m} times, definition 1 implies that the set M forms 2^{k-m} tests for an m-input Exclusive-OR gate.

Network testability:

Two conditions are necessary for a network of Exclusive-OR gates to be completely tested. First, each gate must receive a set of input vectors that forms a test. Second, any one gate error must be detectable at the network output. For the first condition it is necessary that the set of vectors from which the tests are taken be closed under the operation performed by the k-input Exclusive-OR gates. The second condition requires that any erroneous output vector produce an erroneous network output vector. The structure of this set of vectors and their generation will be discussed in the following sections.

AN EXAMPLE

The test pattern generation procedure is so simple and easy to apply that it will be presented by way of an example before the theoretical properties of the desired sequences are discussed. The algorithm proceeds by selecting an arbitrary output sequence and



WHERE a = 00010111, b = 00101011, c = 01001101, d = 01110001

Figure 2—Three input Exclusive-OR gate as a vector processor

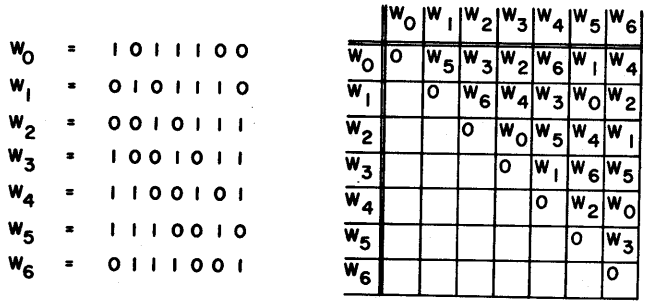
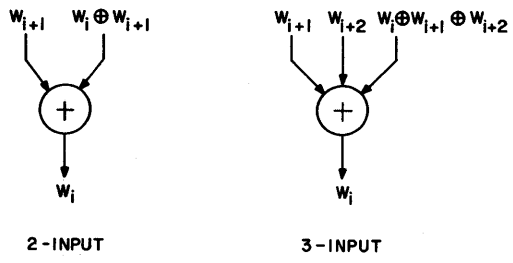


Figure 3—Test sequences and their addition table

then successively determining input sequences which test each gate to produce the desired output.

Figure 3 presents the seven sequences and the associated addition table that will be used in the example. Figure 4 illustrates the gate labeling procedure which will be used to determine the inputs when the output is specified. Figure 5 shows the parity tree with 57 inputs and 30 Exclusive-OR gates of two and three inputs arranged in a four level tree. The procedure generates eight test patterns which will completely test all 30 gates of the tree.

The procedure is initiated by assigning an arbitrary sequence to the output of the tree. In the example, W_0 is selected as the final output sequence. Employing the 3-input gate labeling procedures shown in Figure 4, the inputs are determined to be W_1, W_2, and W_4. With these three sequences, the gate is completely tested. These inputs are then traced back to the three gates in the third level. Using the gate labeling procedure again, the inputs for the gates from left to right are W_2, W_3, W_5; W_3, W_0; and W_5, W_2. The sequences assigned to the inputs can be determined quickly and easily by making use of tracing and labeling. Under proper operation, each gate is completely tested and a single gate failure will produce an incorrect sequence



NOTE: $W_i \equiv W_i \pmod 7$

Figure 4—Gate labeling procedures

at the output. Above each input the required sequence is listed, and the correct output is the sequence W_0. The test patterns are obtained by reading across the sequences and noting the correct output. The test is completed by adding the all zero test pattern. This should produce a zero output.

THEORETICAL PRELIMINARIES

Consider the set of vectors generated by taking all mod-2 linear combinations of the k vectors of a given test set K. This set is obviously closed under mod-2 vector addition. In a parity check tree network an input of any subset of vectors from this set will produce vectors in the set at all input-output nodes of the Exclusive-OR gates. Some further insight can be gained by viewing the above set as a binary group code. The generator matrix G of this code, whose rows are k vectors from K, contains all possible k-tuples as columns. If we delete the column of all 0's in G, the resulting code is known as a MacDonald[1] code in which the vector length n is 2^k-1 and the minimum distance d is 2^{k-1}. The cyclic form of the MacDonald code is the code generated by a maximum length shift register.[2]

Theorem 2:

Any independent set of k vectors from the Maximum Length Shift Register Code of length 2^k-1 forms a test set for a k-input Exclusive-OR gate, excepting the pattern of all 0's.

Proof:

Any independent set of k-vectors from the code forms a generator of the code. In the Maximum Length Shift Register Code as well as in the MacDonald Code, $2d-n=1$. This implies*[3] that any generator matrix of the code contains one column of each non-zero type. By definition 2, this forms the test for a k-input Ex-OR gate excepting the test pattern of all 0's.

Corollary:

For an m-input gate, $m \leq k$, any set of m-vectors from a MLSRC of length 2^k-1 forms a sufficient test.

The proof follows from Theorems 1 and 2.

The maximum length shift register sequences can be generated[2] by using a primitive polynomial $p(X)$ of degree k in GF (2). Let $g(X)=(X^n-1)/p(X)$ where $n=2^k-1$. Then the first vector W_0 of the MLSRC is the binary vector obtained by concatenating $k-1$ zeros to the sequence of the coefficients of $g(X)$. The vectors $W_1, W_3 \ldots W_{2^k-2}$ are then obtained by shifting W_1 cyclically to the right by one digit for 2^k-2 times. The method is illustrated for $k=3$. A primitive polynomial of degree 3 in GF (2) can be obtained from tables,[2] e.g., X^3+X+1 is primitive.

$$g(X)=(X^7-1)/(X_3+X+1)=X^4+X^2+X+1.$$

Then W_0 is obtained from $g(X)$ as

$$W_0=1\ 0\ 1\ 1\ 1\ 0\ 0$$

The sequences $W_1, W_2 \ldots W_6$ are obtained by shifting W_0 cyclically as,

$$W_1=0\ 1\ 0\ 1\ 1\ 1\ 0$$

$$W_2=0\ 0\ 1\ 0\ 1\ 1\ 1$$

$$W_3=1\ 0\ 0\ 1\ 0\ 1\ 1$$

$$W_4=1\ 1\ 0\ 0\ 1\ 0\ 1$$

$$W_5=1\ 1\ 1\ 0\ 0\ 1\ 0$$

$$W_6=0\ 1\ 1\ 1\ 0\ 0\ 1$$

Note that when W_{2^k-2} is shifted cyclically to the right by 1 digit, the resulting vector is W_0. For the purpose of uniformity of relationship among the vectors we

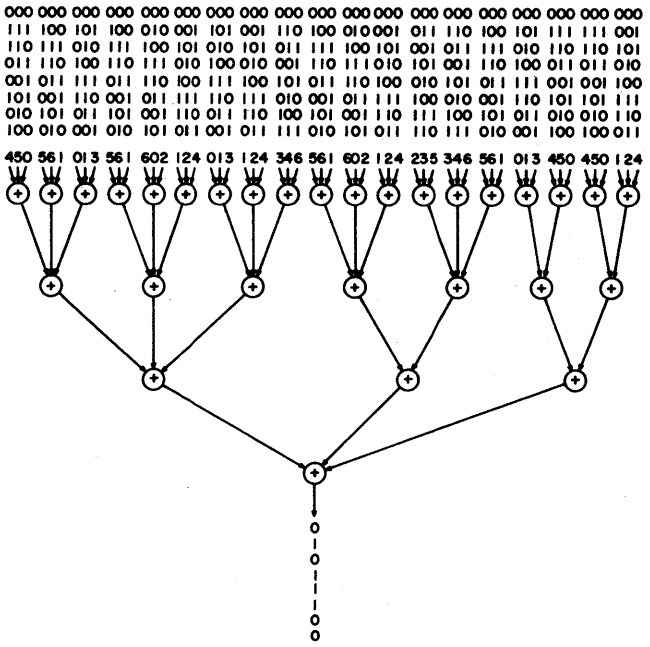


Figure 5—Four level parity tree with test patterns

* In Reference 3 it is shown that in a group code with $2d-n=t>0$, there are t columns of each type.

introduce the notation: $W_i \equiv W_{i(\text{mod } 2^k-1)}$. Now the following theorem gives a method of selecting independent vectors from a MLSRC.

Theorem 3:

The vectors W_i, W_{i+1}, ..., W_{i+k-1} in a MLSRC of length 2^k-1 form an independent set.

Proof:

Suppose $g(X)$ is given by $g(X)=g_rX^r+g_{r-1}X^{r-1}+\ldots+g_1X+g_0$, where $r=(2^k-1)-k$. Then the set of vector W_0, W_1, ..., W_{k-1} are given by

$$W_0 = g_r \; g_{r-1} \; \cdot \; \cdot \; \cdot \; g_0 \; 0 \; 0 \; 0 \; \cdot \; \cdot \; 0$$

$$W_1 = 0 \; g_r \; g_{r-1} \; \cdot \; \cdot \; \cdot \; g_0 \; 0 \; 0 \; \cdot \; \cdot \; 0$$

$$W_2 = 0 \; 0 \; g_r \; \cdot \; \cdot \; \cdot \; \cdot \; \cdot \; g_0 \; 0 \; \cdot \; \cdot \; 0$$

$$\cdot$$
$$\cdot$$
$$\cdot$$

$$W_{k-1} = 0 \; 0 \; 0 \; 0 \; \cdot \; \cdot \; \cdot \; g_r \; \cdot \; \cdot \; \cdot \; g_0$$

Clearly they are linearly independent. Because of the cyclic relationship, this implies that W_i, W_{i+1}, ..., W_{i+k-1} are independent.

Corollary:

The vectors W_{i+1}, W_{i+2}, ..., W_{i+m-1}, and $W_i \oplus W_{i+1} \oplus \ldots \oplus W_{i+m-1}$, $(m \leq k)$, form an independent set. With this as a test to an m-input Ex-OR gate, the correct output vector is W_i.

As a direct consequence of the above theorems we have the following algorithm for the test pattern generation for a given Exclusive-OR network.

Algorithm for test pattern generation:

It is assumed that the Exclusive-OR network is constructed in the form of a tree by connecting m-input Ex-OR gates where m may be any number such that $m \leq k$.

1. Select any vector W_i from a MLSRC of length 2^k-1 as the output of the network.
2. Label the inputs to the last Ex-OR as W_{i+1}, W_{i+2}, ..., W_{i+m-1}, and $W_i \oplus W_{i+1} \oplus \ldots \oplus W_{i+m-1}$.
3. Trace each of the above inputs back to the driving gate with the same vector. Repeat steps (2) and (3) to determine the proper inputs to the corresponding gates.
4. The vectors at the input lines to the Ex-OR tree are then the test input vectors with the correct output as W_i.
5. An additional all 0 pattern as input to the network with 0 as correct output completes the test.

It is easy to see that the test patterns generated by the above algorithm provide a complete test for each Ex-OR gate in the parity check tree. Furthermore, any single gate failure will generate an erroneous word which *will* propagate to the output. This is due to the linearity of an Ex-OR gate. Suppose one of its inputs is the sequence W_i with a corresponding correct output sequence W_j. If the input W_i is changed by an error vector to W_i+e, then the corresponding output is W_j+e. Clearly, the error will appear superimposed on the observed network output.

TEST MECHANIZATION

We have shown that the necessary test patterns for a parity tree can be determined by a simple procedure using a set of k independent vectors or code words W_0, W_1, ..., W_{k-1} from a MLSRC as the input to each gate of k inputs. The result of applying this procedure to a network is an input sequence W_i for each network input and each network output. Testing is accomplished by applying the determined sequences simultaneously to each input and then comparing the expected network outputs with the observed network outputs.

Let the gate having the greatest number of inputs in the network show k inputs. The entire test can be mechanized using a single (2^k-1)-stage feedback shift register. To do this a unique property of the MLSR codes is used. From this property it follows that the entire set of non-zero code words is given by the

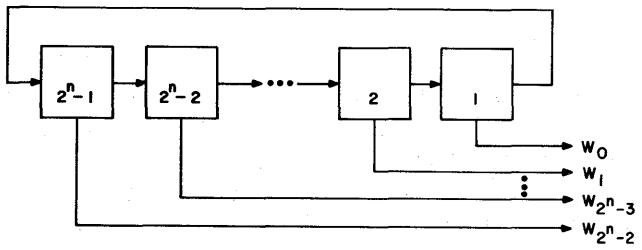


Figure 6—Shift register for generating test patterns

2^k-2 cyclic shifts of any non-zero code word together with the code word itself.

If a (2^k-1)-stage shift register is loaded with a particular code word W_0 as in Figure 6, then the sequence of bits observed at position 1 during 2^k-1 shifts of the register is the code word W_0. Similarly for every other position i, a different code word W_{i-1} is observed, so that the entire set of 2^k-1 sequences is available. Since the correct output of the network is one of the code words, it is also available at one of the stage outputs for comparison. The general test configuration is given by Figure 7.

SELF-CHECKING PARITY TREE

Let us suppose that the test sequences and the shift register connections for a parity network have been determined as in Figure 7. A modification of this mechanization can be used to produce a self-testing parity network under its normal operation. The key idea is to monitor the normal randomly (assumed) occurring inputs to the network and to compare them with the present outputs of the shift register. When and only when a match occurs, the comparison of the outputs of the parity networks with the appropriate code words is used to indicate either correct or incorrect operation, and the shift register is shifted once. This brings a new test pattern for comparison with the normal inputs. Every 2^k-1 shifts of the register means that a complete test for all single failures has been performed on the network.

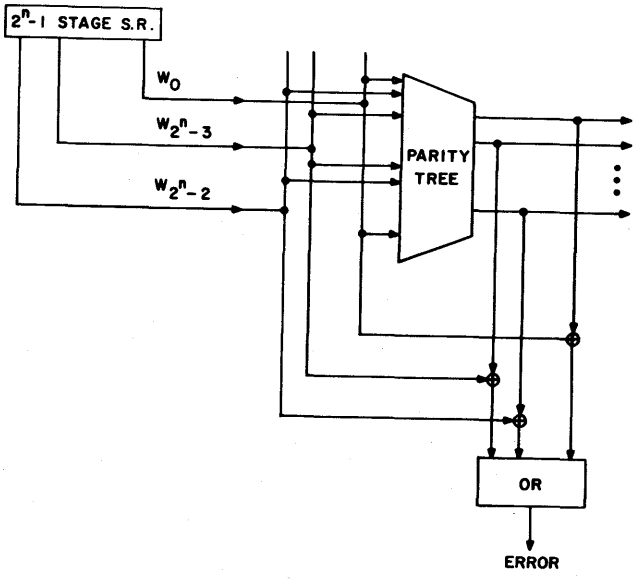


Figure 7—General testing scheme

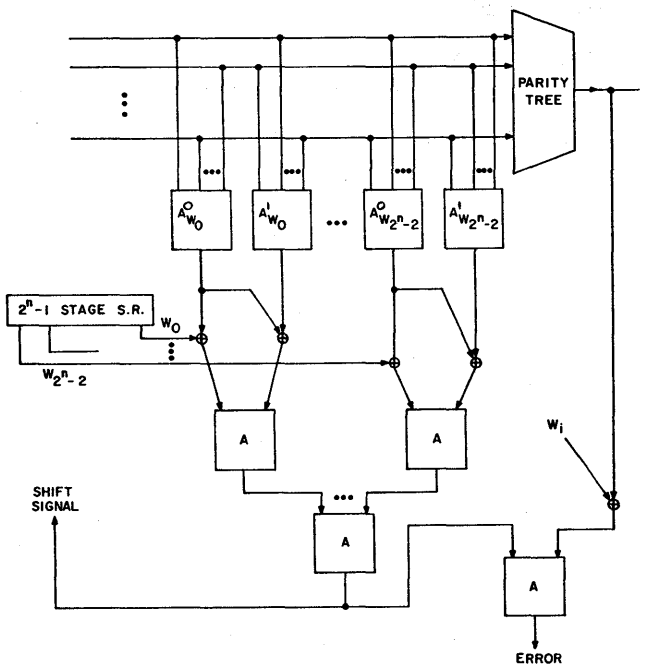


Figure 8—Self checking parity tree

The mechanization of the self-checking parity tree is shown in Figure 8. The inputs to the AND gate $A_{w_i}{}^1$ are the set of input lines of the parity tree which receive the test sequence W_i. The inputs to the AND gates $A_{w_i}{}^0$ are the inverse of the input lines of the parity tree which receive the test sequence W_i.

An alternate approach to self-checking is to use the testing circuit of Figure 7 as a permanent part of the parity tree. The testing is performed on a time-sharing or periodic basis while the circuit is not used in its normal mode. This is easily accomplished by having the clock, which controls the shift register, gated by a signal which indicates the parity tree is not being used. This could be a major portion of the memory cycle when the parity tree under consideration is used for memory ECC.

CONCLUSION

We have shown that a very low and predictable number of test patterns are necessary and sufficient for the complete testing of a parity tree under the single failure assumption. The required tests are easily and rapidly determined by an algorithm which is presented. (An application of this technique is also given for a self-checking parity tree.) Since the effect of the input test patterns is a complete functional test of each gate, the tests are independent of any particular failure mode.

REFERENCES

1 J E MacDONALD
Design methods for maximum minimum-distance error correcting codes
IBM J of R & D Vol 4 pp 43-47 1960

2 W W PETERSON
Error correcting codes
MIT Press Cambridge Massachusetts 1961

3 A M PATEL
Maximal group codes with specified minimum distance
IBM J of R&D Vol 14 pp 434-443 1970

A method of test generation for fault location in combinational logic*

by Y. KOGA and C. CHEN

University of Illinois
Urbana, Illinois

and

K. NAEMURA

Nippon Telegraph and Telephone Public Corporation
Musashino, Tokyo, Japan

INTRODUCTION

The Path Generating Method[1] is a simple procedure to obtain, from a directed graph, an irredundant set of paths that is sufficient to detect and isolate all distinguishable failures. It was developed as a tool for diagnostic generation at the system level, e.g., to test data paths and register loading and to test a sequence of transfer instructions. But it has been found to be a powerful tool for test generation for combinational logic networks as well.

The combinational network to be diagnosed is represented as a set of complementary Boolean forms, where complementation operators have been driven inward to the independent variables using DeMorgan's Law. A graph is then obtained from the equations by translating logical sum and logical products into parallel and serial connections, respectively. A set of paths is generated from the graph, which is irredundant and sufficient for detection and isolation of single stuck-type failures.

The advantage of this approach to test generation lies in the irredundancy and isolation capability of the generated tests as well as the simplicity of the algorithm. Several test generation methods have been develveloped,[2,3,4,5,6] but none attacks the problem of efficient test generation for failure isolation. Some of these papers presented methods to reduce redundancy of exhaustively generated tests to isolate failures or near minimal test generation methods for failure detection, but their methods are impractical to generate tests for actual digital machines. Actual test generation using the method presented in this paper has been done for the ILLIAC IV Processing Element control logic, and is briefly discussed.

PATH GENERATING METHOD

In this section, test generation by the PGM (**P**ath **G**eneration **M**ethod) to a given directed graph will be discussed briefly.

Let us consider a graph with a single input and a single output such as that shown in Figure 1. If this actual circuit has multiple inputs or outputs, we add a dummy input or output node and connect them to the actual inputs or outputs so that the graph has only one input and one output node.

There exist thirteen possible paths from the input node N_0 to the output node N_5 of the digraph in Figure 1, but not all of these are needed to cover every arc of the graph. We arrive at a reduced number of test paths in the following manner.

Starting at the input node, we list all the nodes which are directly fed by the input node, i.e., have an incident arc which originated at the input node, and draw lines corresponding to the arcs between them. Level zero is assigned to the input node and level one to the nodes adjacent to the input node. Nodes directly connected to the level one nodes are then listed and assigned to level two. This step is repeated until all

* This work was supported by the Advanced Research Projects Agency as administered by the Rome Air Development Center, under Contract No. US AF 30(602)4144.

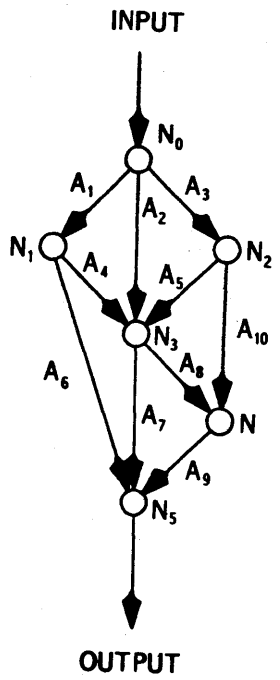


Figure 1—A directed graph

nodes are covered. If a node has already occurred on a higher level or previously on the same level, we define it as a pseudo-terminal node and cease to trace arcs down from it.

Whenever a path from the input reaches a pseudo-terminal node, we complete the path by arbitrarily choosing any route (usually the shortest) which goes from it to the output. Six paths are obtained from the digraph in Figure 1 as shown in Figure 2, where shortest paths are selected after reaching a pseudo-terminal node.

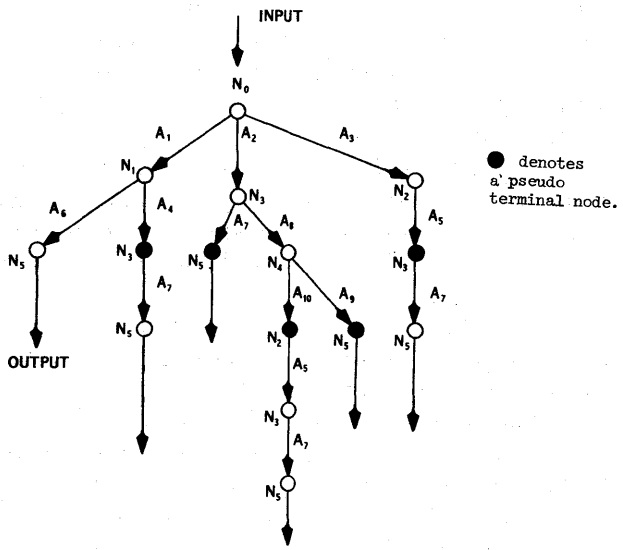
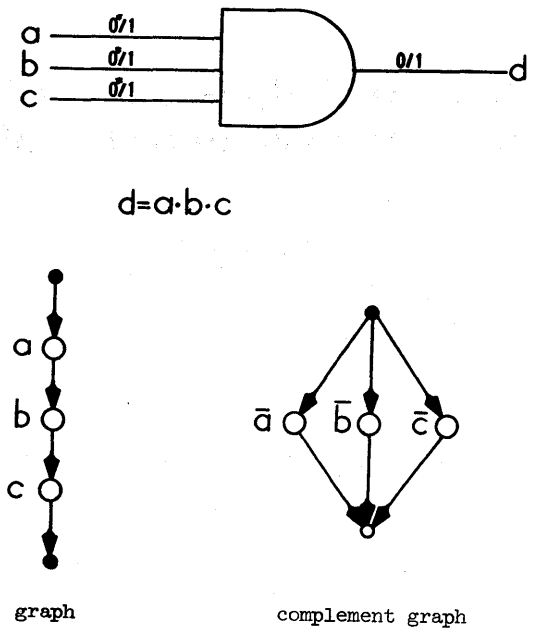


Figure 2—Generated test paths



0 and 1 denote a stuck-at-one and stuck-at-zero failure, respectively, and * denotes a masked failure by the output failure.

Figure 3—AND gate and its graphic representation

The main advantage of this test generation method is that the set of paths generated by the PGM is an irredundant set which is *sufficient for detecting and locating any distinguishable single failure* within any cycle-free graph. It should be noted that any arc in the graph is assumed to be able to be actuated independently for a test path.

GRAPHIC REPRESENTATION OF COMBINATIONAL LOGIC

To apply this PGM to a combinational logic network, a graphic representation of a combinational logic which takes into account stuck-type failures must be used.

An AND gate with three inputs and one output has possible s-a-1 (stuck at one) and s-a-0 (stuck at zero) failures. A s-a-0 failure at output d is indistinguishable

from each s-a-0 failure of the inputs a, b and c, but there exist five distinguishable failures, as shown in Figure 3.

Let us consider the straightforward graphic representations of this AND gate and its complement expression. In this example, a, b and c can denote simple variables or sets of subgraphs representing parts of a logic network. Note that if the four paths are assumed to be paths to test the AND gate where these paths can be actuated independently, all distinguishable faults can be detected and single faults can be located. The graphic representation is slightly modified to demonstrate this, as shown in Figure 4, where $\bar{F}_{d=0}$ means no such fault that the output d is s-a-0.

It is obvious that any one of five distinguishable faults can be located by the four test paths, where only one test path should be completed for each test. To generate a set of test inputs, variable values should be assigned such that only the path to be tested is completed and the rest of the paths are cut off. The test values for the variables (a, b, c) are determined to be (1, 1, 1), (0, 1, 1), (1, 0, 1) and (1, 1, 0) for a three input AND gate.

If one input variable is dependent on another then normally distinguishable failures may become indistinguishable. For example, if variable a is dependent upon variable b, then a s-a-1 failure at input a and a s-a-1 failure at input b may become indistinguishable or undetectable.

Whenever any one of the variables a, b, and c is replaced by a subgraph which represents a part of a logic network, the same discussion is extended to the complex

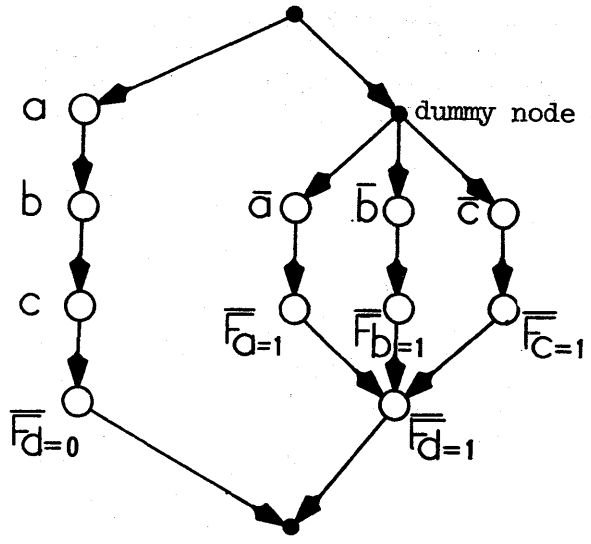


Figure 4—Complex graph for test generation to take into account failures

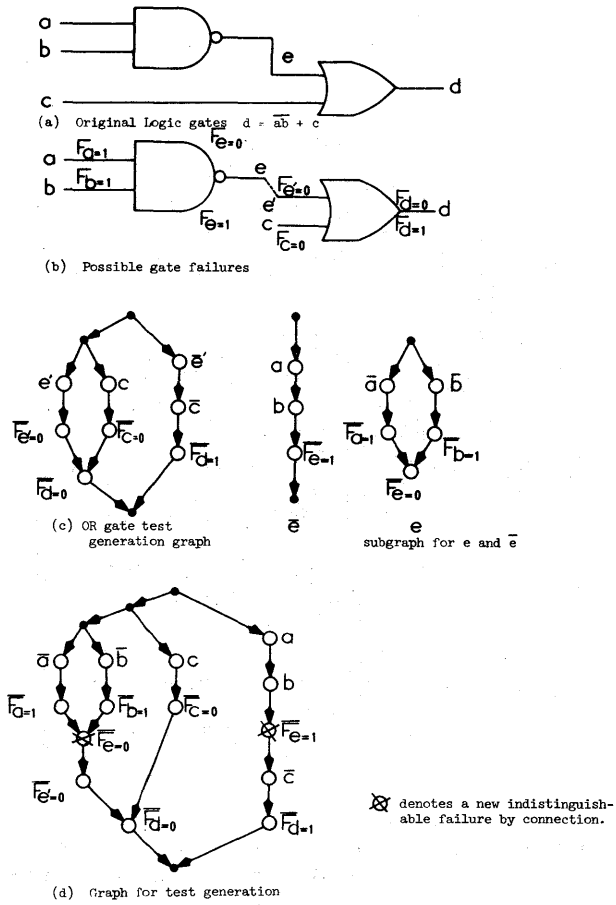


Figure 5—A logic network containing a negation

graph. Also, a similar argument can be applied to an OR gate. If a NOT operation appears between two successive gates, the input variables to the following gate are replaced by the dual subgraph of the preceding gate. Alternatively, the graph will be given directly from equations modified such that negations are driven inward to the primary input variables by applying DeMorgan's Law to the given Boolean equation. For example, the graph for test generation with the logic network in Figure 5a is given as shown in Figure 5d.

The same graph is derived from the transformation of the Boolean equation as

$$d=\overline{ab}+c=\bar{a}+\bar{b}+c$$

and the graph for test generation is given directly by the above equation. It is obvious that distinguishable failures in the original logic network are still distinguishable in the complex graph representation for test generation.

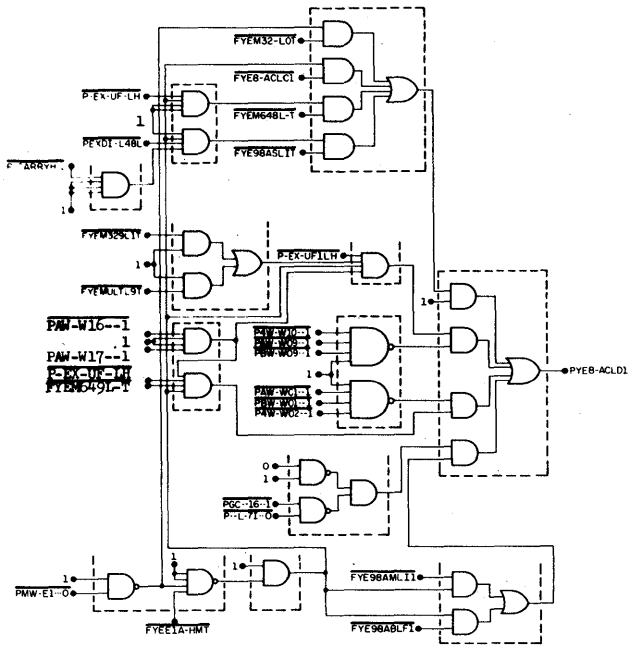


Figure 6—An example of control logic of ILLIAC IV PE (Closed dotted line denotes an IC package and half one denotes a part of IC package)

From the previous discussion it will be noted that if those input variables which correspond to the nodes in a path test through the original graph of a logic function are activated, the combinational logic network will give an output of a logic 1, whereas if the path goes through the complement graph, the output will be a 0. For example, if we set $a=1$, $b=1$ and $c=1$ in Figure 4, the output of the network is a logical 1. If a, b or c stucks at 0, the faulty network will produce output 0 instead of 1. This test can detect single failures a, b, c or output d stuck at 0.

In order to detect the s-a-1 failure of input line a, b, c and output line d, the path tests in the complement graph are required. A s-a-0 type failure of one node in an original graph will become a s-a-1 type failure in the complement graph and s-a-1 type failure of one node in an original graph will become s-a-0 type failure in the complement graph. Now it is clear that the complement graph of the original graph is required for the output stuck at 1.

In test generation methods which have been presented in the past, the relationships between test generation and distinguishable failures in a combinational network were not clearly established. The main advantage of the graphic representation of a combinational network (including the complement expression) is that the graph contains failure information explicitly as discontinuities of arcs or nodes instead of s-a-0 and s-a-1 failures in the original combinational logic network.

TEST GENERATION FOR COMBINATIONAL CONTROL LOGIC

The output of any portion of a computer control logic is usually governed by many input conditions, but the fan-in of a typical logical element is usually restricted to only a few inputs. This causes the number of gate levels necessary to generate a function to increase and the structure of control logic becomes tree-like. The network shown in Figure 6 is a typical control logic of the ILLIAC IV PE. Since there are about 50 distinguishable failures in the network, about 50 iterations of a path sensitizing would be required by conventional technique, or more than 8000 terms would have to be handled by Armstrong's method.[2] In both cases, neither the irreducibility of tests nor the isolation capability of distinguishable failures would be guaranteed.

The network of Figure 6 is translated into the graph of Figure 7 and Figure 8, from which the PGM will generate tests, and the irredundancy and isolation capability of the generated tests are guaranteed as well as the simplicity of the algorithm.

To make a test path in the graph, the variables on the path under test should be actuated and the rest of the paths should be cut off. If the original logic network does not have a complete tree structure, a few conflicts may occur in assigning values to variables to

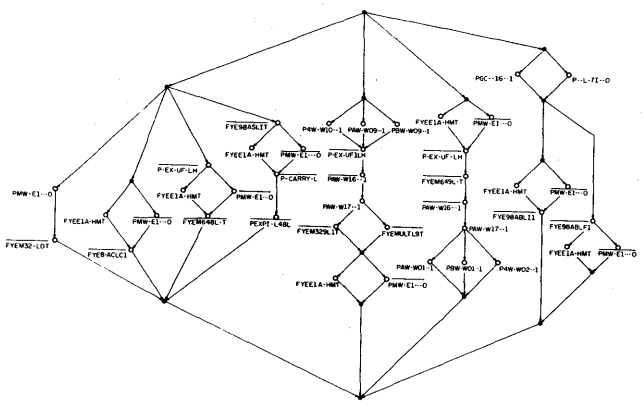


Figure 7—A graph representation of Figure 6 logic diagram

make a test path generated by the PGM. These may easily be resolved, as will be shown later.

Transformation of boolean equations to arc descriptions

The description of a combinational logic network is assumed to be given by a set of Boolean equations using the operators AND, OR and NOT.

For example, from Figure 6 of a part of the control logic of the ILLIAC IV PE, the Boolean equation is

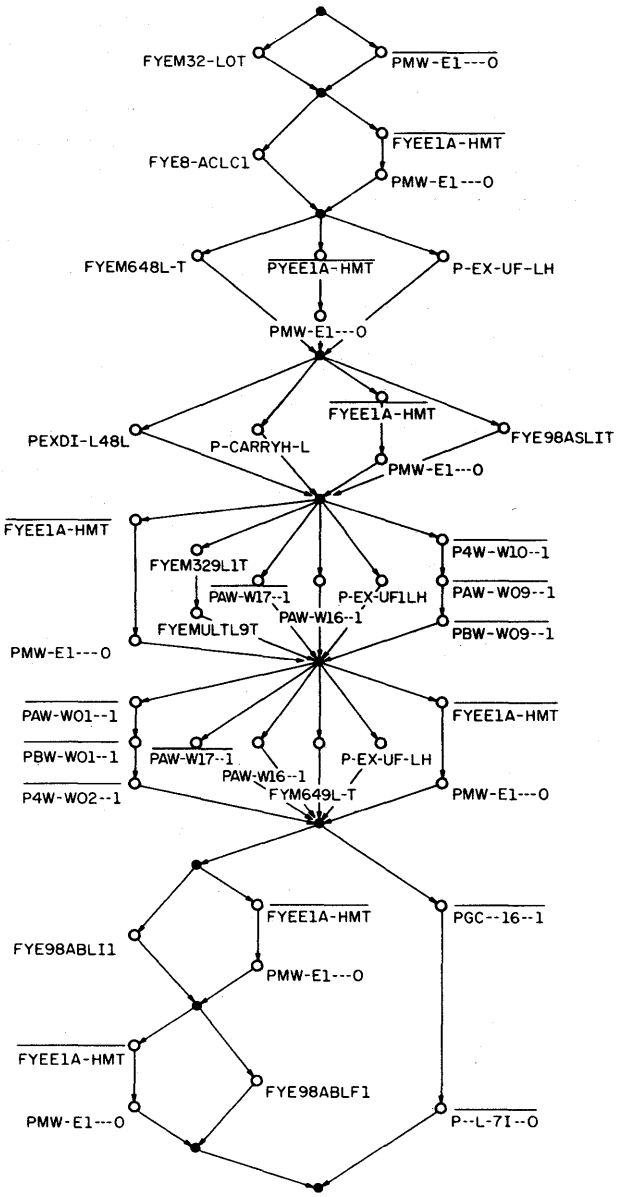


Figure 8—A complementary graph of Figure 6 logic diagram

```
PYE8-ACLD1 ←(((PMW-E1---0 AND NOT FYEM32-LOT) OR
            ((FYEE1A-HMT OR NOT PMW-E1---0) AND NOT
             FYE8-ACLC1) OR
             ((NOT P-EX-UF-LH AND
             (FYEE1A-HMT OR NOT PMW-E1---0)) AND NOT
             FYEM648L-T) OR
             (NOT FYE98ASLIT AND
             ((FYEE1A-HMT OR NOT PMW-E1---0) AND NOT
             P-CARRYH-L AND NOT PEXDI-L48L))) OR
             ((P4W-W10--1 OR PAW-W09--1 OR
             PBW-W09--1) AND
             (NOT P-EX-UF1LH AND
             (NOT PAW-W16--1 AND PAW-W17--1) AND
             (NOT FYEM329L1T OR NOT FYEMULTL9T) AND
             (FYEE1A-HMT OR NOT PMW-E1---0))) OR
             (((FYFE1A-HMT OR NOT PMW-E1---0) AND NOT
             P-EX-UF-LH AND NOT FYEM649L-T AND
             (NOT PAW-W16--1 AND PAW-W17--1)) AND
             (PAW-W01--1 OR PBW-W01--1 OR
             P4W-W02--1)) OR
             ((PGC--16--1 OR P--1-7I--0) AND
             (((FYEE1A-HMT OR NOT PMW-E1---0) AND NOT
             FYE98ABLI1) OR
             (NOT FYE98ABLF1 AND
             (FYEE1A-HMT OR NOT PMW-E1---0)))));
```

Figure 9—Squeezed equation of Figure 6

derived* and this equation was then 'squeezed' by a program as shown in Figure 9, where logical constants (used to disable unused gate inputs) are removed from the functional Boolean expression, and NOT operators are driven into the innermost individual variables of the equation by use of DeMorgan's Law.

Now we try to transform the Boolean equations into the graph descriptions. AND operations are trans-

* In the case of the ILLIAC IV PE design, the Boolean equations are automatically generated from wiring information. This same equation set was also used for debugging the logical design.

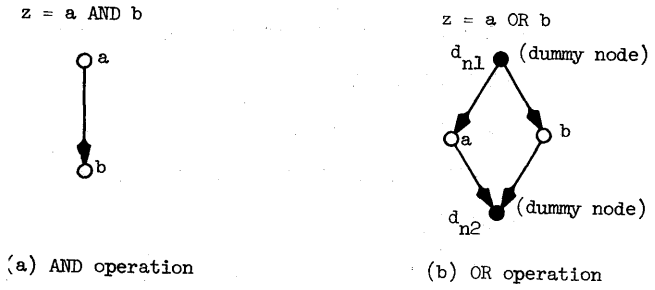


Figure 10—Transformation of the Boolean equations into graph. AND operation in graphic form

formed into series connections and OR operations into parallel connections as shown in Figure 10.

The graphic representation of a combinational logic network is translated as arc description for the input to the PGM program. The AND operation, a AND b, is translated as $b \leftarrow a$, where a is the source node and b the destination node. The OR operation is translated as $a \leftarrow dn1$, $dn2 \leftarrow a$, $b \leftarrow dn1$ and $dn2 \leftarrow b$, where dn represents dummy node.

In the arc description generation program which we developed, redundant dummy nodes are removed insofar as possible. For example, dummy nodes can be eliminated from the OR operation in the various ways shown in Figure 11 depending on the original Boolean equation.

For ILLIAC IV PE control logic we get 111 Boolean equations. The 111 Boolean equations and their 111 complemented equations can be visualized as 222 subgraphs and all connected to an input node and output node. The arc descriptions of this big graph are processed by a program (PGM algorithm) to produce a set of 464 paths for diagnosis.

Conflict and sneak paths

In a graphic representation, every path on the graph is assumed to be able to be actuated independently to the other paths, but this assumption is not always true in the case of combinational logic network representations.

For example, if there is a variable on a path such that the variable simultaneously completes one portion of the path and opens another portion of the path, that is, the variable x appears as both x and $\bar{x}$ in one path, then no test path actually exists.

In the following theoretical discussion, these problems will be analyzed accurately.

Let z be a Boolean function of the Boolean variables $x_1, x_2, \ldots, x_n$ and expressed as $z = z(x_1, x_2, \ldots, x_n)$. Let P be one of the path tests generated from the arc description of the Boolean function z, and be defined by a set of Boolean variables on the path as $P = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$ where $x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha} \in \{x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$.

A path $P = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$ is said to have a *conflict* if there exists at least one x_i such that $x_i \in \{x_1, x_2, \ldots, x_n\}$, $x_{l_j} = x_i$ and $x_{l_k} = \bar{x}_i$ for x_{l_j}, $x_{l_k} \in \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$.

The conflict in the path will cause some trouble in the assignment of the variables. Most of the time, they can be avoided and this will be discussed in the next section.

Let $P = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$ be one of the paths and $(\gamma_1, \gamma_2, \ldots, \gamma_n)$ be one of the value assignments where $\gamma_i = 1$ if $i \in \{l_1, l_2, \ldots, l_\alpha\}$ and the other γ_i values are arbitrarily chosen. If there exists another path P' such that $P' = \{x_{h_1}, x_{h_2}, \ldots, x_{h_\beta}\}$ where $x_{h_1}, x_{h_2}, \ldots, x_{h_\beta} \in \{x_1, x_2, \ldots, x_n\}$ and $x_{h_1} = \ldots = x_{h_\beta} = 1$ after the above assignment, the path $P' = \{x_{h_1}, x_{h_2}, \ldots, x_{h_\beta}\}$ is called a *sneak path.*

The sneak path P' is actually a path with its variables being assigned 1 in addition to the path P in which we are interested. The test values assigned to the variables of the path test $P = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$ can detect stuck type failures s-a-0 or s-a-1 for each literal in the path. For example, if one of the input signals is $x_i (\in P)$ then the test pattern derived from P can detect a s-a-0 failure at input x_i. If one of the input signals is $x_i (\in P)$ and its literal $\bar{x}_i$ appears in P, that is $\bar{x}_i \in P$, then the test pattern derived from P can detect a s-a-1 failure at input x_i. Note that this detectability of the failures associated with the input x_i is under the assumption

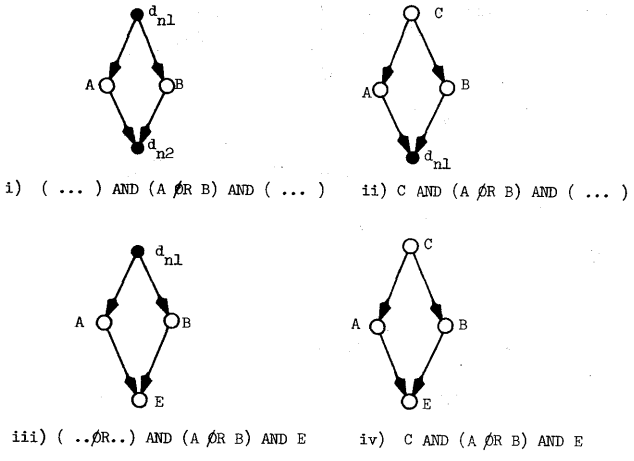


Figure 11—OR operation in graphic form

that there are no conflicts or sneak paths for any test value assignment to the variables in the path. Apparently redundancy in the original logic network causes sneak paths in the graph representation, and these sneak paths reduce the detectability of failures by the path tests.

This is discussed more precisely as follows: Let $P=\{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$, where $x_{l_j}(j\in\{1, \ldots, \alpha\})$ is a literal in the path we are interested in and $P'=\{x_{h_1}, x_{h_2}, \ldots x_{h_\beta}\}(\neq P)$ is a sneak path as defined previously. Let a subset P'' be defined as $P''=P\cap P'$. Then the test value assignment to the variables of path P can at most detect stuck-type failures in the input signals $x_{l_1}, x_{l_2}, \ldots, x_{l_i}\in P''$. The test pattern cannot detect failures in $x_{l_{i+1}}, \ldots, x_{l_\alpha}\in P-P''$.

This is proved as follows:

If a path P is in the original graph, a sneak path P' cannot be in the complement graph. Let P be in the original graph corresponding to the logical network function f and P' be in the complement graph corresponding to the complemented logical network function $\bar{f}$. Then we can express f and $\bar{f}$ as follows:

$$f=x_{l_1}.x_{l_2}\ldots x_{l_\alpha}+R_1$$

$$\bar{f}=x_{h_1}.x_{h_2}\ldots x_{h_\beta}+R_2$$

By sneak path definition $x_{l_1}, x_{l_2}, \ldots x_{l_\alpha}, x_{h_1}, \ldots x_{h_\beta}$ are assigned 1, therefore $f=1+R_1=1$. But $f=1$ contradicts $\bar{f}=1+R_2=1$. So a path P being in the original graph and the sneak path P' being in the complement graph cannot exist. Similar arguments can be applied to prove that a path P be in the complement graph and a sneak path P' being in the original graph cannot exist. So P and P' both must be in the original graph corresponding to the Boolean function f or both in the complement graph corresponding to the complemented function $\bar{f}$.

First, assume that the path P and the sneak path P' are in the graph, not including complement expression, corresponding to the original logic function f. If all the variables in the path P are ANDed together the result is $x_{l_1}x_{l_2}x_{l_3}\ldots x_{l_\alpha}$. This is a term of the Boolean expression of the logic network function f after expansion but before simplification. Similarly for the sneak path P' we get another term $x_{h_1}x_{h_2}\ldots x_{h_\beta}$ for the Boolean function f. Let $f=x_{l_1}\ldots x_{l_\alpha}+x_{h_1}\ldots x_{h_\beta}+R$. Where R is the logic sum of the remaining terms of the Boolean function f.

Since $x_{l_1}, x_{l_2}, \ldots, x_{l_i}\in P''=P\cap P'$, we can rearrange the function f as follows:

$$f=x_{l_1}x_{l_2}\ldots x_{l_i}(x_{l_{i+1}}x_{l_{i+2}}\ldots x_{l_\alpha}+x_{h_{i+1}}x_{h_{i+2}}\ldots x_{h_\beta})+R.$$

According to the value assignment and sneak path definitions, we assign 1 to $x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}$ and $x_{h_1}, \ldots, x_{h_\beta}$ for the variables corresponding to the path P. A test with logic value assignment 1 to x_k can detect a s-a-0 failure at location x_k if the change of the logic value from 1 to 0 will result in a change of the logic values at the output. On the other hand a test with logic value assignment 0 to x_k can detect a s-a-1 failure at location x_k if the change of the logic value from 0 to 1 will result in a change of the logic value at the output. First consider the s-a-0 failure for x_{l_j} where $x_{l_j}\in P''$ and x_{l_j} is positive. Under the value assignment scheme $x_{l_1}=1, x_{l_2}=1, \ldots, x_{l_\alpha}=1, x_{h_1}=1, \ldots$ and $x_{h_\beta}=1$, also $R=0$. If x_{l_j} stucks at 0 and R still remains at 0, this will change the function value from 1 to 0. This corresponds to the change of the output of the combinational logic network from 1 to 0. If R contains such one term in the form of sum of products, as $\bar{x}_{l_j}x_{k_1}x_{k_2}\ldots x_{k_\gamma}$ and $x_{k_1}=x_{k_2}=\ldots=x_{k_\gamma}=1$ and $\bar{x}_{l_j}=0$ under the previous assignment, the stucking at 0 of x_{l_j} will change R from 0 to 1. This keeps the output remain at 1 when the input x_{l_j} stucks at 0. Therefore, the test derived from the path P cannot detect the s-a-0 failure a x_{l_j}. This will not occur when x_{l_j} is a one-sided variable. So the test can detect the s-a-0 failures for those positive one-sided variables x_{l_j} in P''. For the variables $x_{l_{i+1}}, x_{l_{i+2}}, \ldots$ and $x_{l_\alpha}\in P-P''$, the test cannot detect the failures. Assume $x_j\in P-P''$ is a positive variable and stucks at 0. The term $x_{l_{i+1}}x_{l_{i+2}}\ldots x_{l_\alpha}$ becomes 0 but $x_{h_{i+1}}x_{h_{i+2}}\ldots x_{h_\beta}$ is still 1 under the same value assignment scheme. Since all x_i's $\in P''$ are assigned logical 1, the function value still remains at 1 regardless of whether $x_j\in P-P''$ is 0 or 1. So the test cannot detect the s-a-0 failures for any positive variable x_j in $P-P''$.

Similar arguments can be applied for s-a-1 failure of x_i and its literal $\bar{x}_i\in P$. Now we have only proven those paths in the original graph which correspond to the Boolean function f. Similar arguments can be applied to those paths in the complement graph except the function is $\bar{f}$ instead of f.

If $P''=P\cap P'$ is an empty set, the test derived from P cannot detect any failure. Thus this test is useless, and such a path P is said to have *a fatal sneak path* P'.

Test generation

The PGM program generates a set of paths from the arc descriptions of the combinational logic network. These paths will be processed to produce a set of test sequences to detect and locate the failures.

Let z be a Boolean function of Boolean variables $x_1, x_2, \ldots, x_n$ and expressed as $z=z(x_1, x_2, \ldots, x_n)$. Without loss of generality, assume z is positive in $x_1, x_2, \ldots, x_i$ and negative in $x_{i+1}, x_{i+2}, \ldots, x_j$, that is,

x_1 through x_i appear in uncomplemented form and x_{i+1} through x_j appear in complemented form only. But z is both positive and negative in $x_{j+1}, x_{j+2}, \ldots, x_n$. That is, both x_k and $\bar{x}_k (j+1 \leq k \leq n)$ appear in the irredundant disjunctive form of z. For example, if $z(x_1, x_2, x_3, x_4) = x_1x_2 + \bar{x}_2\bar{x}_3 + \bar{x}_4$ then z is positive in x_1, negative in x_3 and x_4 but either positive or negative in x_2. Let us define those variables $x_1, x_2, \ldots, x_i$ and $x_{i+1}, \ldots, x_j$ as one-sided variables and those variables $x_{j+1}, x_{j+2}, \ldots, x_n$ as two-sided variables.

Suppose the PGM program produces paths $P_1, P_2, \ldots, P_m$ from the arc description of the equation $z = z(x_1, x_2, \ldots, x_n)$. Consider only one path P_1. Let P_1 be defined by a set of variables on the path as $P_1 = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$, where

$$x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha} \in \{x_1, x_2, \ldots, x_n\}.$$

Let $x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}$ be defined as *variables inside path* and other variables as *variables outside path.* For example, if we have $z = z(x_1, x_2, x_3, x_4)$ and $P_1 = \{x_1, x_2\}$, then x_1 and x_2 are variables inside path and x_3 and x_4 are variables outside path.

If $P_1 = \{x_{l_1}, x_{l_2}, \ldots, x_{l_\alpha}\}$ is one of the paths produced by PGM program from the arc descriptions of the equation $z = z(x_1, x_2, \ldots x_n)$, then one can get the test from P_1 by the following procedure:

1. Set the positive variables inside path at 1 and the negative variables inside path at 0.
2. Check two-sided variables inside path. If x_i and $\bar{x}_i$ appear in the path, conflict occurs. Stop. If only positive form x_i of the two-sided variables x_i appears in the path, set it at 1. Otherwise at 0.
3. Set the positive variables outside path at 0 and negative variables outside path at 1.
4. Set the two-sided variables outside path at 0.
5. Check for sneak paths.
6. If a sneak path exists, change one of the two-sided variables. Go back to step 5. If the sneak path still exists after checking all the combinations of the binary values of two-sided variables outside path, check for the fatal sneak path.
7. If no fatal sneak path appears, the assignment of the logic values is good. Therefore, a test is determined.

When the PGM was applied to the ILLIAC IV PE control logic, only six of 111 equations were discovered to have path conflicts. Many of these conflicts may be avoided by rearranging the input cards to the PGM program, since the paths selected depend somewhat on the ordering of the input equations.

Application to ILLIAC IV PE control logic

The ILLIAC IV PE can be divided functionally into three major portions, the data paths, the arithmetic units such as the carry propagate adder, the barrel switch, etc., and the control logic unit. Tests for the data paths and arithmetic units have been generated by other methods.[1]

To diagnose the ILLIAC IV PE completely, control logic tests have been generated by an automatic test generator system which uses the methods presented in the previous sections.

The control logic test generator system consists of the following subsystems:

1. Equation generation and simplification program
2. Transformation program to change Boolean equations into arc descriptions
3. PGM program
4. Test generation program
   a. Conflict checking
   b. Value assignment to variables
   c. Sneak path checking

They are combined into the system shown in Figure 12.

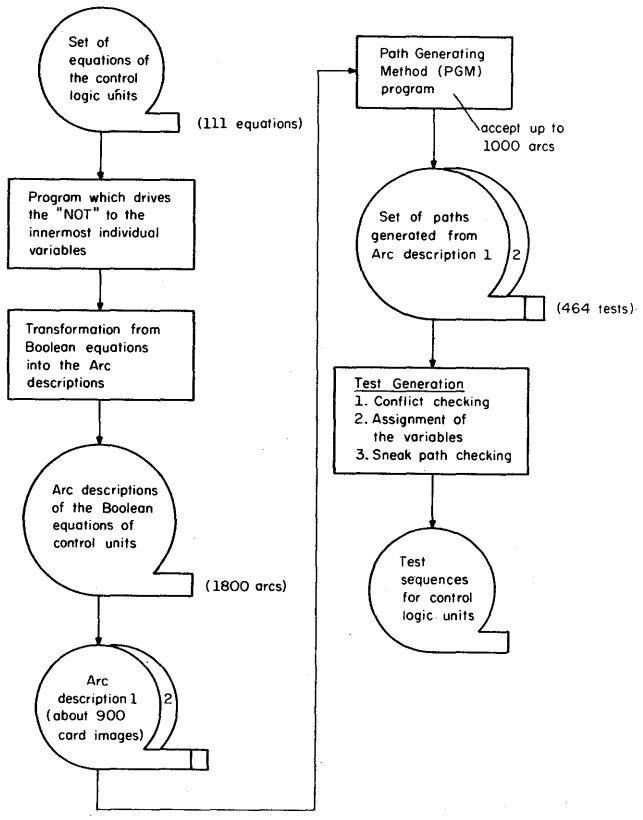


Figure 12—Control logic test generation system

TABLE I—Value Assigned Tests for Combinational Logic Network of Figure 6 Diagram
(THE OUTPUT SIGNAL IS PYE8-ACLD1)

SIGNAL NAMES	PATH NUMBERS 111111111122222222223333333333444 123456789012345678901234567890123456789012
FYE8-ACLC1	11111111110111011111111111111111111111011
FYE98ABLF1	11111100111111111011101010000000000000000
FYE98ABLI1	11111011111101111111110010000000000000000
FYE98ASLIT	11111111011111110111111111111111100011111
FYEE1A-HMT	01011001000011101011100110111011101101011
FYEM329L1T	10111111111111101110000000000010000000000
FYEM32-LOT	11111111110111111111111111111111111111101
FYEM648L-T	11111111101110111111111111111111111100111
FYEM649L-T	11000111111111111101111010000111111111111
FYEMULTL9T	01111111111111111111100000000010000000000
P4W-W02—1	00100000000000000010011011111111111111111
P4W-W10—1	10000000000000010000111111111011111111111
PAW-W01—1	00001000000000000000110111111111111111111
PAW-W09—1	00000000000000000010111111110111111111111
PAW-W16—1	00000111111111101100000000100000100000000
PAW-W17—1	11111000000000100111111110111110111111111
PBW-W01—1	00010000000000000000110111111111111111111
PBW-W09—1	01000000000000000000011111111011111111111
PEXDI-L48L	11111111011111110111100000000000001000000
PGC—16—1	00000101000010000100011010000000000000000
PMW-E1—0	01011001000111101011111111111111111111101
P-CARRYH-L	11111111011111110111100000000000000100000
P-FX-UF1LH	00111111111111101110011111111000001111111
P-EX-UF-LH	11000111101110111011000000010000000001000
P—L-7I—0	00000010000000000000110100000000000000000
	THE FIRST 21 PATHS ARE FOR THE OUTPUT "PYE8-ACLD1" WHICH CORRESPONDS TO THE ORIGINAL GRAPH. THE REST OF THE PATHS ARE FOR THE COMPLEMENTARY OUTPUT "NOT PYE8-ACLD1" WHICH CORRESPONDS TO THE COMPLEMENTARY GRAPH.

Table I shows the variable assignment for the control logic tests in Figure 6.

Test dictionaries for failure location can be generated by a system similar to the test dictionary generator system associated with the PGM program. The test dictionary generation will be reported in a separate paper.

CONCLUSION

The path generation method for test generation for combinational logic has been discussed and an example of the test generation system for ILLIAC IV PE control logic has been presented.

Test generation by means of graph representation of the Boolean functions of combinational logic networks has several advantages over other methods. First, distinguishable faults are explicitly expressed as nodes in the graph. A test which is derived from one path in the graph can detect stuck-type failures, if no sneak paths exist. The nodes in the graph correspond to the failure locations and failure types (s-a-0 or s-a-1) in the combinational logic network.

Second, a complete set of tests for fault location can easily be generated from the graph by the PGM program. If no conflicts or sneak paths exist in the set of paths generated by the PGM, the corresponding set of tests is sufficient for locating failures in the combinational logic network.

This method is a powerful tool for testing tree structure logic networks. If the structure of a logic network is not of the tree type, the conflicts may occur.

A method of checking for conflicts and sneak paths has also been presented. This is used to determine the validity of the tests for the combinational logic network. Conflicts can easily be reduced by replacing tests or rearranging of the PGM inputs after inspection of the generated tests. It is noted that these conflicts are not a result of our approach, but rather a property of the network itself.

Generally, conflicts will be few in control logic networks because their structure is close to a pure tree structure, and no sneak paths exist if there is no redundancy in a logical network.

ACKNOWLEDGMENT

The authors would like to thank Mr. L. Abel for his enthusiastic discussion and our advisor, Professor D. L. Slotnick.

This work was supported by the Advanced Research Projects Agency as administered by the Rome Air Development Center, under Contract No. US AF 30(602)4144.

REFERENCES

1 A B CARROLL M KATO Y KOGA K NAEMURA
A method of diagnostic test generation
Proceedings of Spring Joint Computer Conference pp 221-228 1969

2 D B ARMSTRONG
On finding a nearly minimal set of fault detection tests for combinational logic nets
IEEE Trans on Computers, Vol EC-15 No 1 pp 66-73 February 1966

3 J P ROTH W G BOURICIUS P R SCHNEIDER
Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits
IEEE Trans on Computers Vol EC-16 No 5 pp 567-580 October 1967

4 H Y CHANG
An algorithm for selecting an optimum set of diagnostic tests
IEEE Trans on Computers Vol EC-14 No 5 pp 706-711 October 1965

5 C V RAMAMOORTHY
A structural theory of machine diagnosis
Proceedings of Spring Joint Computer Conference pp 743-756 1967

6 W H KAUTZ
Fault testing and diagnosis in combinational digital circuits
IEEE Trans on Computers Vol EC-17 pp 352-366 April 1968

7 D R SHERTZ
On the representation of digital faults
University of Illinois Coordinated Science Laboratory Report R-418 May 1969

The application of parity checks to an arithmetic control

by C. P. DISPARTE

Xerox Data Systems
El Segundo, California

INTRODUCTION

As circuit costs go down and system complexity goes up, the inclusion of more built-in error detection circuitry becomes attractive. Most of today's equipment uses parity bits for detection of data transfer errors between units and within units. Error detection for arithmetic data with product or residue type encoding has been used to a limited extent. However, a particularly difficult area for error detection has been control logic. When an error occurs in the control, the machine is likely to assume a state where data is meaningless and/or recovery is impossible. Some presently known methods of checking control logic are summarized below.

Methods of checking control logic[1]

Sequential logic latch checking

A parity latch is added to a group of control latches to insure proper parity. The state logic must be designed such that there is no common hardware controlling the change of different latches.

Checking with a simple sequential circuit

A small auxiliary control is designed which serves as a comparison model for the larger control being checked.

Using a special pattern detecting circuit

An auxiliary sequential machine is designed which repeats a portion of the larger sequential machine's states in parallel. This gives a check during part of the cycle of the larger machine.

Checking with an end code check

A check on the control outputs is accumulated and sampled at an end point.

Inactivity alarm

Checks the loss of timing or control signals.

Method of checking an arithmetic control

The application of parity checks for error detection in an arithmetic control appears to have been first suggested in 1962 by D. J. Wheeler.[2] He suggested the application of a "parity check for the words of the store" as an advantage of the fixed store control where parity checks would be applied to each microinstruction word. In a conventional type control, the method of applying parity checks is similar provided that the parity bits are introduced at the flow chart stage of the design. The present method is applied to an Illiac II type arithmetic control which is a conventional control rather than a read only store control. The method gives single error detection of the arithmetic control where errors are defined as stuck in "1" or "0".

THE ILLIAC II

The Illiac II which was built at the University of Illinois is composed of four major subsystems as shown in Figure 1. The Executive Subsystem includes Advanced Control, the Address Arithmetic Unit and a register memory. The Arithmetic Subsystem contains Delayed Control, the Link Mechanisms and the Arithmetic Unit. The Core Memory Subsystem is the main storage. The Interplay Subsystem contains auxiliary storage, I/O devices and the associated control logic.

The Illiac II arithmetic subsystem

The arithmetic Subsystem of the Illiac II shown in Figure 2 performs base 4 floating point arithmetic. The input and output channels carry 52 bits in parallel.

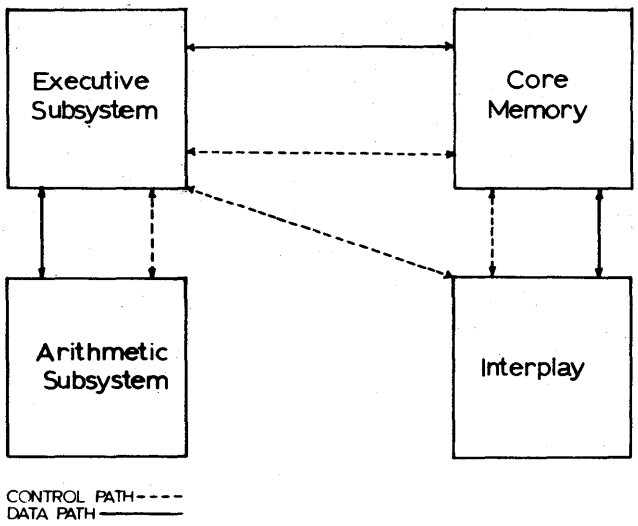


Figure 1—ILLIAC II organization

The first 45 bits of the operand are interpreted as a fraction in the range $-1 \leq f < 1$. The last 7 bits are interpreted as an integer base 4 exponent in the range: $-64 \leq X < 64$. Both the fraction and the exponent have a complement representation. The other input data channel carries a six bit Delayed Control order which specifies the operation performed by the Arithmetic Subsystem.

The Arithmetic Subsystem is composed of three principal units. The Arithmetic Unit (AU) contains the computational logic and is divided into two major subunits as indicated. The Main Arithmetic Unit (MAU) and the Exponent Arithmetic Unit (EAU) handle the fractional and exponential calculations respectively. The second principal unit of the subsystem contains the Link Mechanism (LM) logic. This logic transmits commands from Delayed Control to the Arithmetic Unit (AU). It may further be divided into gate and selector mechanisms and status memory elements. Delayed Control is the third principal unit of the Arithmetic Subsystem. Delayed Control logic governs the data flow in the AU via the LM.

The order being executed by the AU is held in the Delayed Control register (DCR). A new order cannot be transferred to DCR until the order presently held has been decoded and initiated by Delayed Control. If the order requires an initial operand, Advanced Control (AC) determines whether Delayed Control has used the operand presently held in F1(IN). If so, AC places the new operand in IN; otherwise, it must wait. If the order requires a terminal operand (i.e., a store order) AC checks the contents of the OUT register before the store order is completed.

SPINDAC, a small delayed control

Delayed Control is constructed with a kind of logic known as "speed-independent". The theory of speed independence holds that a speed independent logic array retains the same sequential properties regardless of the relative operating speeds of its individual circuits.[3] The theory permits parallel operations while at the same time precluding the occurrence of critical races.

A smaller version of Delayed Control called SPINDAC (**SP**eed **IND**ependent **A**rithmetic **C**ontrol) has been used as a model for the present study. SPINDAC was designed by Swartwout[4] to control a subset of the Illiac II floating point arithmetic instructions. The relatively simple arithmetic unit which SPINDAC controls performs thirteen arithmetic instructions including addition, multiplication, exponent arithmetic, and four types of store orders. For the purposes of this study, SPINDAC has been divided into eight subcontrols as shown in Figure 3. Each of the subcontrols has one or more states. The Add subcontrol, for instance has five states A1 through A5. In general, there is one flip-flop in SPINDAC for each state. The entire SPINDAC has 29 states.

The MAU, EAU, and LM

The essence of this description is due to Penhollow.[5] The Arithmetic Unit (AU) consists of the Main Arithmetic Unit (MAU) and the Exponent Arithmetic Unit (EAU). These two units operate concurrently, but are physically and logically distinct. Both receive their operands from the 52 bit IN register. The first 45 bits of this are interpreted as a fraction, $-1 \leq f < 1$, and is the MAU operand. The last 7 bits are interpreted as

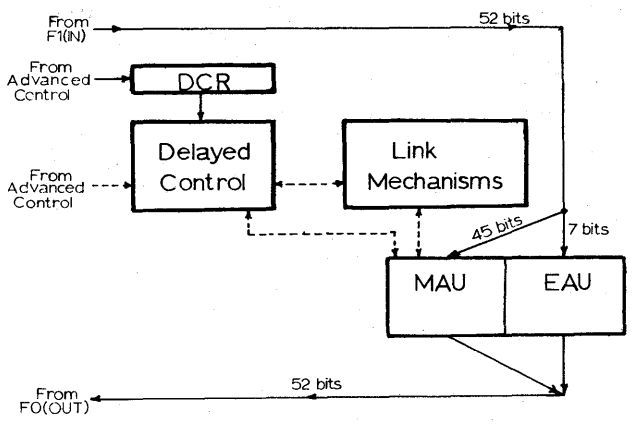


Figure 2—The arithmetic subsystem of the ILLIAC II

an exponent, $-64 \leq X < 64$, and is the EAU operand. The complete floating point operand contained by IN may be expressed as $p = f \cdot 4^x$. Floating point results placed in OUT have the same form. Both f and x are in complement representation.

The block diagram of the Illiac II MAU is shown in Figure 4. Registers A, M, Q and R each have 46 bits, while Ṡ has 48 bits. Since the two adders yield sums in base 4 stored carry representation, A and S also contain 23 and 24 stored carry bits respectively.

The MAU

During the decode step of every Delayed Control order, the gate F1gMEM transfers the first 45 bits of IN to M even though the order does not use an initial operand. The results of the previous operation are generally held in A and Q which represent the primary rank of the double length accumulator. The S and R registers form the secondary rank of the double length accumulator which usually holds an intermediate result at the end of an operation. During the store step of every store order, the RESgRO gate transfers a modified copy of R to the OUT register.

The two adders shown in Figure 4 are composed of base 4 adder modules. The A adder has the contents of the A register as one input and the output of the MsA selector as the other. In either case, the selector output in two's complement representation is added to the stored carry representation held in A or S. A subtraction is accomplished by causing $\bar{M}$ to appear at the selector output and then adding an extra bit in the 44th position.

The selector mechanisms have memory. Once a particular selector setting has been chosen by Delayed Control it remains in effect until a new setting is made. The settings shown in Figure 4 are easily interpreted, provided the outputs of the A and S adders are used in place of the register outputs.

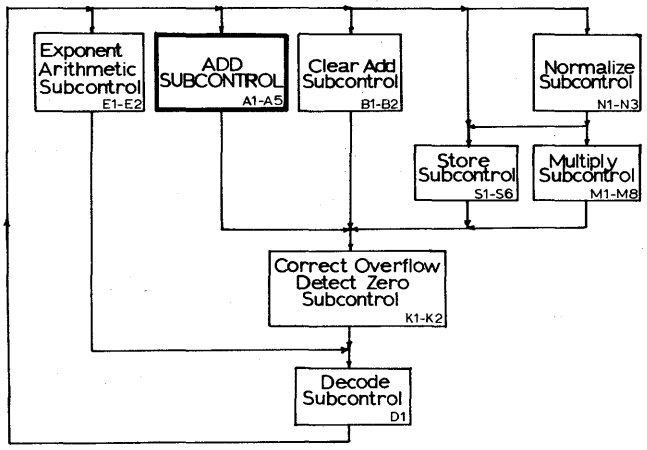


Figure 3—SPINDAC (SPeed INDependent Arithmetic Control)

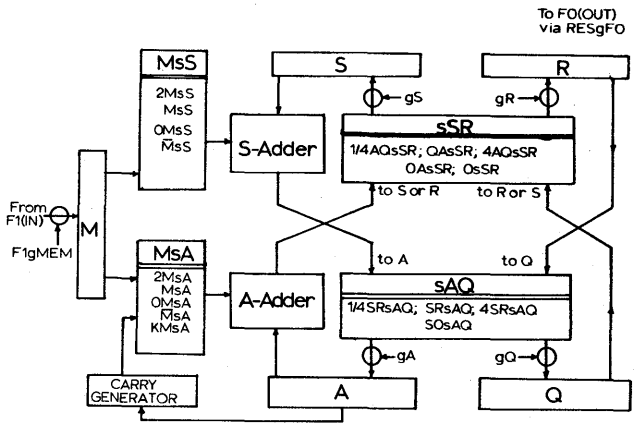


Figure 4—The ILLIAC II main arithmetic unit

The gate mechanisms do not have memory, so they must be activated each time the contents of the associated registers are changed. If the gate is not activated, the register simply retains its old contents regardless of the bit configuration appearing at its inputs.

The EAU

The block diagram of the Illiac II EAU is shown in Figure 5. The EA, ES, EM and E registers each contain 8 bits. The EAU does arithmetic modulo 256. An 8 bit adder (D-adder) with an optional carry into the 0th position provides the capability of doing exponent arithmetic and counting. It accepts the outputs of the EA register and the sD selector as inputs, and yields a sum, D, which can be placed in ES via gES or in E via DgE. The selector sEA controls the input to EA via gEA. The gate mechanism EMgE controls the input to E. During the decode step the contents of F1 are transmitted to EM via F1gMEM. At the end of an operation the exponent of the result is left in E.

The EAU decoder is a large block of logic whose inputs are the outputs of the D-adder. Its purpose is to detect specific values and ranges of the adder output. Knowledge of these values is used in the execution of the floating add instruction. Detection of whether the output is inside or outside the range $-64 \leq x < 64$ is also accomplished at this point. Since knowledge of the previous range or value of d must be remembered during the time the inputs to the adder are changed, gES or DgE will gate the outputs of the EAU decoder into a

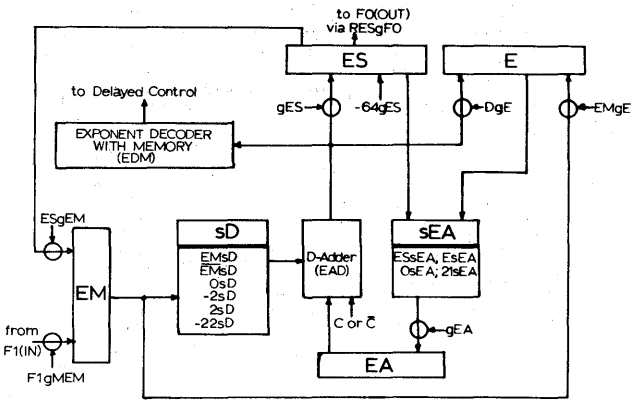


Figure 5—The ILLIAC II exponent arithmetic unit

register called ED. The memory elements of this register are named according to the range of d they represent.

The Link Mechanisms (LM) include gates, selector mechanisms, and control status memory elements. Delayed control directs the data flow and processing in the Arithmetic Unit (AU) via the LM. Delayed Control requests may determine the state of the LM directly or indirectly through the outputs of decoders. The inputs to these decoders may be the outputs of registers, adders, or status memory elements. Selector mechanism and status memory elements remember their present state. The outputs of certain status memory elements influence the branching of Delayed Control. The setting of one or more of these elements during the present control step partially determines the sequence of future control steps.

SPINDAC flow chart

A partial flow chart for the SPINDAC Add sequence is shown in Figure 6. The actions in two of the five states A3 and A4 are indicated. Inside each of the boxes are the control outputs in the form of gating and selector outputs. On the lines leading to each box are the conditional inputs in the form of decoder, status element and selector outputs. They determine which of the control outputs is to be energized. The signals in the boxes on the center line preceded by ALL are always energized when in that state.

The action effected by states A3 and A4 is the alignment of operands for floating point addition and subtraction. Rather than attempting to explain each of the symbols in the flow chart, only the simple case for equal exponents (i.e., no alignment required) will be explained.

The Equal Exponent Case

In Figure 7 the contents of A and Q are first right shifted (¼ AQsSR) then left shifted (4SRsAQ) so that AQ remains with its original unshifted contents. The exponent is gated into ES at A3 by gES in the ALL box. The selector (with memory) EMsD was initially set in state A1 (not shown) which sensitizes this path. In state A4, the exponent is passed back to EA via gEA and ESsEA. EMgE gates the addend (subtrahend) exponent to E. OMsS in A3 effects only a transfer. KMsA is always the final step before leaving the A3–A4 loop. This is to assimilate carries which have been in base 4 stored carry representation until the final pass through the loop. XA controls the exit from the loop (exit if XA=1). The least significant carry into the adder is cleared by C̄. All of these control signals are dependent on the various outputs of the exponent decoders such as d>0, es=2 etc. The actions 2sD and

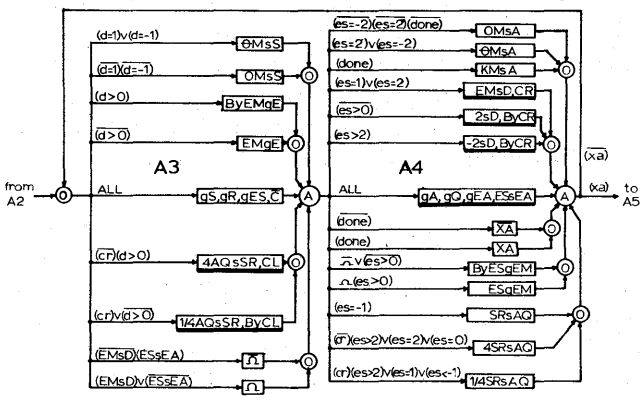


Figure 6—SPINDAC add sequence partial flow chart

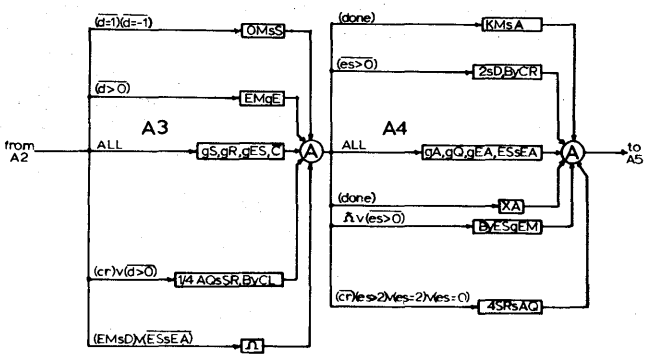


Figure 7—Partial add sequence for equal exponents

Ω (setting status element Ω) are meaningless for the case of equal exponents.

Speed-independent hardware

The logic design procedure used for Delayed Control (and SPINDAC) employs Basic Logic Diagrams (BLD's) developed by Swartwout and others at the University of Illinois.[6, 7] A digest of this work as well as the design for a new error checking BLD is in the Appendix.

In the logic design procedure, each state of the flow chart such as A3 or A5 is associated with a control point (CP). The CP in turn has some associated logic and a flip-flop. Using this terminology, it can be said that the Add sequence has five control points (five states) and the entire SPINDAC control has 29 CP's. Using this design procedure, the entire SPINDAC control can be mechanized with 27 flip-flops and 346 gates.

THE APPLICATION OF PARITY CHECKS

A general arithmetic control model is shown in Figure 8. Here a bit pattern at the output of the control represents a pattern of the gating and selector signals transmitted to the arithmetic unit. The pattern will be a function of: (1) the instruction presently being executed, (2) the conditional inputs and (3) the current step of the control. The control must be protected against two types of errors: first, an erroneous bit pattern at the outputs, and second, an incorrect sequence of the internal states of the control. In a "speed-independent" control, the internal states of the control change one memory element at a time. In most practical designs, this means that the internal states of the control must be encoded with a high degree of redundancy. One systematic way of achieving a speed independent control, for instance,

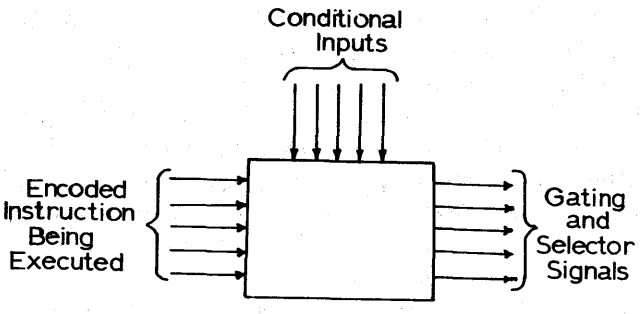


Figure 8—An arithmetic control model

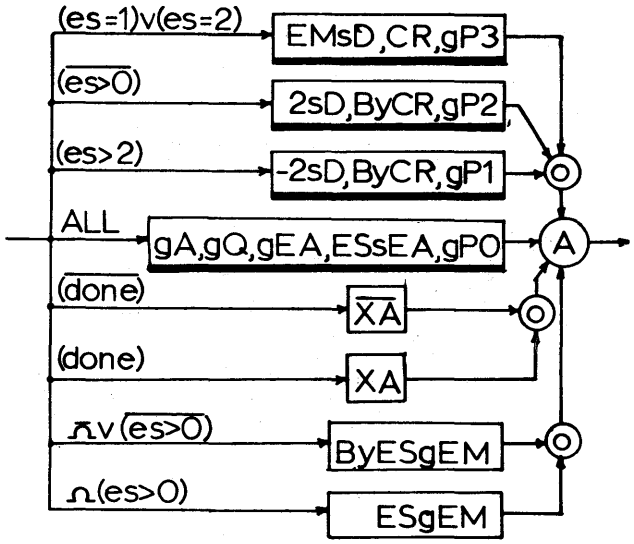


Figure 9—Application of parity checks to simplified control point A4

shifts the single active control point down a shift register. If any two bits (control points) are true at the same time, the control is known to be in an incorrect state. A method is suggested here of applying one or more parity check symbols to the outputs of the speed-independent control so that an erroneous output bit pattern may be easily detected. If the control action with faulty outputs can be detected before the effect has been propagated, a replacement control may be switched in or maintenance may be initiated.

Method

The method of applying single error detection parity checks is explained with reference to simplified SPINDAC control point A4 in Figure 9. In the flow chart, some boxes are entered conditionally. These conditional boxes are the ones which have gating or selector outputs which are energized only if the appropriate conditions are true. The signals gP0, gP1, gP2, and gP3 are gating parity checks which have been chosen according to the following three rules:

1. If a conditional box has an even number of gating and/or selector signals, add an odd parity checking gate to each box (gP1, gP2 and gP3 in the example).
2. If a conditional box has an odd number of gating and/or selector signals, no parity checking gates are added.

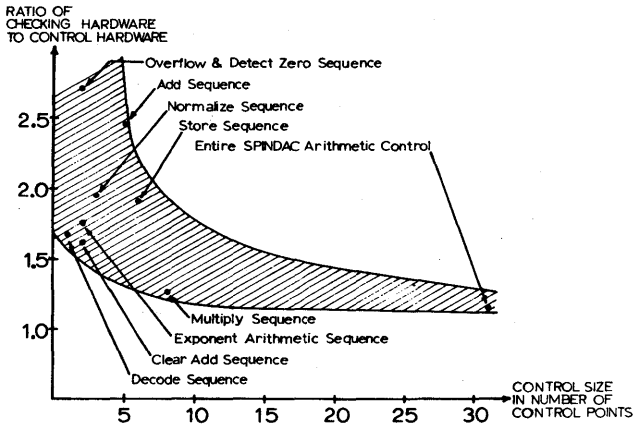


Figure 10—Checking hardware vs. control size

3. Encode the entire ensemble of selector and gating signals for the control point with an even parity gate assuming that only one each of mutually exclusive conditional signals is on. This even parity gate, if required is placed in the ALL box. (gP0 in the example).

The control point as finally encoded has an overall even parity. If a single gating, selector or conditional signal fails, an odd parity results so that detection may be very simply accomplished.

If this procedure is applied to the add sequence, eight parity gates must be added. The boxes to which parity gates must be added in the partial add sequence flow chart are indicated by heavy underscoring in Figure 6. If the procedure is applied to each of the 29 control points in the complete SPINDAC flow chart, 28 parity checking gates are required to be added to its 81 gating and selector outputs.

Hardware cost

While the hardware cost for the checker shown in the Appendix is quite modest, the parity check hardware requirement is significant. In particular, if there are only a few control points in the control with a fairly large number of gate and selector signals, the logic required to check the control may be more than is required for the control itself. The add sequence is a case in point. If we use the conversion factor one flip-flop = 6 gates, the mechanization of the add sequence requires 99 equivalent gates while the number of equivalent gates required for the checker (including the parity checker) is 235. This gives a ratio of the checking hardware to the control hardware of 2.37. This is worse than can be achieved by triplication! If each of the eight SPINDAC main sequences is considered to be a separate control, the lowest ratio of checking hardware to control hardware that can be obtained is 1.27 for the multiply sequence.

If the entire SPINDAC control is checked, the lowest ratio of all is obtained, 1.15. Here a check on 81 gating and selector requests must be made along with 28 parity gate outputs. The majority of the 477 equivalent gates required by the complete SPINDAC checker are used in the parity check decoder. This decoder requires 337 of the total 477. The control itself requires an equivalent of 412 gates giving the ratio of checking to control hardware of $477/412 = 1.15$.

Cost vs. control size

By considering each of the eight principal sequences of the SPINDAC as separate controls, as shown in Figure 3, a plot of their respective ratios versus the number of control points for each sequence can be obtained. This plot is shown in Figure 10. In addition to the eight separate sequences, the plot for the entire SPINDAC control is given. If the entire Illiac II Delayed Control were to be plotted, it is supposed that its ratio would lie somewhat below the 1.15 for SPINDAC. This supposition stems from the fact that SPINDAC was designed to include the use of almost all of the gating and selector signals of Delayed Control so that their number would not appreciably increase for the larger control. Another observation which seems to indicate a decreasing ratio for larger controls is that the number of parity gates required appears to increase at approximately one parity gate per control point added.

Time penalty

The time penalty associated with the checker is also a significant consideration. The parity checkers considered above were tree-like structures used to obtain a minimum hardware count. For instance in the complete SPINDAC checker which required 477 equivalent gates, eight logic levels were required to complete the parity check. If the check of the control is to be made in an "on-line" mode, this delay could be intolerable. For example, if the micro-operation being checked required only three logic levels for its control, an additional five levels would have to be waited before proceeding. This could slow execution more than a factor of two.

A compromise approach is to permit the gating and selector signals to be passed on to the arithmetic unit and to complete their parity check in parallel with actual

execution. For the final step in the sequence, execution could be held up so that erroneous results from a faulty control would not be propagated outside the unit. With this approach, arithmetic execution for the checked unit can be made close to the execution rate of the unchecked unit.

SUMMARY

For the rather small model of SPINDAC which was considered, the amount of hardware required for an arithmetic control with built-in single error detection is 2.15 times the size of an unchecked control. Indications are that the ratio would be lower for a larger control. Though the control can be designed to inhibit propagation of control signals until they have been checked, a better design appears to be one where each sequence is allowed to proceed in parallel with the checking of control signals. Release of arithmetic results is held up until the control outputs for the final step of sequence have been checked.

Though the cost in gates for this method appears to be quite high, it offers the advantages of ease of design and protection against conditional input failure. Since the input gating logic to the flip-flops is separately designed, a sequential error will often manifest itself by a control "hang up." The degree of error localization is also enhanced by the control point organization. When an error is detected, it will most likely be in the conditional inputs or outputs associated with that control point.

ACKNOWLEDGMENT

The author wishes to acknowledge the contribution of Dr. Algirdas Avizienis who suggested the use of a speed-independent control as a model.

REFERENCES

1 F F SELLERS M HSIAO L W BEARNSON
Error detecting logic for digital computers
McGraw-Hill New York 1968 Chapter 13

2 D J WHEELER
Read-only stores for the control of computers
Proceedings of the Second International Conference on Information Processing Munich August-September 1962

3 D E MULLER W S BARTKY
A theory of asynchronous circuits
Proceedings of an International Symposium on the Theory of Switching Harvard University April 1957 Annals No 29 of the Computation Laboratory of Harvard University Cambridge Massachusetts Harvard University Press 1959

4 R E SWARTWOUT
Further studies in speed-independent logic for a control
University of Illinois Graduate College Digital Computer Laboratory Report No 130 Urbana Illinois December 13 1962

5 J O PENHOLLOW
The arithmetic subsystem of the new Illinois computer
University of Illinois Graduate College Digital Computer Laboratory Report No 160 Urbana, Illinois January 24 1964

6 D B GILLIES
A flow chart notation for the description of a speed-independent control
Proceedings of the Second Annual Symposium on Switching Circuit Theory and Logical Design Detroit Michigan October 1961 AIEE Publication S134

7 R E SWARTWOUT
One method for designing speed-independent logic for a control
Proceeding of the Second Annual Symposium on Switching Circuit Theory and Logical Design Detroit Michigan October 1961 AIEE Publication S134

APPENDIX

The BLD logic design method

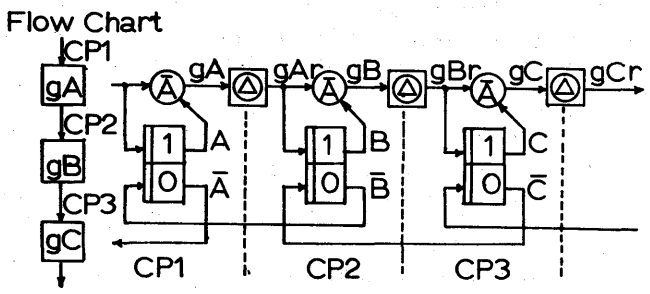
The logic design procedure used for the Illiac II arithmetic control employs Basic Logic Diagrams (BLD's) developed by Swartwout and others at the University of Illinois. Some of these BLD's as well as a new one for control logic checking are described below.

It will be noted by examination of Figures 2 through 4 that several standard and nonstandard symbols are used. The symbol A is used for an AND circuit and O for an OR circuit. The memory element used in control is designated as a vertical rectangle with a "1" indicating the true side and a "0" indicating the false side. It is said to be in the "1" state when the "1" output is a "1". These memory elements are Eccles-Jordan devices; however, they are different in that they react to an input zero rather than a one. Logically the element has two NAND elements cross coupled. This same element can be built with a reply signal as shown in Figure 4, Setting of Status Memory Elements.

The triangle enclosed in a circle and square is used to designate the logical equivalent of the gate reply logic. That is to say, when a gate is energized, the reply duplicates the input signal after some time delay. There may be many combinatorial circuits and amplifiers involved in the gate logic; however, the total system is equivalent to a delay.

A sequencing control

In order to assist in understanding the basic sequencing logic, an example is given in Figure 1. As shown in its flow chart, this logic realizes the sequential energization



A	Ā	gA	gAr	B	B̄	gB	gBr	C	C̄
1	0	0	1	0	1	1	1	0	1
1	0	0	0	0	1	1	1		
1	0	0	0	1	1	1	1		
1	0	0	0	1	0	1	1		
1	1	0	0	1	0	1	1		
0	1	0	0	1	0	1	1		
0	1	1	0	1	0	1	1		
0	1	1	1	1	0	1	1		
0	1	1	1	1	0	0	1	0	1
				1	0	0	0	0	1
				1	0	0	0	1	1

Figure 1—An example of a sequencing control

of gates. The table of states is given to show the normal progression of a signal through such a circuit.

In this type of logic, the quiescent state of the memory elements is the "0" state and thus the NAND normally has a "1" output causing all of the gate replies to also be "1". The first state shown in the table is for the condition that the A memory element at Control Point 1 (CP1) is active (in the "1" state) and the NAND of CP1 has gone to "0" but the gAr (gA reply) and all other signals are in their quiescent state. When the gAr signal changes to "0" this does not excite the next NAND but it does cause the B memory element to turn over. Note that the B memory element has the temporary output state of 1–1 during the transition; however, the zero side of the element is not excited until the one side has changed to one. When $\bar{B}$ changes to zero, memory element A becomes excited and goes to the quiescent state of "0" which in turn causes the NAND to go back to "1", which causes the gate to turn off and its reply to go to "1". Thus the sequencing control has moved from control point 1 to control point 2 and the same sequence of events will occur with the B and C memory elements and the gate gB.

It is important to note that with regard to the action

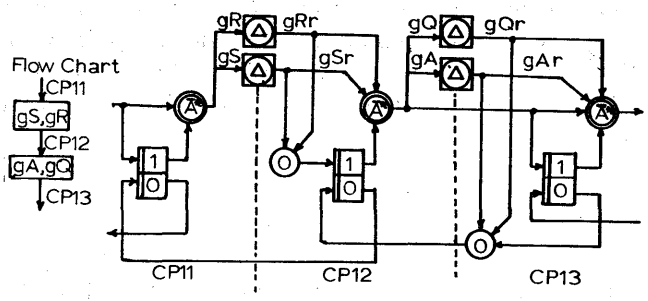


Figure 2—Two forms of parallel operation

of the gate (gA), the sequencing step from control point 1 to 2 had two substeps. In the first substep the gate is turned on and a reply signifying this fact is received in the control, then follow some control actions culminating in turning the gate off. The first substep ends when the reply is received indicating that the gate is on, and the second substep ends when the reply is received that the gate has turned back off again. These two same substeps are associated with every control step regardless of whether the operations performed are those of gating, changing selectors or setting memory elements. That is to say, regardless of the nature of the devices controlled, the reply signal will eventually duplicate the input signal. Thus this example, and in fact all of the basic logic diagrams, while given in terms of gates is an example of any sequencing operation.

Common BLD features

For convenience sake, the sequencing portions of control have been divided into areas called control points (abbreviated CP). A CP includes the NAND and such other logic as is necessary to perform the con-

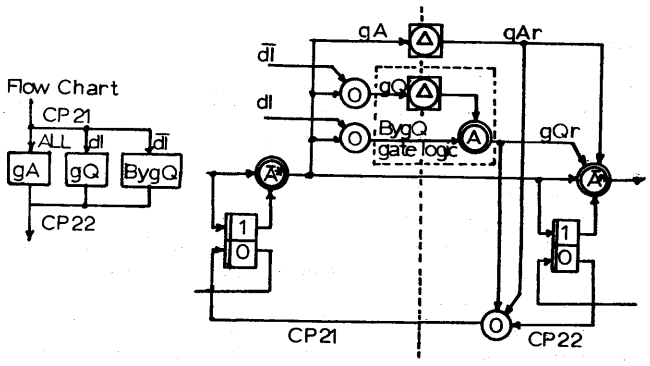


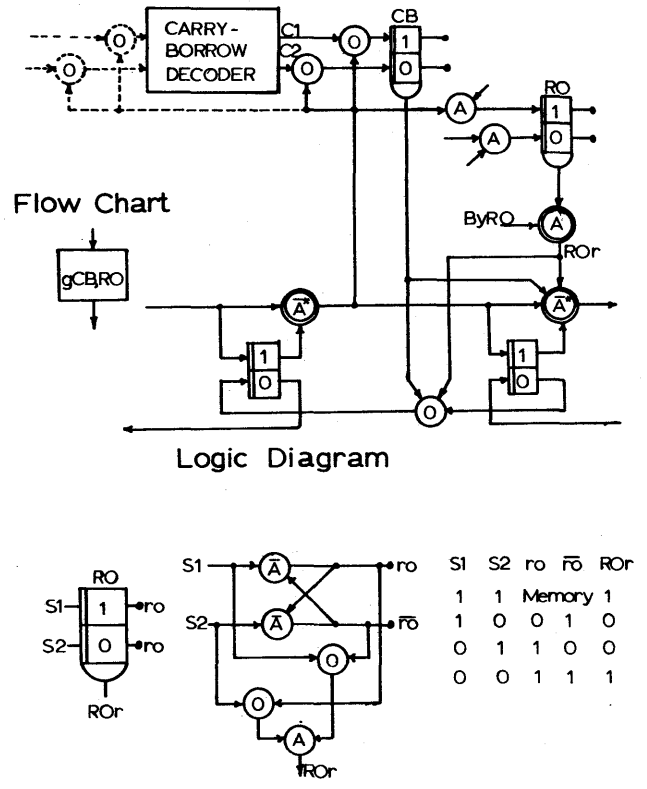
Figure 3—Conditional operation and bypassing

ditional and sequencing operations. The CP numbers appearing on the BLD logic drawings and flow charts are there for reference purposes and the dashed lines are used to separate the various CP's.

The logical symbolism used is discussed in connection with the above example. In addition to these, two other symbols of considerable importance are found throughout the BLD's. A circular symbol with only one circle implies a nonrestoring circuit (non-restored logic level) whereas a symbol with two concentric circles implies a restoring circuit. Some of the restoring circuits also have an asterisk after the logical description. These circuits have shifted thresholds which were designed to maintain speed-independent operation at a point where a signal drives two restoring inputs.

BLD descriptions

Figure 2 shows two different methods for performing two gating operations at the same time. The logic shown for CP's 12 and 13 is preferred over that for CP's 11 and 12 since it operates faster. In the first mentioned method the time required for the memory element at



S1	S2	ro	$\overline{ro}$	ROr
1	1	Memory		1
1	0	0	1	0
0	1	1	0	0
0	0	1	1	1

Figure 4—Setting of memory elements

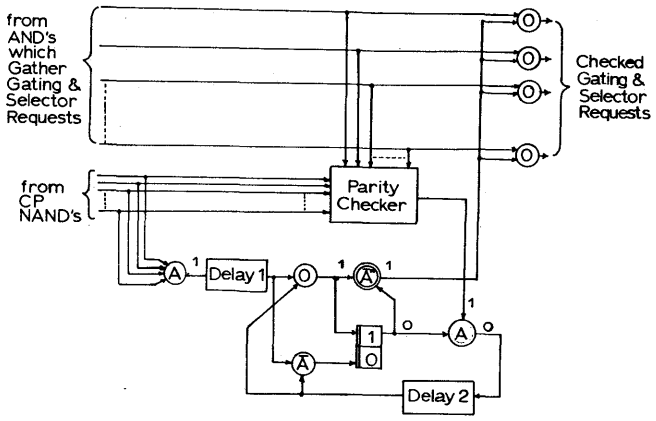


Figure 5—Basic logic diagram for control logic checker

CP 13 to turn over is paralleled with the time required to turn the gate on and for the reply to arrive at CP 13.

One of the most commonly used means of simplifying control logic is to create conditional request signals. The sequencing NAND output opens the conditional OR's and the conditional signal determines the specific operation to take place. One form of conditional request is the Bypass signal as shown in Figure 3. This signal causes no action within the Arithmetic Unit but creates a fictitious reply signal which tricks the sequencing control into reacting as though a machine operation had taken place. Any gate, selector, or memory element can be bypassed through the use of an AND circuit as shown in Figures 3 and 4.

Figure 4 presents both the methods used to set memory elements and a logical equivalent of the memory element with reply signal. The logic of the memory element will be discussed first. The quiescent state for the element is with both inputs "1" and the outputs (1–0) or (0–1). Action will be initiated whenever a request (input "0") is received. If the element is in the state requested by the input "0" the reply will come quickly. If the memory element must change state, a step-by-step check of the signal flow through the circuit will reveal that the reply is the last signal to be generated. Thus when a "0" reply signal is received, the memory element is in the state requested. During the second substep of a control step, the request signal is returned to "1" which causes the reply to change to "1".

Control logic checking BLD

Figure 5 shows a Basic Logic Diagram which has been devised for control logic checking. Delay 1 is required to cover the propagation delay of the decoder.

Delay 2 is greater than Delay 1. The "1's" and "0's" indicate the quiescent states of the checker. The checked gate and selector requests are active when they go to a logical zero. If an error is detected the control is literally "hung-up" by the control logic checker. While the use of delays in the checker makes its logic something less than speed-independent, it may be pointed out that delay lines were also used in the Illiac II to avoid races.

Scheduling in a general purpose operating system*

by V. A. ABELL, S. ROSEN and R. E. WAGNER

Purdue University Computing Center
Lafayette, Indiana

INTRODUCTION

In recent years there has been a great deal written and published about scheduling and storage management in time sharing systems. During the same period there has been a significant trend toward the development of more general purpose operating systems on large computers. Such systems support a high volume batch processing operation and at the same time provide modes of computation usually associated with time sharing systems. They are multiprogramming and multiprocessor systems that execute jobs that enter the job stream from local and remote card readers, and from local and remote on-line consoles. Some jobs are interactive during execution and some are not. Many jobs use interactive file creation and editing and debugging processors even though they are basically batch jobs.

This paper describes some aspects of an operating system of this type that is now running at the Purdue University Computing Center on a CDC 6500 supported by an IBM 7094. The paper deals mostly with the scheduling mechanisms and strategies used in the system. These mechanisms and strategies are probably not new, since all kinds of scheduling disciplines have been proposed and discussed in the literature.[1] However, we believe that this is the first time that scheduling and job movement techniques of the type described here have been implemented and used in a very large system with the high job volume and diversity that characterize a large university computing center.

* The work described in this paper was partially supported by the National Science Foundation under Grant No. GJ-245 for the establishment of the Indiana Regional Computer Network. It was also partially supported by Control Data Corporation through a research grant to the Purdue University Computing Center.

The Purdue MACE operating system

The Purdue MACE operating system is based on the MACE[2] operating system which was originally designed by Mr. Greg Mansfield of the Control Data Corporation.

MACE is an outgrowth of the first operating system for the Control Data 6000 series that was developed at CDC's Chippewa Falls Laboratory.[3] The underlying design of that first system, the Chippewa Structure, has formed the basis for several of the most successful operating systems for the CDC 6000 series. These include SCOPE 2.0, SCOPE 3.0–3.4, and MACE.

The Chippewa Structure is successful, to a large degree, because it is closely integrated with the unique hardware organization of the CDC 6000 series.[4] That organization consists of one or two central processors (CPU's), and ten peripheral processors (PPU's), all of which share a large, fast central memory of 60 bit words. The CPU minor cycle time is 100 ns, while for the PPU it is one microsecond.

The peripheral processors each have a full instruction complement, including arithmetic, shift, and input/output instructions, and 4,096 12 bit words of private storage. They share access to twelve, one megacycle 12 bit wide data channels. The PPU's are primarily designed for input/output tasks, communicating through the common central memory with the CPU, which is used mainly to perform computational tasks for executing programs.

CDC markets several variants of the 6000 design, each of the same structure, differing from the others only in CPU configuration. The 6600, the fastest system, has a CPU with parallel arithmetic units. The 6400 has a slower CPU with sequential arithmetic units. The 6500, which is the system in use at Purdue, has two 6400 CPU's. The 6700 has one 6600 and one 6400 CPU.

Central memory in the Purdue 6500 system consists of 65,536* 60 bit words. The memory is organized in phased banks with access time of 100 nsec and cycle time of 1 microsecond.

Central memory organization and the control point

In the Purdue MACE operating system the large central store is divided into a user portion and a central memory resident system area. The system area which now occupies just under 11000 words, contains allocation tables, routine and file directories, a small amount of system central processor code (most of the system executes in the peripheral processors), a number of key peripheral processor routines, and a set of job control blocks, known as control points.

A control point is a pivotal area, occupying 128 words of central memory, through which job execution is controlled, and to which the resources for job execution are allocated. The control point may be thought of as the control element of an individual computer, and the entire set of control points as a division of the hardware machine into a number of separate machines, each of which can execute an independent task.

The number of control points was fixed at eight in the original Chippewa System and was retained at that number in most derivative systems. One control point is allocated to various system overhead functions—storage movement, mass storage space allocation, etc. The remaining control points can be assigned to active jobs, including the control of input-output devices such as card readers, line printers, remote batch stations, and keyboard consoles. While the MACE system retains this control point allocation method, it provides for the optional declaration of as many as 26 control points at system load time.

A job is assigned to an active control point after it has been queued to a system mass storage device (usually a disc storage unit). The resources required for the execution of the job are allocated to the control point. These include central memory space, central processor time, peripheral processor assistance, mountable equipment (tapes, disc packs, etc.), mass storage space, and file pointers.

The resources are allocated to control points through a monitor program which runs in a dedicated peripheral processor. A second dedicated peripheral processor runs a display program, DSD, that provides operator-system communication via a twin screen, display-keyboard console.

The remaining peripheral processors are pooled for input-output and job sequencing functions. Each contains a small resident executive containing communication, overlay loading, and mass storage driver subroutines. The pool peripheral processors constitute one of the resources assigned to control points by the monitor, and execute programs which communicate with the monitor through central memory registers.

The control point area, which occupies a fixed portion of central memory, contains pointers relating to job status, and the resources assigned to the job. Included in the control point area are a 72 word buffer, used to contain the control statements supplied for the processing of the job, and a 16 word area called the exchange package.

The exchange package is used by the system monitor to control CPU allocation. A special hardware instruction, called an exchange jump, permits the monitor to interrupt a running CPU, save its register contents, and load all registers with new contents in a single operation. The exchange jump instruction, which executes in 2 microseconds, uses the read portion of the core memory cycle to obtain a word of register contents from the exchange area, and the write portion to store the previous contents of the corresponding registers of the interrupted CPU.

When a job is at a control point waiting for the CPU, the exchange package area contains the register contents that are required to start or resume processing of the job. When the monitor performs an exchange jump for that control point, the registers are loaded from the control point area, and the control point area is loaded with an exchange package that the monitor uses to return control to the system when the job is interrupted or terminated.

The rapid CPU switching capability provided by the exchange jump operation works in conjunction with a relocation and limit register in each CPU to provide an efficient method of memory allocation. The relocation registers in the CPU permit the assignment of a contiguous region of central memory to a program, which is totally isolated from any other area, and which can be moved rapidly to, from and within the user portion of central memory.

Limitations of the Chippewa structure

While the Chippewa Structure in its basic design permits effective multiprogramming use of the 6000 system, it includes some static elements which seriously limit system performance. The major achievement of the MACE system was the relaxation of some of these control point restrictions. This process has been carried further in the Purdue MACE system.

* The memory is to be expanded to either 98,394 or 131,072 words in the summer of 1970.

The major static control point restriction in the original Chippewa Structure is embodied in the fact that once a job was brought to a control point, that control point was committed to that job until it either completed or aborted. In almost all cases this meant that once a job was brought into central memory it remained there until it was completed. This affected the design of resource allocation and job sequencing to such an extent that control point and job became almost inseparable.

The closeness of the association between control point and job seriously affects the ability of a system of this type to respond to changing job loads. Thus, while the system can schedule jobs to control points on a priority basis, a new job of higher priority which enters the queues normally must wait until terminating jobs release sufficient resources.

Early attempts to resolve the problem resulted in processors which permitted the system operator to manually suspend the processing of a control point job, and to dump its allocated core memory to disc storage in order to permit another to be loaded and processed. This process was severely limited by the slowness and inaccuracy of operator intervention, and by the fact that it did not free the control point even though the job itself was no longer in memory.

The Autoroll system

One of the major advances of MACE over the earlier implementations based on the Chippewa model is the ability of the system itself to suspend a job and free its control point. The process, called rollout, consists of the copying of the complete job status, including control point data, to a mass storage file. The control point thus freed can be assigned to another job. In the reverse process called rollin, a rolled out job can be assigned to any available control point, the data in the file can be copied back into any available area in main memory, and the job can be resumed.

The scheduling mechanism in the MACE system is of the type that has been called preempt resume in some recent publications.[1] Among users of CDC 6000 series equipment it is more frequently referred to as an autoroll system. The basic component of the system is a job scheduler that can interrupt jobs and cause them to be rolled out from main memory to make room for other jobs that, at least temporarily, have higher priority. The queues from which the scheduler selects the jobs that are to be brought into memory consist of input files and rollout files. The major function of the scheduler is to use the autoroll mechanism to control the movement of jobs between the job queue and main memory in such a way as to provide for optimum utilization of system resources.

There are many possible job movement strategies that could be implemented within the framework of such a system. The particular strategy described here is the one now in use in the Purdue MACE system. It seems to function well in the university environment, and provides adjustable parameters that permit fairly significant changes to be made in response to changes in the character of the job mix.

Job movement strategy

The job movement strategy of the Purdue, MACE system is a dual function of the system monitor and a peripheral processor program, the job scheduler itself. The job scheduler executes on a short, periodic cycle (five seconds in the present system). It is also executed whenever a job sequencing operation changes the state of the machine—e.g., a job terminates or a new job enters the input queue.

The job scheduler is priority driven. Each job in the system carries a single, twelve bit priority value, called the queue priority. A large value signifies high priority; a small value, low priority. Several priority classes and values are reserved for identifying jobs in special states, such as being rolled out or in, manually rolled out, or waiting for some operator action.

Each time the job scheduler executes, it constructs a snapshot of the executing, control point environment. This includes data about the jobs running and the resources allocated to them. Against this picture, the job scheduler matches the jobs awaiting execution in the input and rollout queues.

In descending order of queue priority value, the job scheduler compares the resources required by jobs in the queues against those available or in use by jobs of lesser queue priority. In the simplest case, where sufficient unused resources are available, the job scheduler requests the assignment of a peripheral processor to the job by the monitor. That processor proceeds to roll in the job or begin its execution for the first time, while the job scheduler continues to search the job queues.

When a waiting job requires resources in use by executing jobs, the scheduler must consider the nature of the resources required. Many of them, such as central memory, the control point, central processor usage, and file pointer space, can be reassigned, since the rollout file will carry the status of their usage. Others, such as magnetic tape units, remain assigned to the job for its duration for practical reasons.

After the job scheduler has selected a job for which resources can be made available, it constructs a rollout

sequence which will free the required resources. The rollout sequence is built from the list of running jobs whose queue priorities are lower than that of the job being scheduled. Central memory space and control point availability are the two factors considered.

Rollout density is controlled by the system monitor. In the normal job scan cycle, a job marked for rollout is assigned a peripheral processor by the monitor, unless a prespecified number of rollouts are already in progress. In the Purdue MACE system, the monitor limits the number of concurrent rollouts to two.

Once the job scheduler has started a rollout sequence, rather than wait for the sequence to complete, it continues to search for lower priority jobs which can be assigned to control points without affecting the rollout sequence, or starting another sequence. When the scheduler exhausts the lists of waiting jobs, it terminates.

The scheduler is recalled periodically, at the end of each rollout step, or when some other job sequencing operation changes the state of the machine. When recalled, the scheduler builds a new snapshot of the environment, effectively "forgetting" the job which started the rollout sequence. Because the scheduler "forgets" that job, it can respond very quickly to changes in the queues. Thus, for example, if a job enters the queues with a priority higher than the one which started the rollout sequence, that job can be executed first. Or, for example, if a job outside the rollout sequence terminates before the sequence is complete, the job causing the rollout sequence can be assigned for execution as soon as the required resources become available.

On a sub-multiple of its basic period, the job scheduler executes an overlay which adjusts queue priorities. The queue priority adjustment overlay modifies the priorities of jobs in the input-rollout queues, and those of jobs in execution at control points. The modification of priorities for queued jobs is essentially an aging operation, to insure that jobs of equal starting priority and resource requirements proceed on a more-or-less first-in, first-out basis.

The queue priorities of jobs in execution are modified as a major tactic in queue balancing. This modification is a portion of a three level management of job queue priority, in which the queue priority of a job is set to a high value when the job enters the input queue, is dropped to a lower value after an allotment of execution time has elapsed, and is incremented each succeeding time the job reaches a control point.

When a job enters the input queue, it is assigned two queue priority values, a "first pass" and an "execution" priority, both based on its resource parameters. The first pass queue priority is based upon a user specified (but account limited) value, an input increment, and an origin increment. Currently each job receives an input increment of 6000_8 points, and an origin increment of zero for local batch, 100_8 for remote batch, 300_8 for remote teletype, and 500_8 for interactive origination. The user value ranges from zero to 24_8.

The second queue priority value is based upon job parameters and account code classifications. The job parameters include central memory requirement, central processor time requested, and the predicted output volumes. The execution queue priority value is constructed from a table of range increments for each parameter. In general, the larger the parameter the smaller the increment it will add to the execution queue priority.

When the job input file is completed, it is queued at its first pass queue priority value. The execution queue priority value is stored in the job input file. When the job reaches a control point, the execution queue priority is stored with other job description parameters in a control point area. Thus it is available to the queue priority adjustment overlay of the job scheduler.

In scanning control point jobs, the queue priority adjustment overlay is preset to consider those jobs which have accumulated a specified amount of execution time. When a job has reached that level, its first pass queue priority is replaced with the execution value. In almost all cases the result is a drop in queue priority.

Currently, the first pass queue priority is replaced by the execution priority after a job has accumulated a total of twenty five seconds of central and/or peripheral processor usage. With a large input stream volume, the modification usually results in the rollout of the job. However, in the Purdue job mix, 75 percent of all jobs complete before the modification takes place. For the user, the chosen time increment permits rapid turnaround for compilation-debugging runs, and usually guarantees that a job which aborts because of compilation errors will pass through the system very rapidly.

The remaining jobs which do not complete before the queue priority modification takes place must run to completion at their execution queue priority values. Several factors combine to enhance their throughput. The first is a dynamic storage reduction performed by the relocatable loader. This improves job throughput because compilation and loading usually require more memory space than execution and usually complete before the queue priority modification takes place. Thus the additional execution time which the job requires can often take place at the reduced field length set by the loader.

Secondly, jobs are aged by the scheduler's queue

priority adjustment overlay. Thus as a job remains in the queues, its priority gradually increases. Finally, each job which is scheduled to a control point receives a small, additional queue priority increment.

The control point increment, which is currently set to four aging units, is designed to protect the rollin time investment. The job is given a queue priority boost in an attempt to keep it in execution for a long enough time to make its rollin time cost reasonable. Otherwise, one could easily envision a job mix in which rollin-rollout operations enter a rapid cycle, induced by the aging process.

Control point and central processor utilization

The Purdue MACE system typically runs in an eleven control point configuration with one control point allocated to basic system functions as described in an earlier section. Three others are reserved for use by system input-output processors, one for the queuing (spooling) of peripheral I/O, one for remote batch terminal control, and one for PROCSY, an on-line console system. These three control points require small amounts of memory, determined by the number of active devices. They use very little central processor time, and a larger amount of peripheral processor time for the input-output operations required.

The remaining control points are used for the execution of user problems. The two central processors are cycled among active jobs on a round-robin basis. Each job at a control point which requires a CPU is allocated one for a 65 millisecond time slice. The exchange jump operation keeps the switching overhead very low. Typically it is less than 100 microseconds per transfer.

A job that issues an input-output request may retain the CPU for the full time slice and attempt to overlap its own computing with its I/O transfers. Alternatively, it may give up the central processor for the duration of the I/O transfer. A job that surrenders the CPU when it makes an I/O request is given another 65 millisecond time slice as soon as the I/O transfer is completed.

Other algorithms for the scheduling of central processors to jobs are being considered, but so far there is no evidence that the other algorithms provide any advantage over the round robin with a relatively short time slice.

Job mix

Since the jobs that are running in the system may vary greatly in their demands on system resources, it is good scheduling strategy to attempt to maintain a mix of active jobs at control points that require different resources and that make full use of these resources. Ideally there should be one or two jobs whose demands on CPU time are large compared with their input-output requirements, and one or two complementary jobs which require only small bursts of CPU time, and have a great deal of I/O activity involving non-conflicting devices.

The Purdue MACE job scheduler does not now consider these job mix factors in its calculation of queue priorities, since that would require data about the job profile that is not currently available in a form in which it can be used by system routines. Some job mix factors can be introduced manually in the present system through operator typeins that alter the queue priorities assigned by the system.

A more dynamic automatic scheduling algorithm depends on the measurement and efficient encoding of job parameters relative to CPU and input/output and other resource usage on a continuing basis during the course of the execution of each job.

The effectiveness demonstrated by our current use of the priority structure suggests that it would be possible to incorporate job mix factors in the priority value. We are presently considering a priority evaluator system to be implemented as a secondary level in addition to and separate from the scheduler already described. The priority evaluator would use the job profile data, the machine environment, and scheduling constraint parameters to assign priorities which could provide an improved job mix. This type of priority evaluation could be performed at longer intervals, possibly in terms of minutes, could use the faster capabilities of the central processors, and would not affect the ability of the primary scheduler to react to rapid changes in system load.

Tapes, disc packs and permanent files

One of the major advantages of the autoroll system is the fact that it permits the handling of requests for allocation and mounting of tapes and disc packs and the queuing of requests for access to permanent files in such a way that little or no system resources are consumed by a job while it is waiting for equipment to become available or for tapes or disc packs to be mounted.

Consider a job that enters the system with a jobcard parameter that indicates that it will use magnetic tape. The jobcard indicates the maximum number of tape units that will be required in parallel, and to simplify the discussion we shall assume that this number is one. The job is scheduled to a control point based on

its first pass queue priority and processing continues as for other jobs until a call for a tape occurs in the control statement stream. When such a tape request occurs the job is rolled out, and is marked as a job waiting for a tape unit.

When a tape unit becomes available it is assigned to the job of highest execution queue priority that is waiting for a tape. The tape unit remains assigned to that job until the job terminates unless the job itself releases the tape unit prior to termination.

The job is not automatically rolled back in by virtue of the fact that is has a tape unit assigned to it, but its execution queue priority is raised by 300_8 points to help speed it through the system.

While a job has a tape unit assigned it is rolled out every time a tape mount request is processed, and remains rolled out and ineligible for scheduling until the operator mounts the requested tape and types a message to that effect.

Since the number of tape units is quite limited, it is very desirable that a job with assigned tape units be allowed to run to completion as soon as possible. This would suggest that the very highest priority should be given to jobs with assigned tape units. However, this approach might produce time periods in which only tape jobs could be run, a situation that might be intolerable because of the requirements of on-line users, and the stated goal of providing fast turnaround for short debugging runs. As in most aspects of job scheduling, it is necessary to compromise between the very desirable goal of making most efficient use of a resource (such as tape units) and the many other goals established for the system as a whole.

Mechanisms similar to those used for tape staging are used for access to disc packs and for write access to permanent files. The situation is complicated in these latter cases by the fact that more than one job in the system at a given time may require access to a particular disc pack or permanent file.

Console support

A very large number of jobs come into the System by way of PROCSY (**Pur**due **R**emote **O**n-Line **C**onsole **SY**stem). PROCSY, which is described in more detail in another report, uses an IBM 7094 to drive a large number (50 at present) of teletype consoles as a remote job entry system for the 6500. The 7094 creates job files on a common disc pack unit. The jobs are executed by the 6500 and output may be returned to the consoles by way of the 7094. Rapid response during file creation and editing is provided by the 7094. Fast turnaround for job execution is guaranteed by the scheduling strategy that provides a special increment for console origin in addition to the 6000 point first pass increment. If the job is such that it can be completed in less than 25 seconds of CPU and/or PPU time the results will be available at the console very rapidly. If the job takes more than 25 seconds it will probably be rolled out one or more times before completion, and may be in the system for quite a long time. The user at the console could ask that the results be stored in the permanent file system for his later retrieval, or he could simply come back later and list his output file, or he could divert the output file to the high speed printers in the computing center.

In addition to PROCSY there are several interactive systems that can be operating on the System at any given time. These include NAPSS, the **N**umerical **A**nalysis **P**roblem **S**olving **S**ystem, PICLS (**P**urdue **I**nstructional and **C**omputational **L**earning **S**ystem) ALFIE (**A**lgebraic **L**anguage **F**or an **I**nteractive **E**nvironment), and CRT, an interactive graphics system using the CDC 252 graphics console.

The details of handling teletypes and the graphic console are slightly different from those in PROCSY, but the basic system is the same. The same scheduler using the same mechanisms causes the appropriate programs to be rolled in when needed to handle a line of text or some other interaction. They are rolled out when done to free system resources for other jobs. Here again, if the interaction stays within the basic first pass time limit, the response time is very good. If it requires more time per interaction, it is not considered a proper interactive job in this environment, and response time may be very poor.

In most cases the same mechanism that guarantees good turnaround for relatively short batch jobs also provides good response for interactive users without placing an intolerable load on system resources.

Efficiency of job movement

The job movement system discussed here differs from that in most systems using preemptive scheduling techniques in that the whole job is moved as a unit between peripheral storage and main memory. Most systems of this type make use of paging and/or segmenting hardware and a software system that moves parts of a program between peripheral and main memory as required. In most paging systems, a fixed length page of 512 or 1024 words is the unit of information that is moved.

Rollout compared to paging

Consider a job in a paging system. In order for the job to become active a relatively large number of

pages (the working set according to Denning [5,6]) must be loaded. When the job finishes its time slice, the pages that it had been using are scheduled to be rolled out to the paging drum or disc. In an active system it is very likely that all of the storage that was occupied by a job will be needed by other jobs, so that by the time the interrupted job is once again scheduled into core memory none of the pages that it had been using during its preceding time slice are still in core. This situation is almost exactly the same as if the job had been rolled out in its entirety from core memory. Various strategies have been suggested for such paging systems that would roll out all active pages on completion of a time slice. A prepaging strategy would then roll the job, or at least a working set of the job, back in when it was again made active. A system of this type comes very close to an autoroll system in which the whole job is rolled out and brought back in when it is reactivated.

There are some advantages in moving a whole job rather than individual pages. These advantages arise because of the greater efficiency of writing tracks rather than individual blocks during peripheral transfers. In the particular storage system in use at Purdue, a half track consisting of 3136 60-bit words is read or written during every 50 msec disc revolution, possibly after an initial delay of 20-100 msec for seek time. It does not take much longer to move the whole job than it would take to move a few selected pages of the job.

There are of course other advantages, and possibly other disadvantages to paging systems. It is not our intention to discuss these here. Rather it is our intention to point out that autoroll systems are not necessarily inherently less efficient than paging systems.

Use of extended core storage

The efficiency of the autoroll system is enhanced in the Purdue Mace System by the use of Extended Core Storage (ECS) as a buffer for the rollout process.

Extended core storage is a large core storage system designed to be used for streaming data to and from central memory. In its full configuration, with a minimum of 500,000 60-bit words, a streaming rate of 100 nsec per word can be realized. The present 125K ECS at Purdue transfers data at 400 nsec per word. The 250K configuration scheduled to be installed in the summer of 1970 will increase the streaming rate to 200 nsec/word.

In the Purdue Mace system, whenever a rollout is signalled, the entire central memory field length of the job being rolled out is moved to ECS at the full ECS streaming rate. The space that was occupied in central memory is then immediately released for use by other jobs. The contents of the field length that was streamed to ECS is then moved from ECS to a disc storage file at the same rate as it could have been moved directly from central memory to disc storage. Since the transfer rate to disc storage is about 62000 words per second, the use of ECS to buffer the rollout process makes the field length of central memory that is being freed available from several hundred milliseconds to several seconds earlier than in the unbuffered system.

The reverse process of staging input or rollin files in ECS is under consideration, but its implementation would require some major changes in job movement strategies which are now being studied and evaluated.

Performance

Every job that goes through the system causes a sequence of messages to be written in a system file called DAYFILE. These messages tell when the job entered the input queue, how long it took in compilation and loading, how many times it was rolled out, what error conditions were encountered, when the job entered the print queue, etc. The DAYFILE data is used for billing purposes, and also serves as a data base for a number of programs designed to present a picture of the performance of the system. Some of the details of the programs and techniques used will be presented in another report.

The Purdue MACE operating system was phased into operation during the summer of 1969 and took over as the only production system by the end of August. In the first full month of operation, September, 1969, a total of just over 25,000 jobs were run. Of these about 9000 were remote console jobs submitted by way of the newly introduced PROCSY system. By October of 1969 the total number of jobs was over 60,000 of which about 25,000 were PROCSY jobs By February of 1970, the last month for which statistics are available at the time this is being written, the total number of jobs run in the relatively short month had risen to 80,000, of which about 45,000 were PROCSY jobs. The system has been able to absorb this very large increase in console-submitted jobs without seriously affecting its ability to handle batch jobs.

The console system is now almost exclusively a remote job entry system. During the next few months we expect a very large volume of interactive computing to be added as the new interactive text editor and a new interactive algebraic language processor come into full production. Some hardware and system software

changes are being made to accommodate this increased load, but it will be essentially the same system with the same scheduling and job movement mechanisms.

On a typical busy day there are now in excess of 10,000 rollouts. Short jobs, whether entered through card readers or through typewriter consoles get very good turnaround. Longer jobs are mostly relegated to the rollout queues during the main shift in which input activity is very heavy. Most of them are completed during the late night shift when the console system is turned off.

There is of course a very substantial amount of system overhead associated with rolling out and rolling back in over 10,000 jobs per day. This overhead does not seem to be too high a price to pay for the ability to handle interactive jobs, and the ability to implement scheduling strategies like those that give fast turn-around to short debugging runs.

In addition, statistics gathered before and after the introduction of the autoroll system show that the Purdue MACE system is more efficient in its CPU utilization and in its central memory utilization than its predecessor systems.

REFERENCES

1 E G COFFMAN JR L KLEINROCK
Computer scheduling methods and countermeasures
AFIPS Conference Proceedings Vol 32 Spring 1968 p 11-21

2 *Control Data Mace operating system preliminary reference manual*
Control Data Corporation Publication No 44613900

3 *Control Data Chippewa operating system reference manual*
Control Data Corporation Publication No 60134400

4 *Control Data 6400/6500/6600 computer systems reference manual*
Control Data Corporation Publication No 60100000

5 P J DENNING
The working set model for program behavior
Comm of the ACM 11 5 May 1968 p 323-333

6 P J DENNING
Virtual memory
Technical Report Number 81 Computer Science Laboratory
Princeton University January 1970

Scheduling TSS/360 for responsiveness

by WALTER J. DOHERTY

IBM T. J. Watson Research Center
Yorktown Heights, New York

INTRODUCTION

The performance of Release 4 of TSS/360 at the T. J. Watson Research Center was dramatically improved in the three-month period from November, 1969, through January, 1970. The improvements consist of an increase in system responsiveness by a substantial factor together with an increase in throughput. This was achieved by methodically adjusting the parameters of the TSS/360 Table-Driven Scheduler in accordance with the Principles of Balanced Core Time and Working Set Size.

The purpose of this paper is to set forth principles and methodology used to achieve the above initial results. The available evidence of improvement will be exhibited so that each reader can judge for himself the validity of the results.

CONCEPTS AND PRINCIPLES

Performance

Performance is a highly subjective term having a broad spectrum of connotation to different *classes* of people. Fundamentally, performance is the degree to which a computing system meets the *expectations* of the person involved with it. The terms responsiveness, throughput, turn-around time, availability, reliability, number of terminals supported, CPU utilization, channel and device utilization, channel balance, and efficiency are but a few of the concepts that are usually included as aspects of performance.

Responsiveness

To a user of TSS/360, sitting at a terminal, the ability of the system to respond to his commands is his predominant view of performance.[1] He does not care if only one other person is using the system simultaneously with him or one hundred people. If he *expects* that TSS/360 will respond to his EDIT request in two seconds and it takes four seconds, he is usually far more irritated than if he expects a response of ten minutes to some partial differential equation and it takes thirty minutes. The system should be substantially more responsive to those requests to which the user expects an immediate reply, than to those during which he turns his attention elsewhere. *This is the primary assumption* I made when I set out to improve the performance of TSS/360.

On the other hand, if a person expects that his request will take awhile, say ten minutes, he usually turns his attention to other activities, or else he executes it in the background. Since his attention is not concentrated on the response, he doesn't feel large delays nearly as intensely. In the days of batch computing, turn-around times in the range of one to two hours were frequently not distinguished by users who only turned their attention to it every two-and-a-half hours.

Throughput

To a system manager, the number of terminals he can support with TSS/360 is most important. Of course it is also important to consider the categories of work that the users are doing. Thus it is not unreasonable to speak of ranges from two to one hundred simultaneous users when qualified by the work categories. An intuitively obvious but rarely mentioned concept is that, for some categories of trivial work, as responsiveness improves, the number of terminals in use may increase only after a threshold of human performance is reached. That is, if the system is responding at a rate slower than a person's response time, any initial improvements in system response will first result in the individual users getting more work done; only then will the system be able to handle more users at that level of responsiveness. *This is a most important*

consideration. Allowing variable delays in processing longer-running programs to build up as the load increases insures that the very fast ones can constantly provide their users with a fast response. This delay for long running programs is analogous to the concept of turn-around time in batch but is on the order of a few seconds instead of a few hours.

Folded forms of programs[2]

"By the unfolded form of a program we mean the form a program would take if it had available to it a large enough uniform memory to hold both itself and its data. . . . On the folded forms the addresses have been rearranged-folded-to-fit into the smaller address space actually available."[2] In the TSS/360, unfolded forms of programs and data exist in virtual memory. When the program is executed, portions of the program and its data are automatically brought into main memory for execution. This will result in automatic folding of the program if its complete execution space requirements are larger than the main memory available to hold it. It is important to fold a program into as small a space as possible without causing undue inefficiencies (called thrashing) due to an unnatural folding. A high degree of folding is important since it then permits many programs to be folded into main memory simultaneously, thereby providing a potentially significant increase in the level of multiprogramming. The relocation hardware on the Model 67 makes automatic folding possible.

Program locality of reference[3,4,5]

"Program performance on any paging system is directly related to its page demand characteristics. A program which behaves poorly accomplishes little on the CPU before making a reference to a page of its virtual address space that is not in real core and thus spends a good deal of time in page wait. A program which behaves well references storage in a more acceptable fashion, utilizing the CPU more effectively before referencing a page which must be brought in from back-up store. This characteristic of storage referencing is often referred to as a program's 'locality of reference.'"[4] Thus a program's locality of reference influences the degree of folding to which that program can be subjected with a minimal impact on its performance. A program with good locality will run more efficiently in a small execution space than one with poor locality.

The working set of a program[5,6]

The *working set* W(t,T) of a program is the set of pages referenced in the T page references immediately prior to time t. As t progresses, W(t,T) may or may not change; the better a program's locality, the less likely is it that $W(t+1, T) \neq W(t, T)$. It appears natural to try to fold a program in such a way that the program's working set for a given time interval fits entirely in core. Clearly, no more core is needed for that program in that time interval.

The working set size of a program[5,6]

The *working set size* s(t, T) of a program at time t is the number of pages contained in the working set W(t, T). Thus it is quite possible to have the working set change and the working set size remain unchanged. It appears natural to try to refold the program whenever its working set changes. This currently is difficult to do since it is not known in advance just when the working set is changing. In most paging systems, a working set size change is more easily detected. Thus it is possible to detect working set changes at least when the working set size changes. This paper describes a method for doing this. The relocation hardware of the Model 67 makes the application of this concept possible.

To put the concepts of locality of reference, working set and working set size in perspective, consider this:

> During a single interaction between a user at a terminal and TSS/360, several programs are usually executed for that user. Thus for the virtual execution time which spans this interaction, the working set size may or may not change; however, the working set will almost always change several times. Furthermore, for those programs having good locality of reference, the working set size during any one time slice will usually be much smaller than the working set size for the whole interaction time interval. And, in addition, the maximum working set size for all the time slices will probably always be smaller than the working set size for the whole interaction time interval. For those programs having poor locality of reference, the working set size for each time slice may frequently approach the working set size for the entire interaction time interval. Good locality relates more to the rate at which new pages enter W(t, T) than to its actual size.

Balanced core time

Programs having poor locality of reference and a large working set size would greatly reduce the level of multiprogramming if allowed to remain in core for very long periods of time. This would initially appear to

affect throughput. However, responsiveness is also affected since new requests for service cannot be quickly honored if core is currently tied up. Therefore the scheduling strategy proposed here will penalize programs with poor locality and large working set size.

The Principle of Balanced Core Time states that the length of the time slice in terms of virtual CPU execution time for any one task is inversely proportional to the working set size in that time interval. This will minimize the elapsed time that any large program can clog memory. It will also allow programs with good locality to progress very rapidly. If there were no overhead associated with paging these programs in and out of memory this balanced core time principle could be applied in its pure form. But this is not the case. Therefore a minimum time slice length will be established for programs having a large s(t, T) and poor locality to prevent paging overhead from dominating the system. To compensate for this compromise, the duration between such time slices will be considerably longer than the duration between slices for programs with smaller working set sizes. Since the latter constitute an observed large majority, the aggregate paging load on the system will decrease. The multiprogramming level will increase since more core is available more often. Responsiveness will also improve for the same reason. In addition the degree of CPU utilization will increase. These trends should be evident in the RESULTS section of this paper.

Thus a paging system strikes back by reducing the service it provides to those who would misuse it. These scheduling characteristics become more a function of the goodness of the program than of the length of time it has been running. Therefore well-behaving programs will *clearly* be good and bad programs will hopefully become obsolete.

TSS/360 table driven scheduler[7,8]

The TSS/360 table driven scheduler consists of a set of programs in the resident supervisor of TSS/360 used for scheduling, and a table with many rows (levels) of entries. The entries in any one level of the table contain sufficient information to completely control any one task. Each task in the system has another table describing itself to the system. This table is called the **T**ask **S**tatus **I**ndex (TSI). Each TSI has a pointer to some level in the schedule table. Thus by changing the value of that pointer a task is given a completely new set of scheduling parameters. These parameters include:

1. Time, Space, and I/O limits to be used when executing.
2. Priority, Space, and Time values to be used to determine when to schedule a task to be run.
3. Pointers to other levels of the table which will replace the current schedule pointer in the task's TSI when some special condition occurs, or when one of the execution limits is reached.

The supervisor programs used for scheduling are described in the TSS/360 Program Logic Manual for the Resident Supervisor.[7] The schedule table entries are described in the TSS/360 Program Logic Manual called System Control Blocks.[8]

Structuring the table entries

A broad spectrum of scheduling strategies can be implemented by changing only the entries in the schedule table. In this section of the paper one of several strategies implemented at the IBM T. J. Watson Research Center will be described. It attempts to embody the concepts and scheduling principles described above. As such it should not be confused with the scheduling strategy normally distributed with TSS/360.

To better understand the scheduling strategies in the table it is helpful to consider *sets of levels* grouped according to some primary goals of scheduling.

First note that several specific programs are treated separately from all other programs. They are:

1. The System Operator Task
2. The Bulkio Task
3. Logon
4. Logoff

In this initial work not much attention was paid to applying the above scheduling concepts and principles to these programs.

All other programs are divided into two categories, interactive and batch. In general, the same sets of levels exist for both. The only differences are:

1. Interactive Programs have priority over batch.
2. Initially, interactive programs have greater urgency to get started than do batch.
3. The number of batch programs that are allowed to be run simultaneously is arbitrarily restricted to leave space capacity to handle anticipated interactive programs.

With these exceptions, the following applies for scheduling interactive as well as batch programs.

The interactive sets of table levels are the Starting

Set, the Looping Set, the AWAIT Set, the Holding Interlock Set, and the Waiting for Interlock Set.

The Starting Set

The Starting Set of table levels are used to handle new inputs from the terminal. This is somewhat similar to the pipeline of M. V. Wilkes.[9] This set of table levels has a twofold function:

a. Facilitate a fast reply to the terminal if possible, and
b. make an initial judgment of the current working set size of longer running programs so the best entrance to the Looping Set of table levels can be chosen for this program.

This is accomplished by several successive table levels with high priority, small execution time limits (say 100ms.), and increasingly larger core space limits (say 16, 32, 48 pages). Each program as it enters from the terminal will progress upward through these levels each time it exceeds its space limit.

Whenever it exceeds its time limit at any of these levels, the space limit of that level is used as the estimate of the current working set size of that program. That program is then considered to be a longer running program. Its future execution will be controlled by the Looping Set of table levels.

If the program exceeds its largest space limit, the largest allowable working set size (currently 64 pages) is used as the first estimate for future execution under control of the Looping Set of table levels.

Any time the program finishes it is returned to the initial Starting Set table level for the next input from the terminal.

The Looping Set

The Looping Set of table levels performs three significant functions:

a. It uses the schedule table parameters to follow the working set size of each program by regularly over- and underestimating its time and space requirements in a minimal fashion in accordance with the balanced core time principle.
b. It causes the load generated by long running programs to be spread out in time to allow Starting Set entries to be processed quickly.
c. Finally, it optimizes the CPU utilization and penalizes bad paging programs by causing programs with minimal paging requirements to be selected for running far more frequently than those with large paging requirements. This penalty only occurs when the working set size is large and the program's locality of reference is poor. The Looping Set of table levels quickly detects any change in these situations and dynamically adjusts to them. Thus few programs are penalized throughout their execution, while most receive consistently good service.

The AWAIT Set

The AWAIT set is a special set of table levels reserved for those tasks doing tape I/O and other kinds of AWAIT operations. There is a parameter in each table level called AWAIT extension. This parameter is an elapsed time interval during which the current working set pages of a program are kept in core while the program is idle in the AWAIT state. Since this can cause severe elongations of real time compared to virtual time, smaller values of virtual time are allotted in this set of table levels than for a task of the same working set size in the Looping Set.

The Holding Interlock Set

This set of levels is another special set reserved for all programs which are currently holding interlocks on some system resource. Programs running from this set have high priority so that the interlocked resource may be quickly released. I currently assume that the working set size will not change significantly while holding these interlocks. This needs further investigation.

The Waiting-for-Interlock Set

This is a special set of levels for those programs which are waiting for interlocks currently being held by other programs in the Holding Interlock Set. Programs controlled by this set of table levels will be infrequently considered for dispatching until the interlock is released. The same assumption about insignificant change in the working set size is made here as in the Holding Interlock Set.

TOOLS AND THEIR USE

Tools

The tools used in this work were:

1. The Carnegie Mellon Simulator (called SLIN, not CMS)
2. Conversational SIPE

3. A Basic Counter Unit (BCU)
4. Level Usage Counters

Of these, SLIN and BCU were used to gather some evidence of the improvements; however, the Level Usage Counters and conversational SIPE were the most important tools used for tuning, the primary one being the Level Usage Counters.

SLIN

The Carnegie Mellon Simulator is a program developed at Carnegie Mellon University that can coexist in the Model 67 memory with TSS/360. It simulates multiple terminals and interfaces with TSS/360 at the CCW level of the transmission control unit. Each simulated terminal can use a different set of commands (called a script) or all can use the same script. The overhead is quite low both in core space and CPU time.

Conversational SIPE

SIPE is a selective event trace capable of tracing many combinations of system functions simultaneously.[10] Depending on the events traced, overhead ranges from about 30 percent to less than 1 percent. Conversational SIPE traces all CCWs and their data at the transmission control unit. Its overhead is about 1 percent. It was used only for user session measurements.

The BCU

The BCU is a set of 16 hardware counters capable of measuring summary information about either time duration or frequency of use of the various hardware components of any computer. It counts at a one microsecond rate and was used occasionally to measure loads on the Model 67 CPU and channels during user sessions as well as runs with SLIN.

The Level Usage Counters

The Level Usage Counters are a set of software counters, one for each level of the schedule table, that are incremented by one each time a task is dispatched at the level. They were used during the user sessions as well as during the SLIN runs. They provide information about utilization of the various schedule table levels and sets of levels.

Use of the tools

The initial experiments with the TSS/360 schedule table were run using SLIN, the BCU, and the Level Usage Counters for instrumentation. Although we had two million bytes of LCS on our Model 67, this was rarely included when experimenting so results could be made as relevant to other installations as possible. The SHARE script (Figure 1) was used, initially running on 20 simulated terminals and, later, running on 36 simulated terminals. The script (running with a single user) took approximately 2400 seconds. Thus to minimize the probability of several terminals simultaneously executing the same lines of the script, the logon of all terminals was spread evenly over a 2400-second interval. The first (n-1) terminals repeatedly executed the script. The last terminal to logon executed the script only once. Measurements for all terminals were made from the time when the last terminal logged on until it logged off. Thus all simulated terminals were active during the measurement period, and at least one complete execution of the entire script was assured during that period.

It is important to note that almost any scheduling technique will show similar results under light loads. It is only when system resources begin getting scarce that scheduling differences show up clearly.

All runs after the initial one were made with 36 terminals because earlier measurements had been made at that number and could be used for comparison. It was also a number at which scheduling changes made noticeable differences in results.

Each SLIN run took close to two-and-a-half hours of stand alone machine time when all setup operations are considered. Furthermore, the SHARE script did not even come close to the characteristics of our live user load as measured by the BCU and Level Usage Counters. Therefore we installed the new schedule tables in the user sessions, and made measurements there. Although this was not a controllable load it was measurable with the BCU, conversational SIPE and the Level Usage Counters. During times when the system slowed, and at the end of every day, readings of the Level Usage Counters were taken. The cause of the slowdown was almost always traceable to unexpected use of large working set size levels. It was then possible to create a new table within a day that eliminated the previously detected slowdown. We found that it is unprofitable to regard every program as being unalterable, then attempting to fit the scheduling table to this unmanageable program load. It is instead more profitable to allow the poorly-behaving programs to suffer, in the hope that their creators would rewrite in better fashion or discard them altogether.

RESULTS

These following results are valid for our particular hardware configuration using Release 4 of TSS/360;

```
51  DDEF LIB,VP,DSNAME=LIB1,OPTION=JOBLIB
31  DATA SOURCE.TSSTWO,I
46  1 READ(1,2) A,B,C
70  2 FORMAT(8X,F10.6,1X,F10.6,1X,F10.6)
82  400 IF((B**2).EQ.(4.0*A*C)) GO TO 50
47  20 DISC=B**2-4.0*A*C
34  21 IF(DISC) 40,50,60
48  40 X1R=-B/(2.0*A)
30  300 X2R=X1R
54  X1I=SQRT(-DISC)/(2.0*A)
31  310 X2I=X1I
20  48 GO TO 70
44  50 X1R=-B/(2.0*A)
27  320 X2R=X1R
20  330 GO TO 370
32  60 S+SQRT(DISC)
51  350 X1R=(-B+S)/(2.0*A)
53  360 X2R=(-B-S)/(2.0*A)
27  370 X1I=0.0
22  380 X2I=0.0
45  70 IF(X1I) 90,91,90
51  91 WRITE(2,95) A,B,C,X1R,X2R
68  95 FORMAT(1H0,1P4E15.4,E30.4)
19  150 GO TO 100
34  90 IF(X1R) 80,81,80
35  81 IF(X2R) 80,82,80
50  82 WRITE(2,83) A,B,C,X1I,X2I
65  83 FORMAT(1H0,1P3E15.4,2E30.4)
18  151 GO TO 100
67  80 WRITE(2,84) A,B,C,X1R,X1I,X2R,X2I
47  84 FORMAT(1H0,1P7E15.4)
16  100 STOP
15  END
14  %E
21  MODIFY SOURCE.TSSTWO
46  700,   200 X2R=X1R
16  R,1200
16  D,1200
32  1150,  210 X2R=X1R
15  %E
35  FTN TSSTWO,Y,ISD=Y
26  1000, 58 GO TO 70
12
35  1400,    60 S=SQRT(DISC)
12
16  LOAD TSSTWO
16  QUALIFY TSSTWO
20  AT 1;BRANCH 400
51  SET A=4.0,B=4.0,C=1.0
69  AT 91;DISPLAY X1R,X2R,X1I,X2I;STOP
16  RUN TSSTWO
16  REMOVE 2
99  SET A=5.0,B=7.0,C=2.0
53  AT 70;DISPLAY X1R,X2R,X1I,X2I;STOP
16  RUN TSSTWO
17  REMOVE 1,3
16  RUN TSSTWO
31            3.0         5.0         3.0
42  LINE? SOURCE.TSSTWO,(700,1600)
41  PERMIT SOURCE.TSSTWO,N,RO,*ALL
21  DSS? SOURCE.TSSTWO
18  ERASE LIB1
```

Figure 1—The SHARE conversational script with think times

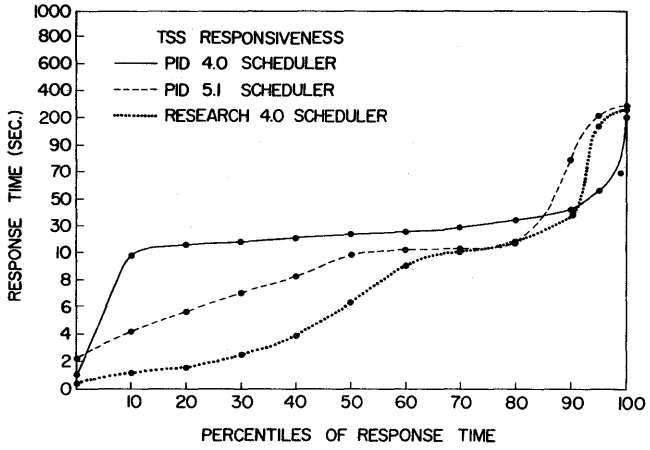


Figure 2—Results of SLIN runs with 36 terminals comparing the release 4 research table T47 with the TSS/360 release 4 table and the best TSS/360 release 5.01 table

they do not necessarily apply to any other TSS/360 release or hardware configuration. However, the principles involved apply to all paging systems.

Evidence of the results come in five forms:

1. Controlled experiments on the SHARE conversational benchmarks.
2. Improvements during user sessions on measured but unrepeatable loads.
3. Improvements in behavior of exceptionally indicative programs.
4. Measures of acutal working set sizes.
5. BCU measurements of CPU usage.

Controlled experiments

The results of using SLIN to run the SHARE conversational script on the T. J. Watson Research Center Model 67 are presented in Figure 2. These results compare responsiveness from the original TSS/360 Release 4 schedule table (Figure 3) with the best one for Release 5.01 (Figure 4) and the Research table T47 (Figure 5). The major effect of the new scheduling tables was to improve the responsiveness across the entire script by significant factors. A significant further improvement seems possible in this area and will be discussed in the section SUGGESTED EXTENSIONS.

Improvements during user sessions

I regard the results to be described here as the most significant evidence of substantial gains in TSS/360 performance via application of the balanced core time

SCHEDULE TABLE ENTRIES (CHASTE)	STE LEVEL	STE PRIOR	STE TSVAL	STE QUANT	STE MAXCR	STE MAXRD	STE PULSE	STE AWTEX	STE DELTA	STE TSEND	STE MPRE	STE AWAIT	STE TWAIT	STE RCMP=80 PRMPT=40	STE MRQ	STE HLCK	STE LCHL	STE WLCK	STE PRJ1	STE PRJ2	STE PRJ3	STE PRJ4	
SYSOPERO	00	05	004C	04	32	FF	00	0000	05	00	00	00	00	00	6A	16	16	2C	00	00	00	00	
CONVERSATIONAL	1-4	5	4C	4	32	FF	1-4	0	5	5D-60	5D-60	1-4	1-4	80	A	17-1A	17-1A	2D-30	39-3C	42-45	4B-4E	54-57	
CONVERSATIONAL	5	5	4C	4	28	32	5	0	1	61	61	5	5	80	A	1B	1B	31	3D	46	4F	58	
CONVERSATIONAL	6-9	5	4C	4	32	FF	6-9	0	5	62-65	62-65	6-9	6-9	80	A	1C-1F	1C-1F	32-35	3E-41	47-4A	50-53	59-5C	
BULKIO	A	5	4C	4	32	FF	A	0	0	A	A	A	A	00	1	20	20	36	A	A	A	A	INITIAL LEVELS
BATCH (MIX)	B-12	5	4C	4	32	FF	B-12	0	F	B-12	B-12	B-12	B-12	C0	0	21-28	21-28	B-12	B-12	B-12	B-12	B-12	
BATCH (ONLY)	13	A	4C	78	32	FF	13	0	0	13	13	13	13	C0	0	29	29	13	13	13	13	13	
LOGON	14	5	4C	4	32	FF	14	0	5	14	14	14	14	00	A	2A	2A	37	14	14	14	14	
LOGOFF	15	5	4C	4	32	FF	15	0	5	15	15	15	15	00	A	2B	2B	38	15	15	15	15	
SYSOPERO	16	0	012F	01	32	FF	16	0	0	0	0	0	0	80	1	0	16	0	0	0	0	0	
CONVERSATIONAL	17-1F	0	12F	1	32	FF	17-1F	0	0	1-9	1-9	1-9	1-9	80	1	1-9	17-1F	1-9	39-41	42-4A	4B-53	54-5C	HOLDING
BULKIO	20	0	12F	1	32	FF	20	0	0	A	A	A	A	80	1	A	20	A	A	A	A	A	INTERLOCK
BATCH (MIX)	21-28	0	12F	1	32	FF	21-28	0	0	B-12	B-12	B-12	B-12	80	1	B-12	21-28	B-12	B-12	B-12	B-12	B-12	LEVELS
BATCH (ONLY)	29	0	12F	1	32	FF	29	0	0	13	13	13	13	80	1	13	29	13	13	13	13	13	
LOGON & LOGOFF	2A-2B	0	12F	1	32	FF	2A-2B	0	0	14-15	14-15	14-15	14-15	80	1	14-15	2A-2B	14-15	14-15	14-15	14-15	14-15	
SYSOPERO	2C	5	4C	4	32	FF	2C	0	0	0	0	0	0	80	1	16	16	2C	0	0	0	0	WAITING
CONVERSATIONAL	2D-35	5	4C	4	32	FF	2D-35	0	0	1-9	1-9	1-9	1-9	80	1	17-1F	17-1F	2D-35	39-41	42-4A	4B-53	54-5C	FOR
BULKIO	36	5	4C	4	32	FF	36	0	0	A	A	A	A	80	1	20	20	36	A	A	A	A	INTERLOCK
LOGON & LOGOFF	37-38	5	4C	4	32	FF	37-38	0	0	14-15	14-15	14-15	14-15	80	1	21-22	21-22	37-38	14-15	14-15	14-15	14-15	LEVELS
PREJUDICE1	39-41	5	4C	4	32	FF	39-41	0	5	1-9	1-9	1-9	1-9	80	1	1-9	1-9	1-9	1-9	1-9	1-9	1-9	
PREJUDICE2	42-4A	5	4C	4	32	FF	42-4A	0	A	1-9	1-9	1-9	1-9	80	1	1-9	1-9	1-9	1-9	1-9	1-9	1-9	PREJUDICE
PREJUDICE3	4B-53	5	4C	4	32	FF	4B-53	0	14	1-9	1-9	1-9	1-9	80	1	1-9	1-9	1-9	1-9	1-9	1-9	1-9	LEVELS
PREJUDICE4	54-5C	5	4C	4	32	FF	54-5C	0	1E	1-9	1-9	1-9	1-9	80	1	1-9	1-9	1-9	1-9	1-9	1-9	1-9	
CONVERSATIONAL TSEND & MAXCR	5D-65	5	4C	4	32	FF	5D-65	0	1E	5D-65	5D-65	1-9	1-9	80	A	17-1F	17-1F	2D-35	39-41	42-4A	4B-53	54-5C	CONVERSATIONAL
TSEND & MAXCR	61	6	4C	4	40	FF	61	0	1E	61	61	5	5	80	A	1B	1B	31	3D	46	4F	58	CONVERSATIONAL

Figure 3—Schedule table T4—the TSS/360 release 4 table

principle and the concept of working set size. Since quantitative comparative measures were not available, the following base can be considered for comparison:

1. The load generated on our system by experienced users who had built up large files of sophisticated procedures, programs, and data was heavier than the one they generated in the first year of TSS/360 usage.
2. When running without LCS, response time for trivial commands had regularly been above 30 seconds when the load reached 15 simultaneous users. This was intolerable to users. The only measures I have of this are over a year old, and they show trivial command response nearer 10 seconds for the 15 to 20 terminal range.

Now for the results. Periodically, beginning in January, 1970, we had run our user session for entire days without LCS but with one of the new tables (Figure 5). On the days before and after these, our simultaneous user load had been between 20 and 30 users for most of the day. This did not change when we ran without LCS with the new schedule table. At the time of writing this paper there is a mass of partially reduced SIPE data which contains a complete record of what all users were doing on those days. The following will attempt to describe the behavior of the system on the day of the heaviest user load, January 14, 1970. The morning is always lighter than the afternoon; so, the afternoon will be described:

1. There were over 40 people who used the system that afternoon.
2. The system came up at 12:42 in the afternoon. By 2:05 we reached a level of 20 simultaneous users.
3. We reached a peak of 29 simultaneous users and then hovered around 25 until 5:00 p.m.
4. The users were not told that we were doing anything unusual.
5. One user complained vociferously. He had a paper he was producing with the RUNOFF command at the terminal. He also knew exactly how fast RUNOFF could go under good conditions. He had a deadline to reach and only a few minutes extra time. He used TSS/360 from 1:10 until 3:23 p.m. In this period he executed 184 commands, primarily editing from a 2741. The response to the 175 editing commands was less than two seconds for all but three commands. The unusual three were:

5.4 seconds	one line of a 10-line PRINT command
3.5 seconds	
33.0 seconds	To execute a FILE command

He began the RUNOFF at 2:03 p.m. This was just when the simultaneous user load was

		LEVEL	PRIOR	TSVAL	QUANT	MAXCRD	MAXRDF	PULSE	AWTEX	DELTA	TSEND	MPRE	TWAIT	AWAIT	RPCRMM PPT	MRQ	HLCK	LCHK	WLCK	PRJ1	PRJ2	PRJ3	PRJ4
	SYSOPERO	00	04	004C	04	32	32	00	0000	01	00	00	00	00	00	0A	16	16	27	00	00	00	00
STARTING	INTERACTIVE	01	04	0026	04	32	32	01	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	02	04	0026	04	32	32	02	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
LEVELS	INTERACTIVE	03	04	0026	04	32	32	03	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	04	04	0026	04	32	32	04	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	05	04	0026	04	32	32	05	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	06	04	0026	04	32	32	06	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	07	04	0026	04	32	32	07	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	08	04	0026	04	32	32	08	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	INTERACTIVE	09	04	0026	04	32	32	09	0000	02	34	34	08	08	80	05	17	17	34	1F	1F	1F	1F
	BULK I-O	0A	04	004C	04	46	46	0A	0000	00	0A	0A	0A	0A	00	01	1A	1A	26	0A	0A	0A	0A
	BATCH	0B	1A	004C	04	46	46	0B	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	0C	1A	004C	04	46	46	0C	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	0D	1A	004C	04	46	46	0D	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	0E	1A	004C	04	46	46	0E	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	0F	1A	004C	04	46	46	0F	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	10	1A	004C	04	46	46	10	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	11	1A	004C	04	46	46	11	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	12	1A	004C	04	46	46	12	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	13	1A	004C	78	73	73	13	0000	00	13	13	13	13	C0	00	1C	1C	13	13	13	13	13
	LOGON	14	02	004C	04	32	32	14	0000	01	14	14	14	14	00	0A	1D	1D	25	14	14	14	14
	LOGOFF	15	04	004C	04	28	28	15	0000	01	15	15	15	15	80	0A	1E	1E	24	15	15	15	15
	SYSOPERO	16	00	004C	04	32	32	16	0000	00	00	00	00	00	00	0A	16	16	27	00	00	00	00
HOLDING	INTERACTIVE	17	05	004C	04	32	32	17	0000	01	34	34	08	08	80	0A	17	17	34	1F	1F	1F	1F
	INTERACTIVE	18	05	004C	04	32	32	18	0000	01	49	49	08	08	80	0A	18	18	49	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	19	05	004C	04	32	32	19	0000	01	49	49	08	08	80	0A	19	19	49	1F	1F	1F	1F
	BULK I-O	1A	05	004C	04	46	46	1A	0000	00	0A	0A	0A	0A	00	01	1A	1A	26	0A	0A	0A	0A
LEVELS	BATCH	1B	05	004C	04	46	46	1B	0000	01	59	59	59	59	80	00	1B	1B	59	59	59	59	59
	BATCH	1C	05	004C	04	73	73	1C	0000	01	13	13	13	13	80	01	1C	1C	13	13	13	13	13
	LOGON	1D	00	004C	04	32	32	1D	0000	00	14	14	14	14	00	0A	1D	1D	25	14	14	14	14
	LOGOFF	1E	05	004C	04	28	28	1E	0000	00	15	15	15	15	80	0A	1E	1E	24	15	15	15	15
PREJUDICE	WRITE	1F	00	004C	04	23	23	1F	0000	00	34	34	08	08	80	0A	17	17	34	1F	1F	1F	1F
	LOGOFF	24	06	004C	04	32	32	24	0000	03	15	15	15	15	80	0A	1E	1E	24	15	15	15	15
WAITING	LOGON	25	06	004C	04	32	32	25	0000	03	14	14	14	14	00	0A	1D	1D	25	14	14	14	14
	BULK I-O	26	06	004C	04	32	32	26	0000	03	0A	0A	0A	0A	00	01	1A	1A	26	0A	0A	0A	0A
FOR	SYSOPERO	27	06	004C	04	32	32	27	0000	03	00	00	00	00	00	0A	16	16	27	00	00	00	00
	INTERACTIVE	28	06	004C	04	32	32	28	0000	03	49	49	08	08	80	0A	18	18	28	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	29	06	004C	04	32	32	29	0000	03	30	30	08	08	80	0A	30	30	29	1F	1F	1F	1F
	INTERACTIVE	2A	06	004C	04	32	32	2A	0000	03	31	31	08	08	80	0A	31	31	2A	1F	1F	1F	1F
LEVELS	INTERACTIVE	2B	06	004C	04	32	32	2B	0000	03	32	32	08	08	80	0A	32	32	2B	1F	1F	1F	1F
	INTERACTIVE	2C	07	004C	04	32	32	2C	0000	03	48	48	08	08	80	0A	19	19	2C	1F	1F	1F	1F
	INTERACTIVE	30	05	004C	04	32	32	30	0000	00	49	49	08	08	80	0A	18	18	49	1F	1F	1F	1F
SECOND	INTERACTIVE	31	03	004C	04	32	32	31	0000	00	30	30	08	08	80	0A	30	30	28	1F	1F	1F	1F
	INTERACTIVE	32	05	004C	04	32	32	32	0000	00	31	31	08	08	80	0A	31	31	29	1F	1F	1F	1F
LEVEL	INTERACTIVE	33	03	004C	04	32	32	33	0000	00	32	32	08	08	80	0A	32	32	2A	1F	1F	1F	1F
	INTERACTIVE	34	08	004C	04	32	32	34	0000	08	33	33	08	08	80	0A	33	33	2B	1F	1F	1F	1F
	INTERACTIVE	40	07	004C	04	46	46	40	0000	00	49	49	08	08	80	0A	19	19	49	1F	1F	1F	1F
THIRD	INTERACTIVE	41	07	004C	04	46	46	41	0000	00	40	40	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	42	07	004C	04	46	46	42	0000	00	41	41	08	08	80	0A	19	19	49	1F	1F	1F	1F
LEVEL	INTERACTIVE	43	07	004C	04	46	46	43	0000	00	42	42	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	44	07	004C	04	46	46	44	0000	00	43	43	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	45	07	004C	04	46	46	45	0000	00	44	44	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	46	07	004C	04	46	46	46	0000	00	45	45	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	47	07	004C	04	46	46	47	0000	00	46	46	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	48	07	004C	04	46	46	48	0000	00	47	47	08	08	80	0A	19	19	49	1F	1F	1F	1F
	INTERACTIVE	49	09	004C	04	46	46	49	0000	01	48	48	08	08	80	0A	19	19	2C	1F	1F	1F	1F
THIRD LEVEL	BATCH	50	10	004C	04	46	46	50	0000	00	50	50	50	50	80	00	5A	5A	50	50	50	50	50
	BATCH	51	11	004C	04	46	46	51	0000	00	51	50	51	51	80	00	1B	1B	51	51	51	51	51
SECOND	BATCH	52	12	004C	04	46	46	52	0000	00	52	51	52	52	80	00	1B	1B	52	52	52	52	52
	BATCH	53	13	004C	04	46	46	53	0000	00	53	52	53	53	80	00	1B	1B	53	53	53	53	53
LEVEL	BATCH	54	14	004C	04	46	46	54	0000	00	54	53	54	54	80	00	1B	1B	54	54	54	54	54
	BATCH	55	15	004C	04	46	46	55	0000	00	55	54	55	55	80	00	1B	1B	55	55	55	55	55
	BATCH	56	16	004C	04	46	46	56	0000	00	56	55	56	56	80	00	1B	1B	56	56	56	56	56
	BATCH	57	17	004C	04	46	46	57	0000	00	57	56	57	57	80	00	1B	1B	57	57	57	57	57
	BATCH	58	18	004C	04	46	46	58	0000	00	58	57	58	58	80	00	1B	1B	58	58	58	58	58
	BATCH	59	19	004C	04	46	46	59	0000	00	59	58	59	59	80	00	1B	1B	59	59	59	59	59
HOLDING INTERLOCK	BATCH	5A	05	004C	04	46	46	5A	0000	00	50	50	50	50	80	00	5A	5A	50	50	50	50	50

Figure 4—Schedule table T5—the best TSS/360 release 5.01 table

		LEVEL	PRIOR	TSVAL	QUANTM	HAXCR	MAXRD	PULSE	AWTEX	DELTA	TSEND	MPRE	TWAIT	AWAIT	RPCRMMPP	MRQ	HLC	LCHK	WLCK	PRJ1	PRJ2	PRJ3	PRJ4
	SYSOPER0	00	05	0026	01	20	FF	00	0000	00	00	00	00	00	00	0A	16	16	23	00	00	00	00
STARTING	INTERACTIVE	01	18	0010	01	10	FF	01	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	02	18	0010	01	10	FF	02	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
SET	INTERACTIVE	03	18	0010	01	10	FF	03	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	04	18	0010	01	10	FF	04	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	05	18	0010	01	10	FF	05	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	06	18	0010	01	10	FF	06	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	07	18	0010	01	10	FF	07	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	08	18	0010	01	10	FF	08	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	09	18	0010	01	10	FF	09	0000	00	47	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	BULK I-O	0A	04	0026	01	20	FF	0A	0000	00	0A	0A	0A	0A	00	01	1A	1A	22	0A	0A	0A	0A
	BATCH	0B	1D	0013	01	10	FF	0B	012F	01	4C	35	35	0B	80	0A	1B	1B	27	0B	0B	0B	0B
	BATCH	0C	1D	0013	01	10	FF	0C	012F	01	4C	35	35	0C	80	0A	1B	1B	27	0C	0C	0C	0C
	BATCH	0D	1D	0013	01	10	FF	0D	012F	01	4C	35	35	0D	80	0A	1B	1B	27	0D	0D	0D	0D
	BATCH	0E	1D	0013	01	10	FF	0E	012F	01	4C	35	35	0E	80	0A	1B	1B	27	0E	0E	0E	0E
	BATCH	0F	1D	0013	01	10	FF	0F	012F	01	4C	35	35	0F	80	0A	1B	1B	27	0F	0F	0F	0F
	BATCH	10	1D	0013	01	10	FF	10	012F	01	4C	35	35	10	80	0A	1B	1B	27	10	10	10	10
	BATCH	11	1D	0013	01	10	FF	11	012F	01	4C	35	35	11	80	0A	1B	1B	27	11	11	11	11
	BATCH	12	1D	0013	01	10	FF	12	012F	01	4C	35	35	12	80	0A	1B	1B	27	12	12	12	12
	BATCH	13	1D	0013	01	10	FF	13	012F	00	4C	35	35	13	80	0A	1B	1B	27	13	13	13	13
	LOGON	14	03	0026	02	20	FF	14	0000	00	14	14	14	14	00	0A	1D	1D	21	14	14	14	14
	LOGOFF	15	04	0026	02	20	FF	15	0000	00	15	15	15	15	80	0A	1E	1E	20	15	15	15	15
	SYSOPER0	16	00	0013	01	20	FF	16	012F	00	00	00	16	00	00	0A	16	16	23	00	00	00	00
HOLDING	INTERACTIVE	17	02	0013	01	10	FF	17	012F	00	47	2B	17	08	00	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	18	02	0013	01	20	FF	18	012F	00	4B	2B	18	08	00	0A	18	18	25	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	19	03	0013	01	30	FF	19	012F	00	4A	2C	19	08	00	0A	19	19	26	1F	1F	1F	1F
	BULK I-O	1A	03	0013	01	20	FF	1A	012F	00	0A	0A	1A	0A	00	0A	1A	1A	22	1A	1A	1A	1A
SET	BATCH	1B	04	0013	01	10	FF	1B	012F	00	4C	35	1B	0B	00	0A	1B	1B	27	1B	1B	1B	1B
	BATCH	1C	04	0013	01	30	FF	1C	012F	00	35	36	1C	13	00	0A	1C	1C	28	1C	1C	1C	1C
	LOGON	1D	01	0013	01	20	FF	1D	012F	00	14	14	1D	14	00	0A	1D	1D	21	14	14	14	14
	LOGOFF	1E	03	0013	01	20	FF	1E	012F	00	15	15	1E	15	00	0A	1E	1E	20	15	15	15	15
PREJUDICE	WRITE	1F	01	0013	01	23	FF	1F	0000	00	1F	1F	08	08	00	0A	17	17	24	1F	1F	1F	1F
	LOGOFF	20	06	0013	01	20	FF	20	0030	23	15	15	15	15	80	0A	1E	1E	20	15	15	15	15
WAITING	LOGON	21	06	0013	01	20	FF	21	0000	23	14	14	14	14	80	0A	1D	1D	21	14	14	14	14
	BULK I-O	22	06	0013	01	20	FF	22	0000	23	0A	0A	0A	0A	80	0A	1A	1A	22	0A	0A	0A	0A
FOR	SYSOPER0	23	06	0013	01	20	FF	23	0000	07	00	00	00	00	80	0A	16	16	23	00	00	00	00
	INTERACTIVE	24	06	0013	01	10	FF	24	0000	23	47	2B	3D	08	00	0A	17	17	24	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	25	06	0013	01	20	FF	25	0000	23	47	2B	3D	08	80	0A	18	18	25	1F	1F	1F	1F
	INTERACTIVE	26	06	0013	01	30	FF	26	0000	23	2B	2D	3F	08	80	0A	19	19	26	1F	1F	1F	1F
SET	BATCH	27	06	0013	01	10	FF	27	012F	23	33	35	27	0B	80	0A	1B	1B	27	0B	0B	0B	0B
	BATCH	28	06	0013	01	30	FF	28	012F	23	33	35	28	13	80	0A	1C	1C	28	13	13	13	13
	GROWING	29	08	0026	08	10	FF	29	0000	00	2A	2B	3D	08	00	0A	17	17	24	1F	1F	1F	1F
LOOPING	DELAYING	2A	08	0026	08	18	FF	2A	0000	23	47	29	3D	08	80	0A	18	18	25	1F	1F	1F	1F
	GROWING	2B	18	0026	02	20	FF	2B	0000	00	2A	2C	3F	08	00	0A	18	18	25	1F	1F	1F	1F
INTERACTIVE	GROWING	2C	17	0026	02	30	FF	2C	0000	00	2F	2D	40	08	00	0A	19	19	26	1F	1F	1F	1F
	GROWING	2D	16	0013	01	40	FF	2D	0000	00	30	2E	41	08	00	0A	19	19	26	1F	1F	1F	1F
SET	GROWING	2E	15	0013	01	40	FF	2E	0000	00	31	32	42	08	00	0A	19	19	26	1F	1F	1F	1F
	DELAYING	2F	14	0026	02	20	FF	2F	0000	23	4B	2B	3F	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	30	14	0026	01	30	FF	30	0000	23	4A	2C	40	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	31	14	0026	01	40	FF	31	0000	23	49	2D	41	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	32	14	0013	01	40	FF	32	0000	23	49	2D	42	08	80	0A	19	19	26	1F	1F	1F	1F
	GROWING	33	13	0026	08	10	FF	33	012F	00	34	35	34	34	00	0A	1B	1B	27	34	34	34	34
LOOPING	DELAYING	34	11	0026	08	18	FF	34	012F	23	4C	33	33	34	80	0A	1B	1B	27	34	34	34	34
	GROWING	35	1C	0026	02	20	FF	35	012F	00	33	36	39	39	00	0A	1B	1B	27	39	39	39	39
BATCH	GROWING	36	1B	0026	02	30	FF	36	012F	00	39	37	3A	3A	00	0A	1C	1C	28	3A	3A	3A	3A
	GROWING	37	1A	0013	01	40	FF	37	012F	00	3A	38	3B	3B	00	0A	1C	1C	28	3B	3B	3B	3B
SET	GROWING	38	19	0013	01	40	FF	38	012F	00	3B	3C	3C	3C	00	0A	1C	1C	28	3C	3C	3C	3C
	DELAYING	39	1A	0026	02	20	FF	39	012F	23	50	35	35	39	80	0A	1C	1C	28	39	39	39	39
	DELAYING	3A	1A	0026	01	30	FF	3A	812F	23	4F	36	36	3A	00	0A	1C	1C	28	3A	3A	3A	3A
	DELAYING	3B	1A	0013	01	40	FF	3B	012F	23	4E	37	37	3B	80	0A	1C	1C	28	3B	3B	3B	3B
	DELAYING	3C	1A	0013	01	40	FF	3C	012F	23	4E	37	38	3C	80	0A	1C	1C	28	3C	3C	3C	3C
	GROWING	3D	08	0026	01	10	FF	3D	012F	00	2A	2B	3E	08	00	0A	17	17	24	1F	1F	1F	1F
AWAIT	DELAYING	3E	0B	0026	01	10	FF	3E	012F	23	47	2B	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	GROWING	3F	18	0013	01	20	FF	3F	012F	00	2A	2C	43	08	00	0A	18	18	25	1F	1F	1F	1F
INTERACTIVE	GROWING	40	17	0010	01	30	FF	40	012F	00	2F	2D	44	08	00	0A	19	19	26	1F	1F	1F	1F
	GROWING	41	16	0004	01	40	FF	41	012F	00	30	2E	45	08	00	0A	19	19	26	1F	1F	1F	1F
SET	GROWING	42	15	0004	01	40	FF	42	012F	00	31	32	46	08	00	0A	19	19	26	1F	1F	1F	1F
	DELAYING	43	14	0013	01	20	FF	43	012F	23	29	2C	3F	08	80	0A	18	18	25	1F	1F	1F	1F
	DELAYING	44	14	0010	01	30	FF	44	012F	23	2B	2D	40	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	45	14	0004	01	40	FF	45	012F	23	2C	2E	41	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	46	14	0004	01	40	FF	46	012F	23	2D	2E	42	08	80	0A	19	19	26	1F	1F	1F	1F
	SHRINKING	47	07	0026	10	08	FF	47	0000	00	48	29	3D	08	00	08	17	17	24	1F	1F	1F	1F
LOOPING	DELAYING	48	07	0026	10	08	FF	48	0000	23	47	29	3D	08	80	08	17	17	24	1F	1F	1F	1F
INTERACTIVE	SHRINKING	49	17	0026	01	30	FF	49	0000	00	4A	2C	40	08	00	0A	19	19	26	1F	1F	1F	1F
SET	SHRINKING	4A	16	0026	02	20	FF	4A	0000	00	4B	2B	3F	08	00	0A	18	18	25	1F	1F	1F	1F
	SHRINKING	4B	15	0026	08	18	FF	4B	0000	00	47	29	3E	08	00	0A	18	18	25	1F	1F	1F	1F
	SHRINKING	4C	13	0026	10	08	FF	4C	012F	00	4D	33	4D	4D	00	08	1B	1B	27	4D	4D	4D	4D
LOOPING	DELAYING	4D	11	0026	10	08	FF	4D	012F	23	4C	33	4C	4D	80	08	1B	1B	27	4D	4D	4D	4D
BATCH	SHRINKING	4E	1B	0026	01	30	FF	4E	012F	00	4F	37	3A	3A	00	0A	1C	1C	28	4E	4E	4E	4E
SET	SHRINKING	4F	1A	0026	02	20	FF	4F	012F	00	50	36	39	39	00	0A	1C	1B	27	4F	4F	4F	4F
	SHRINKING	50	19	0026	08	18	FF	50	012F	00	4C	35	34	34	00	0A	1B	1B	27	50	50	50	50
STARTING	INTERACTIVE	51	17	0008	01	20	FF	51	0000	00	2B	52	3F	08	00	0A	18	18	25	1F	1F	1F	1F
SET	INTERACTIVE	52	16	0006	01	30	FF	52	0000	00	2C	53	40	08	00	0A	19	19	26	1F	1F	1F	1F
	INTERACTIVE	53	15	0004	01	40	FF	53	0000	00	2C	2D	40	08	00	0A	19	19	26	1F	1F	1F	1F

Figure 5—Schedule table T47—The table in use when we first ran without LCS

crossing the 20 mark and building rapidly. The average delay during the RUNOFF between 522 lines of type-out was 3.4 seconds. This included four unusually high delays of 71.6, 85.4, 87 and 117.1 seconds. It turns out that these unusual delays occurred because of a high level of simultaneous load. I was the chief perpetrator of this load and was thus hoisted on my own petard. I was doing a virtual memory sort over 2.5 million bytes of data during the interval of time when the four large delays occurred. This table still had its weak moments.

This, however, was the only cloud on a perfect afternoon. No other user complained.

6. One user was on with an 1130–2250 doing algebraic symbol manipulation with LISP.
7. Three users were on all afternoon using a medical application program.
8. Two users were on editing using the REDIT command with a 2741 for input and a 2260 scope for output. One of these two was the high success story of the day. He was on from 1:08 until 2:55 p.m. During this time he executed 622 editing commands. This was an average interaction time of 10 seconds. This includes type-in time, processing time, think time and display time. And he is not a touch typist! He averaged 5.1 seconds to respond to TSS/360 with each new command. TSS/360, in turn, averaged 4.3 seconds to process his request, change the display and then open his 2741 keyboard for the next request. This includes a single high peak of 104 seconds during my 2.5 million byte sort, similar to the RUNOFF delays.
9. The remainder of the users were in various stages of editing, compiling, executing, debugging, abending, and other normal operations. Most of the users were on the system for one or more hours of consecutive use that day.
10. Ninety per cent of the compilations completed in less than one minute that afternoon.
11. To compound the issue, we were running with what we considered to be a critically low amount of permanent 2314 disk space available on that day. There were between 4,000 and 5,500 pages of public storage available out of a possible 48,000 pages of on-line public storage. Thus I assume higher than normal delays due to disk seeking existed during this day.
12. The most quantitative evidence I can offer from the contribution of the balanced core time and working set size concepts was obtained from the Level Usage Counters.
    a. Of all task dispatches, 89 per cent required less than 32 pages.
    b. Ten per cent required between 32 and 48 pages. This could be even lower if the assumption is made that this simply reflects a working set change and not a working set size change.
    c. The remaining one per cent of all dispatches were for up to 64 pages.
    d. Breaking this down by sets of table levels, there were:

Starting set	28 percent
Looping set	30 percent
Batch	4 percent
Holding interlocks	5 percent
Waiting for interlocks	2 percent
AWAIT	5 percent
BULKIO	10 percent
SYSTEM OPERATOR	5 percent
LOGON	6 percent
LOGOFF	5 percent

13. Since the BCU has not been available since December 20, 1969, there were no BCU measurements made that day.
14. The table in use on that day was T47 (Figure 5), which is very similar to T48 (Figure 6). T48 was created to reduce the impact on other users of people doing in core sorts of several million bytes on a computer which only allocates them several thousand.

Changes in indicative programs

The programs discussed here exhibit to a marked degree improvements which are present to a lesser degree in all programs. They are:

1. A tutoring program, using a 2260 display with a 2741 for some input formerly had the following property:

   If you asked it a question whose answer was unknown it displayed a message to that effect. Then after a few seconds the screen would change to let you choose what to do next.

   Once the first new table was put in use, the first message was usually not seen by most people. This was because the old normal system delay no longer existed. The program's author had to put in a PAUSE after the first message to allow it to be read.
2. When the first new table was put into daily use, only one user was slowed down. He was using

		LEVEL	PRIOR	TSWAL	QUANT	MAXCRD	MAXDSE	PULSE	AWT EXT	DELTA	TSEND	MPRE	TWAIT	AWAIT	RPCRMMPP	MRRQ	HLCK	LLCH	WLCK	PRJ 1	PRJ 2	PRJ 3	PRJ 4
	SYSOPERO	00	05	0026	01	20	FF	00	0000	00	00	00	00	00	00	0A	16	16	23	00	00	00	00
STARTING	INTERACTIVE	01	18	0020	01	10	FF	01	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	02	18	0020	01	10	FF	02	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
SET	INTERACTIVE	03	18	0020	01	10	FF	03	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	04	18	0020	01	10	FF	04	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	05	18	0020	01	10	FF	05	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	06	18	0020	01	10	FF	06	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	07	18	0020	01	10	FF	07	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	08	18	0020	01	10	FF	08	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	09	18	0020	01	10	FF	09	0000	00	2A	51	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	BULK I-O	0A	04	0026	01	20	FF	0A	0030	00	0A	0A	0A	0A	00	01	1A	1A	22	0A	0A	0A	0A
	BATCH	0B	1D	0013	01	10	FF	0B	012F	FE	4C	35	35	0B	80	0A	1B	1B	27	0B	0B	0B	0B
	BATCH	0C	1D	0013	01	10	FF	0C	012F	FE	4C	35	35	0C	80	0A	1B	1B	27	0C	0C	0C	0C
	BATCH	0D	1D	0013	01	10	FF	0D	012F	FE	4C	35	35	0D	80	0A	1B	1B	27	0D	0D	0D	0D
	BATCH	0E	1D	0013	01	10	FF	0E	012F	FE	4C	35	35	0E	80	0A	1B	1B	27	0E	0E	0E	0E
	BATCH	0F	1D	0013	01	10	FF	0F	012F	FE	4C	35	35	0F	80	0A	1B	1B	27	0F	0F	0F	0F
	BATCH	10	1D	0013	01	10	FF	10	012F	FE	4C	35	35	10	80	0A	1B	1B	27	10	10	10	10
	BATCH	11	1D	0013	01	10	FF	11	012F	FE	4C	35	35	11	80	0A	1B	1B	27	11	11	11	11
	BATCH	12	1D	0013	01	10	FF	12	012F	FE	4C	35	35	12	80	0A	1B	1B	27	12	12	12	12
	BATCH	13	1D	0013	01	10	FF	13	012F	FE	4C	35	35	13	80	0A	1B	1B	27	13	13	13	13
	LOGON	14	03	0026	02	20	FF	14	0000	00	14	14	14	14	00	0A	1D	1D	21	14	14	14	14
	LOGOFF	15	04	0026	02	20	FF	15	0000	00	15	15	15	15	80	0A	1E	1E	20	15	15	15	15
	SYSOPERO	16	00	0026	02	20	FF	16	012F	00	00	00	16	00	00	0A	16	16	23	00	00	00	00
HOLDING	INTERACTIVE	17	02	0026	02	10	FF	17	012F	00	47	2B	17	08	00	0A	17	17	24	1F	1F	1F	1F
	INTERACTIVE	18	02	0026	02	20	FF	18	012F	00	4B	2B	18	08	00	0A	18	18	25	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	19	03	0026	02	30	FF	19	012F	00	4A	2C	19	08	00	0A	19	19	26	1F	1F	1F	1F
	BULK I-O	1A	03	0026	02	20	FF	1A	012F	00	0A	0A	1A	0A	00	0A	1A	1A	22	1A	1A	1A	1A
SET	BATCH	1B	04	0026	02	10	FF	1B	012F	00	4C	35	1B	0B	00	0A	1B	1B	27	1B	1B	1B	1B
	BATCH	1C	04	0026	02	30	FF	1C	012F	00	35	36	1C	13	00	0A	1C	1C	28	1C	1C	1C	1C
	LOGON	1D	01	0026	02	20	FF	1D	012F	00	14	14	1D	14	00	0A	1D	1D	21	14	14	14	14
	LOGOFF	1E	03	0026	02	20	FF	1E	012F	00	15	15	1E	15	00	0A	1E	1E	20	15	15	15	15
PREJUDICE	WRITE	1F	01	0026	02	23	FF	1F	0000	00	1F	1F	08	08	00	0A	17	17	24	1F	1F	1F	1F
	LOGOFF	20	06	0026	02	20	FF	20	0000	FE	15	15	15	15	80	0A	1E	1E	20	15	15	15	15
WAITING	LOGON	21	06	0026	02	20	FF	21	0000	FE	14	14	14	14	80	0A	1D	1D	21	14	14	14	14
	BULK I-O	22	06	0026	02	20	FF	22	0000	FE	0A	0A	0A	0A	80	0A	1A	1A	22	0A	0A	0A	0A
FOR	SYSOPERO	23	06	0026	02	20	FF	23	0000	FE	00	00	00	00	80	0A	16	16	23	00	00	00	00
	INTERACTIVE	24	06	0026	02	10	FF	24	0000	FE	47	2B	3D	08	00	0A	17	17	24	1F	1F	1F	1F
INTERLOCK	INTERACTIVE	25	06	0026	02	20	FF	25	0000	FE	47	2B	3D	08	80	0A	18	18	25	1F	1F	1F	1F
	INTERACTIVE	26	06	0026	02	30	FF	26	0000	FE	2B	2D	3F	08	80	0A	19	19	26	1F	1F	1F	1F
SET	BATCH	27	06	0026	02	10	FF	27	012F	FE	33	35	27	0B	80	0A	1B	1B	27	0B	0B	0B	0B
	BATCH	28	06	0026	02	30	FF	28	012F	FE	33	35	28	13	80	0A	1C	1C	28	13	13	13	13
	GROWING	29	08	0026	08	10	FF	29	0000	00	2A	2B	3D	08	00	0A	17	17	24	1F	1F	1F	1F
LOOPING	DELAYING	2A	08	0026	08	18	FF	2A	0000	FE	47	29	3D	08	80	0A	18	18	25	1F	1F	1F	1F
	GROWING	2B	18	0026	02	20	FF	2B	0000	00	2A	2C	3F	08	00	0A	18	18	25	1F	1F	1F	1F
INTERACTIVE	GROWING	2C	17	0026	02	30	FF	2C	0000	00	2F	2D	40	08	00	0A	19	19	26	1F	1F	1F	1F
	GROWING	2D	16	0013	01	40	FF	2D	0000	00	30	2E	41	08	00	0A	19	19	26	1F	1F	1F	1F
SET	GROWING	2E	15	0013	01	40	FF	2E	0000	00	31	32	42	08	00	0A	19	19	26	1F	1F	1F	1F
	DELAYING	2F	14	0026	02	20	FF	2F	0000	FE	4B	2B	3F	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	30	14	0026	01	30	FF	30	0000	FE	4A	2C	40	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	31	14	0026	01	40	FF	31	0000	FE	49	2D	41	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	32	14	0013	01	40	FF	32	0000	FE	49	2D	42	08	80	0A	19	19	26	1F	1F	1F	1F
	GROWING	33	13	0026	08	10	FF	33	012F	00	34	35	34	34	00	0A	1B	1B	27	34	34	34	34
LOOPING	DELAYING	34	11	0026	08	18	FF	34	012F	FE	4C	33	33	34	80	0A	1B	1B	27	34	34	34	34
	GROWING	35	1C	0026	02	20	FF	35	012F	00	33	36	39	39	00	0A	1B	1B	27	39	39	39	39
BATCH	GROWING	36	1B	0026	02	30	FF	36	012F	00	39	37	3A	3A	00	0A	1C	1C	28	3A	3A	3A	3A
	GROWING	37	1A	0013	01	40	FF	37	012F	00	3A	38	3B	3B	00	0A	1C	1C	28	3B	3B	3B	3B
SET	GROWING	38	19	0013	01	40	FF	38	012F	00	3B	3C	3C	3C	00	0A	1C	1C	28	3C	3C	3C	3C
	DELAYING	39	1A	0026	02	20	FF	39	012F	FE	50	35	35	39	80	0A	1C	1C	28	39	39	39	39
	DELAYING	3A	1A	0026	01	30	FF	3A	812F	FE	4F	36	36	3A	00	0A	1C	1C	28	3A	3A	3A	3A
	DELAYING	3B	1A	0013	01	40	FF	3B	012F	FE	4E	37	37	3B	80	0A	1C	1C	28	3B	3B	3B	3B
	DELAYING	3C	1A	0013	01	40	FF	3C	012F	FE	4E	37	38	3C	80	0A	1C	1C	28	3C	3C	3C	3C
	GROWING	3D	0B	0026	01	10	FF	3D	012F	00	2A	2B	3E	08	00	0A	17	17	24	1F	1F	1F	1F
AWAIT	DELAYING	3E	0B	0026	01	10	FF	3E	012F	FE	47	2B	3D	08	80	0A	17	17	24	1F	1F	1F	1F
	GROWING	3F	18	0013	01	20	FF	3F	012F	00	2A	2C	43	08	00	0A	18	18	25	1F	1F	1F	1F
INTERACTIVE	GROWING	40	17	0010	01	30	FF	40	012F	00	2F	2D	44	08	00	0A	19	19	26	1F	1F	1F	1F
	GROWING	41	16	0004	01	40	FF	41	012F	00	30	2E	45	08	00	0A	19	19	26	1F	1F	1F	1F
SET	GROWING	42	15	0004	01	40	FF	42	012F	00	31	32	46	08	00	0A	19	19	26	1F	1F	1F	1F
	DELAYING	43	14	0013	01	20	FF	43	012F	FE	29	2C	3F	08	80	0A	18	18	25	1F	1F	1F	1F
	DELAYING	44	14	0010	01	30	FF	44	012F	FE	2B	2D	40	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	45	14	0004	01	40	FF	45	012F	FE	2C	2E	41	08	80	0A	19	19	26	1F	1F	1F	1F
	DELAYING	46	14	0004	01	40	FF	46	012F	FE	2D	2E	42	08	80	0A	19	19	26	1F	1F	1F	1F
	SHRINKING	47	07	0026	10	08	FF	47	0000	00	48	29	3D	08	00	08	17	17	24	1F	1F	1F	1F
LOOPING	DELAYING	48	07	0026	10	08	FF	48	0000	FE	47	29	3D	08	80	08	17	17	24	1F	1F	1F	1F
INTERACTIVE	SHRINKING	49	17	0026	01	30	FF	49	0000	00	4A	2C	40	08	00	0A	19	19	26	1F	1F	1F	1F
SET	SHRINKING	4A	16	0026	02	20	FF	4A	0000	00	4B	2B	3F	08	00	0A	18	18	25	1F	1F	1F	1F
	SHRINKING	4B	15	0026	08	18	FF	4B	0030	00	47	29	3E	08	00	0A	18	18	25	1F	1F	1F	1F
	SHRINKING	4C	13	0026	10	08	FF	4C	012F	00	4D	33	4D	4D	00	08	1B	1B	27	4D	4D	4D	4D
LOOPING	DELAYING	4D	11	0026	10	08	FF	4D	012F	FE	4C	33	4C	4D	80	08	1B	1B	27	4D	4D	4D	4D
BATCH	SHRINKING	4E	1B	0026	01	30	FF	4E	012F	00	4F	37	3A	3A	00	0A	1C	1C	28	4E	4E	4E	4E
SET	SHRINKING	4F	1A	0026	02	20	FF	4F	012F	00	50	36	39	39	00	0A	1C	1B	27	4F	4F	4F	4F
	SHRINKING	50	19	0026	08	18	FF	50	012F	00	4C	35	34	34	00	0A	1B	1B	27	50	50	50	50
STARTING	INTERACTIVE	51	17	0010	01	18	FF	51	0000	00	2F	52	3F	08	00	0A	18	18	25	1F	1F	1F	1F
SET	INTERACTIVE	52	16	000C	01	20	FF	52	0000	00	2F	53	40	08	00	0A	19	19	26	1F	1F	1F	1F
	INTERACTIVE	53	15	0008	01	30	FF	53	0000	00	30	30	40	08	00	0A	19	19	26	1F	1F	1F	1F

Figure 6—Schedule table T48—A later research table for release 4

an 1130–2250 to do algebraic symbol manipulation using LISP. After a minor adjustment to the Starting Set he reported that he was satisfied with the response and it was better than before.

3. The initial table also slowed down some assemblies, namely those with many macros. Later they performed as follows:
   a. Assembling a program called SIPCON, over 2000 cards with a few macros, takes one minute and 29 seconds when run stand alone. Formerly under the TSS/360 Release 4 schedule table (Figure 3), with 20–25 users on the system with LCS in use, it frequently took 40–45 minutes. Now its maximum time has been 13 minutes. This occurred once. Assembling it over 10 other times during user sessions when 20 or more users were on showed a range from 2:19 to 6:13. Schedule table T48 (Figure 6) produced the 2:19 assembly.
   b. Repeatedly assembling a program called BROWSE which had about 2800 statements and many macros had the following results when more than 20 users were on:

TSS/360 Release 4 table:	Over one hour
Early new tables:	20 min.—one hour
Table T47:	13–22 minutes
Table T48:	6 minutes, 9 seconds

The results presented in the preceding three sections can be summarized as follows:

1. The IBM T. J. Watson Research Center's Model 67, without LCS, could support about 25 simultaneous users on Release 4 of TSS/360, maintaining a consistent response of 3 seconds or less to trivial requests, while simultaneously servicing large users rather well. Formerly we could support fewer than 15 simultaneous users on Release 4 of TSS/360 without LCS, and responses to trivial requests were in the 10 to 30 second range. There is no evidence to indicate that the simultaneous user load could not be significantly further increased.
2. The controlled experiments using SLIN were poor indicators of performance in a live user session. They were also poor indicators of the worth of most schedule tables since they never used most of the levels in the tables. This was due to the inadequacy of the SHARE script. Thus, a major problem to be tackled is to find a truly representative script (or set of scripts) and, since loads are constantly evolving, it is even more important to develop a methodology for automatically producing scripts that are characteristic of a given installation; and automatically verifying that fact.

   A second problem to be tackled is to define a shorter measurement period which produces valid results. The current technique requires more than two hours per run.

Measured working set sizes

In examining the results of daily readings from the Level Usage Counters I find the statistics to be remarkably similar to those reported under Section B, above. The only time this varied significantly was when one user had left a job to be run in batch that involved manipulating a 961 page matrix. That is more pages than are on our drum. It is over 4 million bytes of memory. Instead of running in the batch, it was run from a terminal, in the user session on January 23, 1970. He was on from 8:15 until 11:15 a.m. This caused erratic response but was not intolerable.

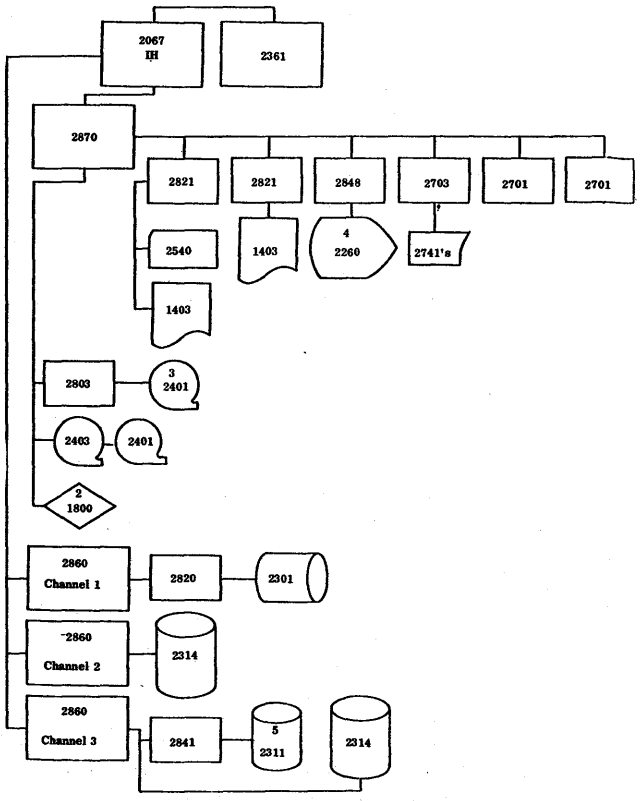


Figure 7—The model 67 configuration at the IBM Thomas J. Watson Research Center

The following compares the two sets of statistics that reflect on Balanced Core and the Working Set Size:

	January 14	*January 23*
<32 Pages	89 percent	80 percent
32 < 48 Pages	10 percent	12 percent
48 < 64 Pages	1 percent	8 percent
Starting Set	28 percent	12.5 percent
Looping Set	30 percent	47 percent
Batch	4 percent	0 percent
Holding Interlocks	5 percent	8.5 percent
Waiting for Interlocks	2 percent	.6 percent
AWAIT	5 percent	9.6 percent
BULKIO	10 percent	12.7 percent
SYSTEM OPERATOR	5 percent	3.3 percent
LOGON	6 percent	3.6 percent
LOGOFF	5 percent	2.2 percent

The primary conclusion drawn from this comparison is that the Working Set Size concept is valid for handling very large programs as well as many small ones. Furthermore, the Principle of Balanced Core Time permits a high degree of automatic folding of programs to occur in a Response Oriented System thus increasing the responsiveness.

BCU measurements

The BCU was only available to us in November and early December. The most significant effect of these measurements was to demonstrate that the user received more CPU time than he had under the Release 4 TSS/360 Schedule Table.

	Release 4.0 Table	*Early New Tables*
CPU/ELAPSED TIME	40–50 percent	70–90 percent
Problem State/CPU	15–25 percent	40–55 percent
Supervisor State/CPU	85–75 percent	60–45 percent

There were also increases in overlapped operations between the Drum, CPU, and our 2 disk channels with the new tables.

There was, however, one day under the TSS/360 Release 4 Table when the CPU/Elapsed Time Ratio reached 90 percent and the Problem Program State received about 50 percent of that time. This was due to one user predominating the system.

It is important to note that the BCU measurements can vary widely as a result of the load. Thus there is a small possibility that a sudden load change accounts for the above characteristics.

SUGGESTED EXTENSIONS

1. The Level Usage Counters can be used to measure Working Set Size Profiles of Programs running Stand-Alone. By varying the Time and Space Limits in the table systematically you could find ideal settings for programs such as EDIT, FTN, ASM, etc.
2. Page Stealing was not available in Release 4. Thus I had to use the Looping Set alone to follow the Working Set Size. Also, I could not distinguish a change in Working Set from a change in Working Set Size. With future releases of TSS/360, it is possible to do the following:
   a. Use the Stealing Percentage to determine if the Working Set Size is changing. If it is, flush out all changed pages, reset the Page Count to 0 and restart at the same level. If the Stealing Percentage is again exceeded, change to a level allowing more core for a larger Working Set Size.
   b. Set up a sequential set of levels of the table, using the Balanced Core Time Principle to establish time and space values for each level. When a program exceeds Time with no stealing, use the page count of the pages used to index into this sequential set of levels. This will enable the Working Set Size to be followed more closely.
   c. The Looping Set is still important for load distribution. Consider keeping a Balanced Core Time Measure in each task's TSI. This could be used in combination with the Table Level Parameters to determine when to delay.
3. It is important to know the frequency with which the Working Set Size changes over various classes of programs. It is possible for blocked paging to hurt performance if the wrong Working Set Size estimate is used and the change in size is frequent. Again the Level Usage Counters could be used to determine this for various classes of programs run stand-alone.
4. I do not yet know how to handle the Waiting-for-Interlock Set properly. There should be a closer correlation between the Waiting and Holding Interlock sets since I suspect that they flip flop rapidly. This is an important area for investigation. This is because it can have a serious impact on all tasks.

5. Different Starting Sets can be set up for users with classically different terminals such as graphic devices, Remote Job Entry, typewriters, and input devices using displays for output.
6. Since the preponderance of programs execute well with Working Set Sizes smaller than 32 and even 16 pages, a careful study should be made in this area.
7. It is important to use the estimated Working Set Size of the Level about to be used to determine if sufficient core is available to dispatch a task. This may completely eliminate low core problems. Currently, the estimate is the number of pages used by the task in its *previous* level.
8. As the level of multiprogramming increases, the importance, and even the usefulness of page blocking decreases to a point where it may be negative. The primary purpose of blocking is to increase the level of utilization of the slower device. If the level of multiprogramming is sufficiently high then the paging device will be used optimally across all users. Furthermore no unwanted pages will be brought in. On the other hand, for low levels of multiprogramming, page blocking for individual users may be beneficial since there is spare power in the paging device.

SUMMARY

This paper presents results of concepts which demonstrate the importance of relocation hardware, not only to Time Sharing Systems, but also to good Multiprogramming Batch Systems.

It describes an initial implementation of the Principle of Balanced Core Time and its effect on responsiveness in a Time Sharing System. It describes the Concepts of Working Set, Working Set Size, Program Locality of Reference, and demonstrates the improvements to be realized by the proper application of these concepts to multiprogramming systems.

Thus, the work described in this paper is the result of the ideas of many men brought together to demonstrate the potential fruitfulness of those ideas when fully implemented. This initial approach has been an engineering one. Thus, while I believe that these early results are dramatic, there is still room for very large improvements.

ACKNOWLEDGMENTS

Many papers say that the list of acknowledgments is too long and so they are frequently suppressed. Because of the nature of this work I feel that they should not be suppressed.

I wish to publicly thank my wife, Peg, and my children, for understanding the unusual demands on my time for these past three months. We seem to go through cycles like this every four years. Next I will thank my friend and manager, Hugh Lynch, for his enthusiastic support and criticism and patience. Peter Markstein recognized the importance of this work very early and provided much encouragement. He also taught me about multiprogramming back in 1961. I am grateful to Jim Considine for his keen scientific approach to my work and for his help with many matters pertaining to TSS/360 structure as well as for his help with the Level Usage Counters. Gerry Waldbaum has been most helpful with his modifications to SLIN and suggestions for its use. Gene McGilton, Jim Dobbs, John Mancini, Rufus Denson, and Wally Harrison have provided valuable suggestions on the operational aspects of my work. And they did this at a time when they were very short of help for themselves.

At a different level of detail I recognize the cooperation, help, and support provided by Don Harrison, Jerry Jaggi, Jim Griffin, Dick LaMaire, Bob Haskell, Jim Hancock, Dennis Weston, Denis McLance, Dave Favor, Lloyd Moore, Wayne Deniston and Dan Cease.

I am grateful for the fine work of Mrs. G. R. Ford and Mrs. L. Dilley. They used the REDIT facility of TSS/360 to type and format the first draft of this paper. This enabled many changes to be easily made. The RUNOFF facility of TSS/360 was used for printing the paper.

Finally the people whose ideas I used are mentioned. They are: Bob Nelson for his total understanding of relocation hardware and its ultimate importance; Dave Sayre for his constant expression of the importance of automatic folding of programs; Las Belady, Carl Kuehner, and Brian Randell for their work on Scheduling, Program Locality of Reference and the usefulness of Segmentation; and last but certainly not least is Peter Denning, whose papers describing the concepts of Working Set and Working Set Size have influenced this work.

REFERENCES

1 H HELLERMAN
Some principles of time-sharing scheduler strategies
IBM Systems Journal Volume 8 Number 2 pp 96-98 1969

2 D SAYRE
Is automatic "folding" of programs efficient enough to displace manual?
CACM Volume 12 Number 12 pp 656-660 1969

3 L BELADY
A study of replacement algorithms for a virtual storage computer
IBM Systems Journal Volume 5 Number 2 pp 78-101 1966

4 B BRAWN F G GUSTAVSON
Program behavior in a paging environment
RC 2194 IBM Thomas J Watson Research Center
Yorktown Heights New York 1968

5 P J DENNING
Virtual memory
Technical Report Number 81
Princeton University Princeton New Jersey 1970

6 P J DENNING
The working set model for program behavior
CACM Volume 11 Number 5 1968

7 IBM CORPORATION
System/360 time sharing system resident supervisor
Program Logic Manual Form *Y*28-2012

8 IBM CORPORATION
System/360 time sharing system, system control blocks
Program Logic Manual Form Y28-2011

9 M V WILKES
A model for core space allocation in a time-sharing system
AFIPS Conference Proceedings Spring Joint Computer Conference Volume 34 p 265 1969

10 W R DENISTON
SIPE: A TSS/360 software measurement technique
24th National ACM Conference Proceedings p 69 1969

Time-sharing for OS

by ALLAN L. SCHERR and DAVID C. LARKIN

International Business Machines Corporation
Poughkeepsie, New York

INTRODUCTION

The objective of the Time Sharing Option (TSO)[1] was to provide a general purpose time-sharing capability within the existing framework of the System/360 Operating System. General purpose time-sharing includes:

- A natural command language in which remote terminal users define their work.
- Support for basic applications such as text editing, problem solving, and program development and testing in a variety of languages.
- Extendibility—a simple way for terminal users or installation management to add specialized interactive applications to the system.

The ground rules for providing this function within the operating system included:

- Preserving compatibility with current OS data set formats and access methods.
- Following OS conventions for user program interfaces.
- Allowing no degradation of existing OS function because of the presence of time sharing in the system.

TSO is designed for System/360 Models 50 and up, with 512K bytes of main storage to operate both batch and time sharing concurrently. With 384K bytes, either time sharing or batch processing can be active, but not at the same time. TSO will also operate on the System/370 Models 155 and 165. TSO will be available in the first quarter of 1971.

TSO FUNCTIONS

We shall look first at the functions and capabilities provided by TSO, and then at how this support was incorporated into the operating system.

The TSO command language is similar in style to some existing terminal-oriented or time-sharing systems, such as CALL/360-OS,[2] Dartmouth BASIC,[3] CTTS,[4] and CP67/CMS.[5] Commands are simple English words or abbreviations: for example, "RUN" to compile, load, and start a program, or "LINK" to invoke the Linkage Editor. Command operands provide flexibility for varied function, but almost all operands normally take a default value, and do not have to be entered. When required information is omitted, the system prompts the user for it. Explanations of command functions and formats are available through a HELP facility. Most messages have been kept brief, but when a novice needs further explanation of a message, he can receive it by typing a question mark.

The command language can be used to: [6]

- Enter, store, modify, and retrieve data at the terminal.
- Solve problems.
- Develop and test programs.
- Execute programs, either interactively, in the time-sharing environment, or by submitting them for batch execution.
- Control most of the operation of the system from a remote terminal.

The installation management determines what capabilities will be made available to each terminal user. A profile of each user's typical processing requirements and authorizations is kept within the system.

The profile is used to allocate resources for the user when he activates his terminal to start a time-sharing session or "logs on" to the system. A new dynamic allocation facility allows the user to create new data sets, or allocate old ones, during the terminal session. The user does not have to specify these data sets in advance, as he would in a batch environment.

Text and data handling

Commands are available to enter, store, edit, and retrieve data sets consisting of text, data, or source programs. For example, the text for reports can be created, stored, edited, and printed in a format of the user's choosing at the terminal. Data sets can be edited by referring to each data record by a line number, or by referring to the data by context. For instance, a particular string of characters, say a symbol name, can be changed to another string, in a single line or wherever it occurs throughout a program, with a single command.

Problem solving

Three language processors specially designed for mathematical problem solving by users who are not necessarily professional programmers are available under TSO. Two are components of the Interactive Terminal Facility (ITF):

- ITF: BASIC, a simple algebra-like language easily learned by anyone familiar with mathematical notation.
- ITF: PL/I, a subset of the full PL/I language, that provides a powerful conversational language. ITF: PL/I statements can be executed line-by-line, as they are entered, or collected into procedures and subroutines for later execution.

An error in either ITF language can be detected as soon as the incorrect statement is entered, allowing the user to correct it before going on. An interactive debugging feature allows the problem-solver to test his procedures in ITF: BASIC and PL/I using his own statement numbers, labels, and variable names.

The third problem-solving language is Code and Go FORTRAN, a version of the full FORTRAN IV language with some extensions included for the time-sharing environment. Fields in the FORTRAN source statements do not need to begin in particular "columns" of the statement, but can be entered free-form. A new list-directed I/O statement is provided to simplify access to the terminal from an executing FORTRAN program. The FORTRAN syntax checker and the DEBUG package are provided for statement scanning and program testing.

Program development

The programmer is able to invoke from his terminal any processor or most programs written to operate under the operating system. In particular, he can call FORTRAN, COBOL, PL/I, ALGOL, assembler language, and other OS-compatible language processors for his source programs. Any of these languages can be used to write interactive programs, since standard OS sequential access methods (BSAM and QSAM) can be used for I/O directed to the terminal.

Some language processors specially modified for the terminal environment are available with the command language. These include an assembler (F), FORTRAN IV (G1), the PL/I Optimizing Compiler, and American National Standard (formerly USAS) COBOL Version III. The modifications to these processors include prompters, to ask the user for compilation options, and specially formatted diagnostic messages and listings designed for the terminal. Because of the compatibility between the batch and time-sharing environments, programs can be developed and tested at the terminal for eventual batch execution. Source language syntax checking is provided for PL/I (F) and all levels of FORTRAN, either line-by-line, as the program is keyed in, or by whole programs.

The TEST command allows programmers to start and stop programs, inspect and change the contents of main storage and registers during execution, trace program flow, and display the value of variables during execution. The TEST command can be entered whenever any program terminates abnormally, or whenever a user interrupts an executing program with the terminal attention key.

Command language extension

The TSO command language is open-ended; any user can add commands to the existing set. Each command entered from the terminal invokes a command processor to perform the function associated with the command name. Command processors are unprivileged programs, and any program can be defined as a command by adding the load module to a command library. Commands can be added to the system command library to be available to all system users, or to a private command library, to be available to only one user or a subset of the system users.

A set of service subroutines is available for use by all command processors written in PL/I or assembler language. These routines implement common functions and ease the task of creating new command processors. The functions provided include:

- Command syntax scanning and parsing.

- Message formatting, and other I/O handling, allowing terminal device type independence.
- Allocation and freeing of data sets as they are needed.

NEW CONTROL PROGRAM FUNCTIONS

TSO has been implemented on the MVT configuration of the operating system, to take advantage of the existing program sharing, subtasking, and storage protection facilities, all desirable in a time-sharing environment. With a few modifications and extensions, the existing job management and task supervision functions were used, ensuring continued compatibility between the time-sharing and batch environments. For each user logging on to the system from a remote terminal, an OS job is created. A large number of these time-sharing (or foreground) jobs, as well as batch jobs, share the available resources of the system. In particular, the jobs created for terminal users can share a foreground region of main storage. A single terminal job is brought into a foreground region and allowed to execute for a short time slice. The other terminal jobs are saved temporarily on an auxiliary storage device. At the end of the active job's time slice, or when it starts waiting for I/O from the terminal user, a main storage image of the job is copied out to auxiliary storage, and another job is brought into the region for its time slice. One or more regions of main storage can be defined as foreground regions, and remaining storage can be used for batch (or background) jobs, up to a total of 14 regions.

The process of copying job images back and forth between main and auxiliary storage shown in Figure 1 is called swapping. A dedicated access method supports swapping to 2311, 2314, and 3330 disk storage, and 2301, 2303, and 2305 drum storage. The channel programs are tailored to the individual devices in use. The process can be set up so that two swap devices are used in parallel, doubling the effective swap rate. In case a particular device becomes full, a spill mechanism is provided. For example, if a 2301 drum is normally used for swapping, but becomes full under a peak load, the overflow can be directed to another device.

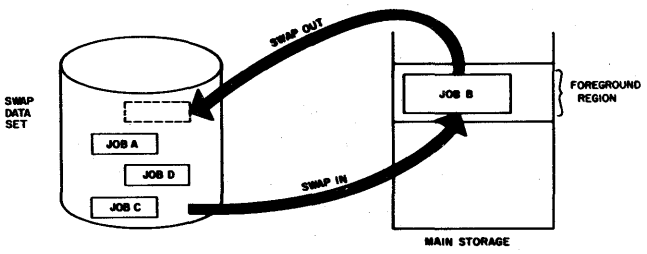


Figure 1—TSO foreground job swapping

Before a job can be swapped out of main storage, it must be "quiesced," or forced into an inactive state. I/O requests are intercepted, or allowed to complete if already in progress, and control blocks are "unhooked" from system queues. Most control blocks, including Task Control Blocks (TCBs) and I/O buffers, are kept in a reserved area within the foreground region, and are swapped out along with the job. Only that portion of the foreground region that the job is actively using is swapped out.

I/O buffers for the terminal are an exception to the quiescing procedure. All I/O for the terminal is handled through the Telecommunications Access Method (TCAM). Because of the relatively long periods of time involved in terminal I/O, quiescing is not practical. Terminal buffers are not swapped out along with the job, but are kept in supervisor main storage. Thus a user at the terminal is not aware of the swapping process: his terminal keyboard remains unlocked and he can enter input whether his job is in main storage or not.

CONTROL FLOW

Figure 2 is a generalized diagram of the flow of control among the routines implementing the time-sharing subsystem. These routines are brought into main storage only when the operator starts the time sharing operation; the system can be started beforehand for batch processing. If the operator stops time-sharing, the associated storage is again made available for batch jobs.

At the highest level of control under the MVT supervision routines are the Time Sharing Control Task and the TCAM Message Control Program. The Control Task handles system-wide functions such as initialization and control of foreground job swapping. The Message Control Program handles all I/O for terminals, both for TSO foreground jobs and any other teleprocessing applications that may be present in the system.

Below the Time Sharing Control Task is a Region Control Task for each foreground region. It supervises those terminal jobs assigned to its region, and handles the quiescing function. A LOGON/LOGOFF Scheduler is invoked for each user logging on to the system. This routine builds the internal job control language necessary to define the user's terminal job to the system, using information stored in the user profile and in a cataloged procedure specified by the user.

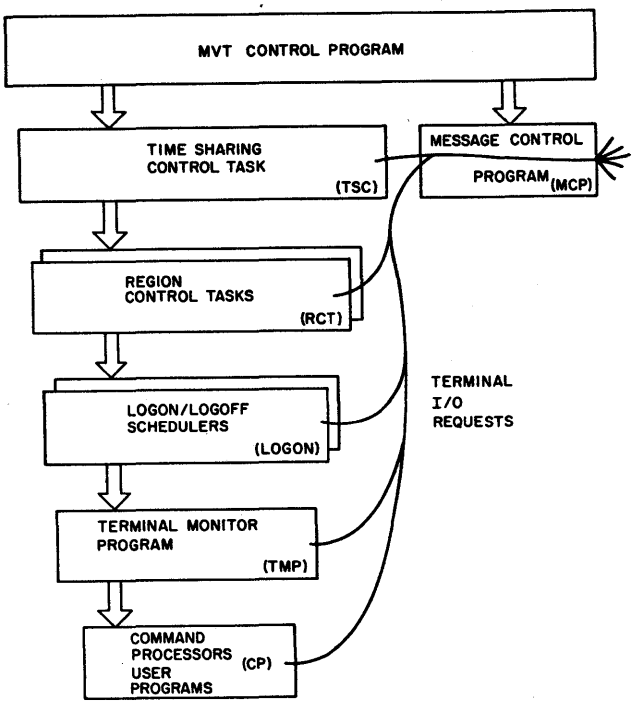


Figure 2—System control flow

LOGON invokes, through MVT job scheduling routines, a problem program, called the Terminal Monitor Program, that handles both TSO and user-defined commands. When the user enters a command, the Terminal Monitor Program attaches the appropriate command processor or user program as a subtask.

Important decision-making functions of the control program have been isolated in a single component called the Time Sharing Driver, shown in Figure 3. Each of the other routines continually passes information on system events to the Driver—time slice end, swap out complete, a job waiting for terminal I/O, etc. From this stream of information, the Driver maintains a picture of the current system workload, and based on this information, makes decisions about what should be done next. The Driver orders actions to be carried out by other control routines through a parameter list interface. The Driver itself is entirely insulated from the rest of the system by the Time-Sharing Interface Program. This design allows installations with interests in special-purpose or experimental time-sharing systems the freedom to replace the Driver with a scheduler of their own design.

SYSTEM TUNING

Time-sharing system performance is often measured by response time at the terminal. Response time is in turn largely determined by three factors: [7,8,9,10]

- The number of users logged on to the system.
- The average user "think" time per interaction (one give-and-take between the user and the system).
- The average service time, per interaction, including processing and swapping overhead.

The second two factors depend on the type of processing the users of the system require. A system set up to handle the relatively short think times and processor times typical of text-editing applications would not perform optimally for a set of users running lengthy numerical analysis programs. To allow for varying installation and user requirements, the TSO Driver accepts controlling parameters from the installation. These variables describe system configuration, such as the number of foreground regions to maintain; they may request the Driver to use one of several algorithms it has for scheduling use of system resources; they may specify constants used in the algorithms. This flexibility allows the installation to tailor the scheduling and time slicing functions to best serve its mix of jobs.

For instance, consider an installation whose users typically do two types of processing—problem solving with one of the conversational languages such as ITF: PL/I, and mathematical calculations that are long-running, compute-bound jobs. This installation will

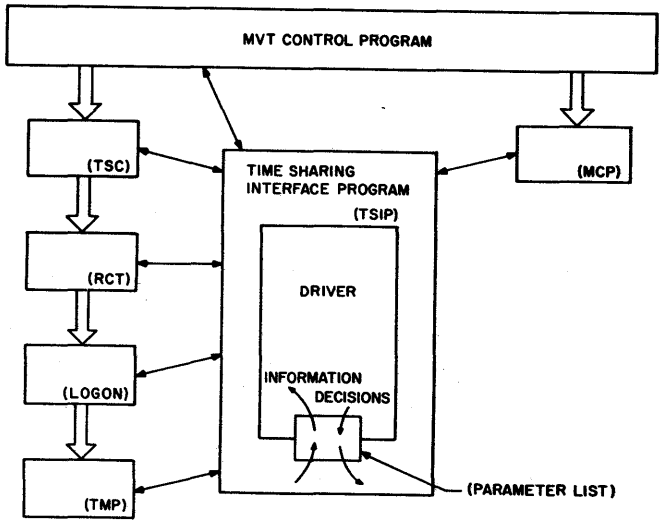


Figure 3—The TSO Driver

probably wish to optimize response to the conversational users, rather than use a simple round-robin type of scheduling.

TSO allows the installation to set up multiple service queues for the jobs assigned to each time-sharing region. In this case, two queues could be defined, with the higher priority queue for the conversational jobs. Jobs on the queue could be given a four or five hundred millisecond time slice—enough to complete almost any single transaction in a problem-solving language. A job that doesn't finish a transaction within a time slice would be shunted to the second queue, where it would be allowed a five or ten second time slice, depending on the number of jobs on the queue. Anytime a job on the higher priority queue is ready to execute —typically when it receives a line of terminal input —it will be swapped in for its time slice immediately, interrupting any second-queue job that happens to be executing. This preemptive scheduling will reduce response time to conversational requests to the minimum, at the cost of lengthening somewhat the time for completion of the compute-bound requests.

Among the scheduling variables that can be set by the installation are the number of service queues for each region, the average service time for each queue, the minimum time slice guaranteed to each job on a queue, the thresholds—either in execution time or program size—that determine to what queue a job is assigned, and the method to be used for dividing execution time among multiple foreground regions and the background regions.

CONCLUSION

With the addition of logic to handle the swapping of terminal jobs in and out of foreground main storage regions, the existing MVT control program capability for multiprocessing was found to be a suitable base for a general purpose time-sharing system and, through the natural command language, the full services of the operating system are available to remote terminal users.

REFERENCES

1 *IBM system/360 operating time sharing option: planning for TSO*
IBM Data Processing Division Form GC28-6698 White Plains 1969 1970

2 *CALL/360-OS description manual*
IBM Data Processing Division Form H20-0673 White Plains 1969

3 J G KEMENY T E KURTZ
BASIC
Dartmouth College Hanover 1966

4 P A CRISMAN ed
The compatible time sharing system: a programmer's guide
MIT Press Cambridge 1965

5 *Control program-67/Cambridge monitor system*
IBM Cambridge Scientific Center Cambridge 1968

6 Certain of the functions described are provided as IBM Program Products, available for a separate license fee

7 G ESTRIN L KLEINROCK
Measures, models, and measurements for time-shared computer utilities
Proceedings 22nd ACM National Conference Thompson Book Company Washington DC

8 J M McKINNEY
A survey of analytical time-sharing models
Computing Surveys June 1969

9 A L SCHERR
An analysis of time-shared computer systems
MIT Press Cambridge 1967

10 A L SCHERR
Time-sharing measurements
Datamation Vol 12 No 4 April, 1966

SPY—A program to monitor OS/360

by R. SEDGEWICK, R. STONE, and J. W. McDONALD

Western Electric Engineering Research Center
Princeton, New Jersey

INTRODUCTION

It is generally agreed that one of the major problems facing the manufacturers of large scale computer systems today is the problem of measuring the performance of a computer in conjunction with the operating system which drives it. This problem is under consideration for operating systems currently in the design and implementation stages through studies on: (i) the establishment of reasonable criteria under which performance can be measured; (ii) means of actually making the measurements; and (iii) ways of using the information obtained to optimize the performance of the system under the established criteria for a particular user's environment.

For existing operating systems the problem is somewhat different, especially in cases where it is not feasible to make any changes to the operating system itself. First, extracting information from the system is a non-trivial problem under the constraint that the system cannot be modified. Second, the information obtained is little more than academic unless it reflects inefficiencies in the system caused by the environment of user jobs being processed, as optimization can only be effected by restructuring this environment or by setting parameters to direct the system to operate according to the environment.

A related problem that has arisen with the advent of time-sharing and multiprogramming systems is that of monitoring system execution. This is an important area of research for two reasons: First, one characteristic of many existing operating systems is that they just cannot function without a fair amount of operator intervention, and the operator must know exactly how the resources of the system are being utilized in order to carry out his job effectively. Second, situations often arise in a multiprogramming or time-sharing environment in which it is not altogether clear exactly what action is being taken by the system. The ability to observe the performance of such a system in operation is of importance to systems programmers and managers to enable them to more effectively "tune" the system and watch the local effects of any changes made. This ability can also be of value to users of a machine—the typical user has no concept of his impact on the operating system, and a certain amount of awareness can enable him to better appreciate how incorrect estimates of his system resource requirements can affect the entire user community.

A considerable amount of work has been done in the general area of system performance measurement in the past, and a wide variety of approaches to the problem are represented.[4,5,11] In general, however, they can be roughly categorized into three classes: theoretical studies (simulations and statistical work); hardware monitoring devices; and software data gathering techniques.

The theoretical studies[2,8,10] generally focus on the much broader problem of improvement of operating system design so that maximum efficiency can be provided under very carefully thought out criteria. The process is to make use of thorough statistical analyses in conjunction with generalized computer system models to develop a coherent theory which clearly defines (and generally suggests solutions to) basic problems in operating system design related to overall system performance. This work is invaluable to the system designer, and is a fundamental step in the development of future computer systems, but it is in many cases not directly applicable to the problems associated with existing systems, especially when the point of view is taken that the internal mechanism of the system must be viewed essentially as a fixed entity.

Hardware measurement[3,6] is accomplished through the use of an independent set of instruments which have the capability of sensing, decoding, and recording selected electronic signals in the system being measured. This approach to the problem has the decided advantage that the measurement device can be used indiscriminately with little effect on normal system opera-

tions—a voluminous amount of data can be obtained on the operation of the system in its natural state. The danger, of course, exists that a good deal of extraneous data could be obtained—the amount of useful data is dependent on the sophistication of the interface to the host computer and the amount of flexibility in the measurement device itself. (An example of a very flexible device can be found in the SNUPER COMPUTER[3] project, where an auxiliary computer, which can be programmed to accept or reject available data, is used as a measurement device.) The main disadvantage in the use of hardware techniques is that it is often difficult to correlate the data being gathered with the software being run on the machine or the software representing the internals of the operating system. The data gathered often shows very clearly how effectively the hardware is being utilized, but aside from reconfiguring the hardware it is in general not clear what steps could be taken to improve upon the observed results.

Software measurement techniques[9,12] generally consist of data gathering programs which run while a system is in normal operation, and reduction procedures which extract useful measurements from the large volume of data thus obtained. The differences in implementation can be found in the types of events monitored, the rate at which they are monitored, and the algorithms used in the data reduction process. There are many different schools of thought in all of these areas—one obvious reason is that measurement of an operating system is quite dependent on the structure of that system. Not only are the data gathering techniques a direct function of the system being studied, but also significant events in one system are quite different from significant events in another. The principal difficulty with software measurement is that the measurement procedures themselves utilize some percentage of the resources of the system—every effort must be made to minimize this percentage.

When considering software measurement techniques, one must make a clear distinction between measuring the operation of a system from within and observing it from the outside. One feature common to most existing system measurement programs is that they are being developed by people who have the luxury of being able to work with the internals of the system being measured. This allows them to facilitate somewhat the process of extracting information from the system, and to provide very accurate measurements of relevant system events. This type of observation from within can be done at many levels—the most desirable method is to provide for insertion of "hooks" in the system early in the design process, and allow for a workable interface between the measurement procedures and the operating system, so that the system can be made responsive to the measurements obtained. (An example of such a system is the XDS Sigma 5/7 BTM system.[7]) However, the point must again be emphasized that in many cases the operating system must be viewed as an unalterable product of the manufacturer: Measurement must be done by merely looking at the internals; and optimization can only be effected by rearranging the environment in which the system operates.

There is far less work in the literature on the subject of dynamically monitoring system execution. Although some capabilities in this area are necessarily provided in most operating systems, a major obstacle is that output is normally done on a typewriter-like console—the state of a multiprogramming or time-sharing system changes far too quickly for such a device. A graphic display is a much more appropriate device for dynamic monitoring for at least two reasons: First, *all* changes in the system can be recorded, allowing the observer to choose those he may consider significant as he watches the system in operation. Secondly, interactions within the system can be easily observed because all information is presented simultaneously—the effect of an activity in any part of the system can be seen in other sections, as it develops. This type of global view of the dynamics of the system would not be possible on anything other than a graphic display. An excellent example of work in this area is the GDM system at Project MAC,[1] which provides dynamic displays of a very flexible format allowing observation of the operation of the MULTICS time-sharing system.

The subject of this discussion, the SPY project, represents one approach to this dual problem of measuring and monitoring operating systems for an existing multiprogramming system—a version of IBM's operating system, OS/360. A system was developed which presents information concerning the jobs being processed by the operating system, the direct access devices connected to the computer, and the state of the System/360 CPU. Data is in general gathered through software techniques, but a facility for rudimentary hardware monitoring is also employed. The information is displayed dynamically and in real time on a graphic display unit and selectively saved on paper tape for later analysis. No modification whatsoever to OS/360 is involved, and a negligible amount of overhead is incurred, so that the system can be used to analyze and observe the operation of OS/360 in day-to-day operations.

Concepts and definition of terms

The version of OS/360 to be analyzed was MVT (Multiprogramming with a Variable number of Tasks),

and any further discussion of our implementation of a monitor would be impossible without the use of some of the terminology and concepts related to OS/360, MVT, and the System/360 hardware. The intent of this section is to provide a rudimentary explanation of the most basic terms.

IBM defines *multiprogramming* as "a general term that expresses use of a computing system to fulfill two or more different requirements concurrently."[14] In general "requirements" are defined in terms of *tasks*, where a task is the smallest independent unit of work for the processor. In multiprogramming, the effect is as though all tasks in the machine at one time were running asynchronously. Tasks can be *system tasks* (readers, writers, schedulers, etc.) or *user tasks*. The users of the system generally submit their work to the system in terms of *jobs*, the individual parts of which are broken off as tasks by the system.

The *computer resources* are those parts of the hardware that can be allocated to individual tasks. The resources about which we will be concerned will be core storage, the central processing unit, and disk (direct access) storage—all of which will be referred to together in this paper as the *resources*.

A *volume* for the purposes of this discussion will be a single disk pack and a *unit* will be the disk drive upon which the volumes can be individually mounted.

The *wait light* is one of five lights on the console of the 360/50 which are meant to indicate the "state" of the machine. In normal operations, when it is off, the CPU is in the process of executing an instruction, and when it is on, the CPU is waiting, generally for some I/O activity.

Hardware configuration

The specific system being analyzed was running on a IBM System 360 Model 50 with 512K bytes of core with an eight drive 2314 direct access storage facility, plus various other peripheral equipment. During development, a 1024K byte 2361 LCS unit and a second 2314 with four drives were added. The monitor system was implemented on this machine and on a PDP-9 with 8K of core equipped with a Graphic-II Display Unit. Communications between the two machines was done via a Parallel Data Adapter (PDA) connected to a 2701 Data Adapter Unit on the 360 and to a specially designed interface on the PDP-9. In addition, the PDP-9 has the hardware capability of monitoring the state of the wait light on the S/360.

It is fully realized that this is a somewhat peculiar hardware configuration, and some effort is being expended to modify the system to allow it to be run on more standard configurations. The only equipment

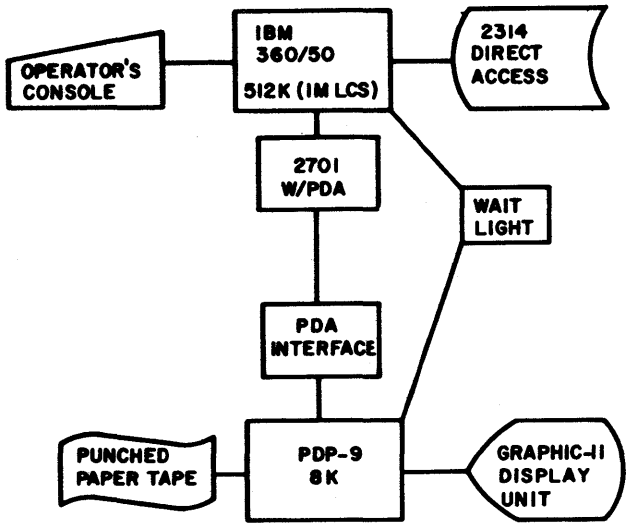


Figure 1—Hardware configuration

absolutely essential is an S/360 Model 30 or higher and some suitable output device—the above equipment was chosen for use simply because of its availability.

Software structure

The final system consists basically of two software packages: one in the 360/50, which takes samples on selected information within OS/360, accumulates statistics, and sends information to the PDP-9 at selected intervals; and one in the PDP-9, which receives the information, independently monitors the wait light, formats all of the information for display, and produces a hard copy record of the important statistics.

The emphasis in implementation of the programs on the 360 was in modularity, and the final system consists of a set of assembly language subroutines. The modules comprising the OS/360 interface portion of the program were debugged in PL/I, so that use could be made of the powerful I/O and error correcting facilities of the higher-level language. Working modules were then translated into PL/360 to gain the advantages of high speed and low storage requirements implicit in assembly language level programs. This method resulted in the production of efficient code quickly, as the advantages of both levels of programming were made use of, and there was little effort in conversion due to the structural similarity of the languages. The transmission and central control modules of the system were written and debugged in assembly language, simply because the tasks to be performed required low-level interfaces to the system.

All of the PDP-9 software for SPY was written and debugged in assembly language on the PDP-9, chiefly because of the lack of any workable higher-level language, and because of the necessity for low-level interfaces to the special purpose I/O devices (the PDA and the wait light).

GENERAL DESCRIPTION OF INFORMATION RETURNED

The principal output of the SPY system is a dynamically changing graphic display consisting of a wide variety of information relating to the operation of the 360/50 and OS/360. The display is divided into four sections (see Figure 2): one containing specific information concerning the jobs being processed; one showing the utilization of the direct access devices (on which the system is heavily dependent); one containing running averages of selected summary information (the information used in system measurement); and one containing a continually recycling graph showing the percentage of CPU utilization. All of the information is updated dynamically as the events it reflects occur, under the constraints outlined in the section below on timing and sampling.

Also output is a punched paper tape containing the data reflected in the summary statistics as well as some data retained from the samples of the wait light.

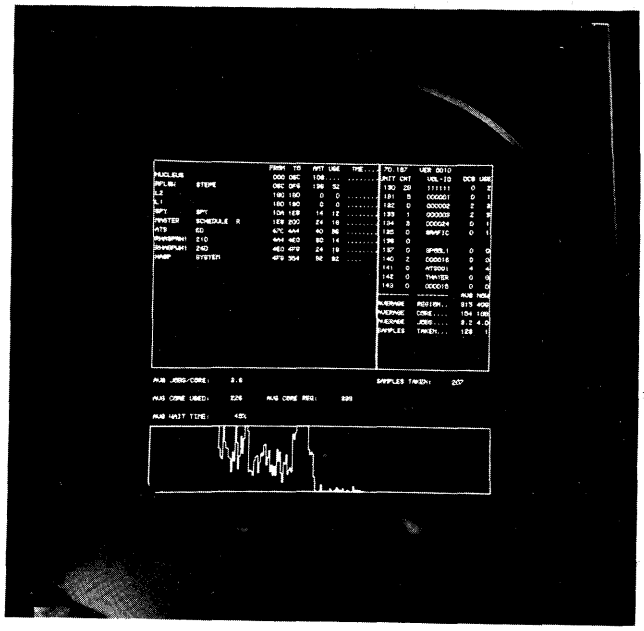


Figure 2—Graphical output from SPY

Identification information is also included, so that a large amount of data can be collected over large periods of time and analyzed statistically to aid in drawing some general conclusions on overall system performance.

Timing and sampling

Most of the activity in the SPY system is regulated by a set of three basic timed intervals. In the S/360, these are maintained with the use of the standard task timing facility (STIMER), which allows the system to operate by taking control of the CPU only for a very short time at the end of each interval and relinquishing it during the interval. It is recognized that this practice of relying on the operating system to do the timing may be somewhat unsafe, since irregularities in the timing algorithms of the operating system could lead to irregularities in the data gathered. However, the PDP-9 utilizes a hardware clock for its timing, so that it is able to ensure accurate timing and program synchronization—the STIMER facility is relied upon solely to relinquish control for real time intervals and has proven adequate for this purpose.

So that the system could be made useful as both a measurement and a monitoring tool, two intervals were needed for timings during the execution of SPY. First, a "*transmission interval*," which represents the time between transmissions to the PDP-9, and hence the time between changes in the graphical display. A second interval needed is the "*sampling interval*" or time between the accumulation of samples on the S/360.

A lower bound on the choosing of a transmission interval was set by the ability of the eye to follow changes in the information presented, and an upper bound resulted from the fact that selection of long intervals resulted in the missing of changes in the system. A reasonable compromise between these two limits appeared to be ten seconds.

The selection of a sampling interval was governed by the need to have a statistically significant number of samples, while not spending so much time sampling that performance might be degraded on smaller or slower systems. The choice of one second resulted in an acceptable number of samples and an extremely low overhead.

Sampling of the wait light in the PDP-9 represents measurement and monitoring of the S/360 hardware rather than the OS/360 software, so that different criteria are used in the timing. The most significant point is that it was considered necessary to sample

the state of the wait light as often as possible—this is done every 1/60 of a second, which is the granularity of the standard clock on the PDP-9. An intermediate average is computed for display purposes every 5/6 of a second, and a final average retained before the graph is recycled. The aim was to produce a dynamically changing graph which moves fast enough so as not to seem frustratingly slow to the observer, and slow enough so that the data displayed is in some way significant.

In addition, it was necessary to define a third interval, a *"statistics gathering interval,"* at the end of which the measurement data is output on hard copy for later analysis. The length of this interval is dictated by the amount of data that needs to be saved, and the rate at which it is being accumulated. In the current implementation of the system, a value of two hundred seconds is used. (This value was chosen for convenience only, as it represents the amount of time it takes the wait light graph to complete one cycle.)

Scheduled jobs and tasks

The information given in this section of the display allows monitoring of the various user jobs and system tasks that have been selected for execution by the operating system. There is one entry for each such task, and for each entry, the following information is given:

(i) An identification of the entry (the job name, step name, and procedure name).
(ii) "FROM" and "TO" addresses indicating the locations in main core assigned the program by the operating system.
(iii) The amount of core requested by the program (and the amount given it by the operating system—this number is equal to the difference of the "FROM" and "TO" addresses).
(iv) The amount of core actually being used by the program (in general very different from the amount requested).

		FROM	TO	AMT	USE	TME
NUCLEUS		000	054	84		.....
JOB2	EXECUTE	054	090	60	60	466.....
JOB3	EXECUTE	090	0D6	70	70	470.....
JOB4	EXECUTE	0D6	126	80	80	473.....
WTRLCS	00E	5C2	5D0	14	10	.....
SPY	SPY	5D0	5DE	14	12	.....
MASTER	SCHEDULER	5DE	600	34	28	.....

Figure 3—Typical job and task information

UNIT	CNT	VOL-ID	DCB	USE
130	25	111111	1	0
131	0	000001	0	0
132	27	000002	3	4
133	11	000003	3	3
134	0	000004	1	1
135	0	000005	0	1
136	0	000006	0	1
137	0	000000	0	3

Figure 4—Typical direct access device information

(v) The time remaining in the time interval allotted the program step by the operating system. Changes in this number reflect the relative CPU utilization of each job.

This information presents a true "picture" of the condition of the machine in relation to the jobs being processed—allowing continuous monitoring of the treatment of the various jobs by the operating system, and the use (or misuse) of the system by the various jobs. Figure 3 shows a "snapshot" of this section of the display during normal system operations.

Direct access device utilization

This area of the display contains information concerned with the direct access devices attached to the system. For each direct access unit, five pieces of information are displayed:

(i) The unit number identifying the disk drive.
(ii) The volume name of the disk pack currently mounted on that unit.
(iii) A measure of the number of potential users of the unit. (The number of DD cards pointing to that unit.)
(iv) A measure of the number of users actually using the unit. (The number of open DCBs pointing to the unit.)
(v) A number from 0 to 100 representing the percentage of time the user of the unit changes. (A measure of contention on the unit.)

With this information one is able to monitor and gather statistics on the operation of the direct access devices and their utilization by both the users and the operating system itself. This is very useful because one characteristic of OS/360 is that, in general, effective operation of the system is dependent on effective

operation of the direct access devices. Figure 4 shows the information typically displayed in this section. (the CNT field is the number described in (v) above—the other fields are self-explanatory).

Global information

This section of the display contains most of the information used in system measurement. Although it is expected that this section will be expanded in the near future, in the current implementation of SPY it contains summary information concerning the non-system tasks. This information is also useful in monitoring the system, so it is presented on the display as both instantaneous data (the values determined in the sample at the end of the transmission interval), and cumulative data (averages of the values sampled since the beginning of the statistics gathering interval). The following is the data presented:

(i) The average number of jobs in execution (this is a much smaller number than the number of tasks displayed in the section described above—a job can occupy core but not be eligible for execution). This is clearly a very important global measure for a multiprogramming system, as it can be taken to more or less reflect the hardware and software capabilities of the system to run multiple jobs.

(ii) The "average region size" of all of the jobs currently in execution (corresponding to the "amount of core requested" above). This number is an indication of the amount of core given to the users of the system after the space for the operating system and space lost due to fragmentation have been subtracted.

(iii) The average amount of core being used by the jobs in execution (corresponding to the "amount of core used" above). This number, when compared to the average region size, shows the ability of the programs to estimate their core resource requirements.

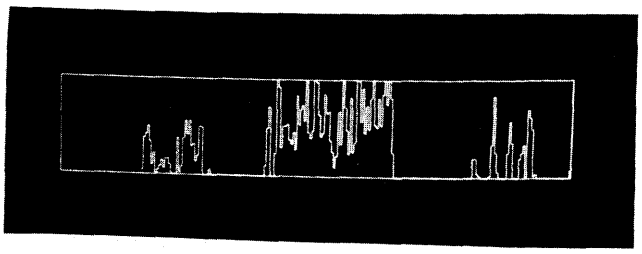


Figure 5—Wait light graph

This information is used in both system monitoring and system measurement to observe the effectiveness of the OS/360 implementation of multiprogramming, the appropriateness of the system parameters that have been selected for the installation, and whether or not the user community is utilizing the system effectively.

CPU utilization (wait light)

The wait light portion of the display consists of a graph on the Graphic-II Display, the X-axis of which is time, and the Y-axis of which is the percentage of time the "wait" light on the 360 is off (i.e., the computer is active).

The result is a dynamic display indicating the degree of activity of the CPU of the S/360 (see Figure 5). Although this information could lead to incorrect conclusions if taken as an absolute measure, this graph has proven very effective, taken in conjunction with the other information displayed, in giving an observer a feel for the load on the CPU at any time.

PROGRAM STRUCTURE

360 program structure

As mentioned above, the 360 program is modular in design and can essentially be divided into three major logical sections: a control program, a transmission package, and an information retrieval package. The control program maintains the three defined sampling intervals and calls the other sections accordingly. The transmission section formats the data for transmission and graphical display and does the actual transmission. The information retrieval package (the OS/360 interface) gathers specified data from the internal workings of the operating system, does the sampling, and puts the data into character format. As required by the design goals, no modification whatsoever is made to OS/360.

This is possible due to the fact that OS/360's data space consists of a series of tables chained together in a complex queue structure.[13] Information about individual tasks in the system is determined by searching the task queue and weeding out the tasks which are active (those tasks which have no "daughter" task on the linked list). Information stored about each task includes its name, core location, core requested, and whether it is a system task. Time information for a task is found from its "mother" task, and actual core utilization is determined by adding the various parts of its region that have not been actually used.

Direct access information is determined by a list of the locations of the status data about the units. This data yields the currently mounted disk pack, whether a mount is in progress, active users, and potential users. In addition, it is possible to trace chains backward to determine the name of the last user, and hence status information is built up concerning the degree of contention.

PDP-9 program structure

The major goal in the development of the PDP-9 program was to produce a clean, readable display, while at the same time acting as a I/O device slaved to the S/360. Functions to be performed were: the reception and display of all the data accumulated by the S/360 programs; the maintenance and display of the wait light graph; and the production of the punched paper tape output. The program gives highest priority to processing related to receiving data from the S/360—at no time does the 360 have to wait for the PDP-9 to accept a transmitted message. If this were allowed to occur, the integrity of the sampling times would be compromised more than necessary.

The main control portion of the program sets up the initial display, initializes the PDA to accept messages from the 360, and sits in a tight loop: awaiting a signal from the PDA handling routine indicating that a complete message has been received; and sampling and maintaining a count on the state of the wait light. At the expiration of every 5/6 second interval, the wait light count is converted into the commands necessary to update the wait light graph, and the main loop reentered. When a message from the 360 arrives, a message decoder determines which information was received and rebuilds the portion of the display which is to be updated. If the message received contained the final average information, an output routine formats and punches out the paper tape hard copy. Upon completion of message processing, the PDA is reinitialized for input and the main loop reentered.

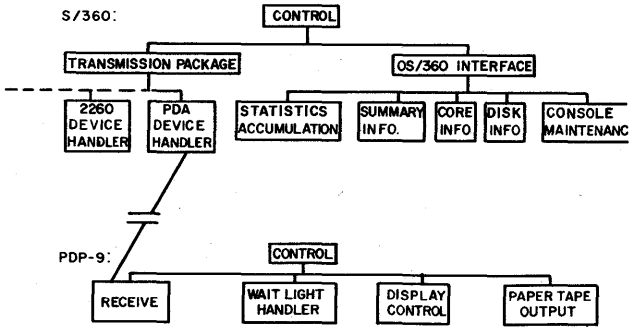


Figure 6—Software structure

COMMUNICATIONS

Introduction to equipment

As mentioned above, the communications link used by the SPY system is a high-speed Parallel Data Adapter interfaced to the IBM 2701. It should be pointed out that transmission at this rate (180,000 bytes/sec) is not crucial to the operation of the system—much slower rates would probably be sufficient. On the other hand, the adapter is almost a hard wire connection between the computers and as such does not require sophisticated low-level programming or extensive error diagnostic routines. The use of the PDA was simply a case of utilizing the most convenient communications device.

Conventions

The communications conventions are rather simple and straightforward—conversation between computers is kept at a minimum and is essentially in one direction with the PDP-9 acting as the receiver and the 360/50 as the transmitter. Acknowledgment of reception is contained and diagnosed within the hardware and the channel of the PDA so that the customary acknowledgment transmission is unnecessary.

The transmission buffers are of a constant length—2048 bytes. The data are preceded by a one byte code describing which type of data format is being sent. All information to be displayed is sent in PDP-9 Graphic-II display code (two 7-bit ASCII characters packed in the low order 14 bits of each 18 bit word) with control characters that determine the physical position on the screen where the data is to be displayed. The "global" information is transmitted in 8 bit ASCII—this is done to make easier the task of producing the punched paper tape. (In future versions of SPY, all character translation will be done in the PDP-9, to reduce overhead on the S/360.)

Transmittal (S/360 program)

The Parallel Data Adapter is a non-supported device under OS/360, and the I/O routines were therefore written at the lowest level allowed users by the system (EXCP—EXecute Channel Program). This has the

additional benefit of keeping core requirements at a minimum.

The PDA is OPENed as an output device once for every run of the program. For every message to be transmitted, the appropriate code conversion is performed (input to the transmission package is EBCDIC), a test made to ensure that the previous transmission was successfully completed, and the EXCP macro issued to direct the system to begin transmission.

Reception (PDP-9 program)

Communications on the PDP-9 is done at the hardware I/O level, with the use of a standard routine developed along with the PDA which does the actual READ into PDP-9 core and sets the various hardware flags to notify the 2701 of proper reception. The main PDP-9 program is notified of the reception of a complete message via a simple "event variable."

INPUT-OUTPUT INTERFACES

Graphic-II display

All information presented by the SPY system is displayed on the Graphic-II display unit, a general purpose display driven by the PDP-9. The display processor generates pictures from a "buffer program" in the PDP-9 core which defines displays in terms of point, line, and character information. The PDP-9 program builds and maintains the buffer program and directs the display interface in presenting and regenerating the display.

With the exception of the wait light graph, all of the information presented is in character form, and this information, along with positioning information, is put into the display buffer by the PDP-9 program. The wait light graph is produced by dynamically modifying the display buffer to define small vectors which trace out the percentages measured.

The wait light

The hardware interface to the S/360 wait light consists of an actual hardware connection from the physical wait light socket on the 360 console to the PDP-9, and an interface on the PDP-9 allowing the state of the light to be tested with a single machine language instruction (S360W—Skip on 360 Wait light).

Operator's Console

It was also deemed necessary to include in SPY an interface to the S/360 operator's console, so that a mechanism could be provided to allow modification of some of SPY's internal parameters at execution time. In addition, the facility was included to allow the operator to request any of the information provided by SPY for output on the console.

The I/O function is performed by sending a request for information to the console (WTOR), resulting in a console message to which the operator can reply at any later time. Rather than waiting for this response from the operator, SPY interrogates OS/360 at the end of every one second sampling interval to determine whether the operator has sent a message. When a message is sent, SPY simulates the occurrence of the end of the transmission-interval, but routes the transmission to the operators console instead of the PDP-9.

Planned additions

Now that a working system has been developed and has proved useful, one direction for future research will be to explore the possibilities of incorporating less expensive display devices into the system, for the purpose of allowing SPY to be generated and used on any System 360 for a modest cost. Since most of the information presented is in character form, this is only a matter of acquiring the hardware and programming new "transmission" packages. The first new device to be included in this way will probably be an IBM 2265 Display Station, or some similar device. It should be pointed out that a major drawback in such a system will be the loss of the wait light graph, although ways of providing some type of replacement for this are also being researched. In general, the emphasis in future development will be on complete generality and modularity in relation to I/O devices. Another area that will have to be researched for versions of the system without the PDP-9 will be in statistics gathering and production of hard-copy results. The development of programs on the S/360 which will save data (either on disks or magnetic tape) is under consideration, but still under the basic constraint of minimizing overhead on the S/360.

A separate area of future work will be in researching ways of making maximum utilization of the hardware currently being used, and developing more special purpose hardware to expand the capabilities of the system. As of this writing, this has already been begun in the implementation of a high-resolution (down to 40 microseconds) clock in the PDP-9 to allow extremely

accurate wait light sampling, and an expansion of the wait light interface to allow other lights on the 360 console to be monitored.

APPLICATIONS OF THE SYSTEM

Analysis of OS/360 operation

Over the past several months, SPY has been used to aid in analyzing the performance of the 360/50 with MVT, and the following conclusions have been reached concerning the operation of this particular computer facility and the impact of SPY upon operations.

The most serious effect of running a highly variable job stream (i.e., jobs with widely differing resource requirements) in a multiprogramming environment on a computer with no virtual memory capabilities is *"core fragmentation."* The core fragmentation problem can be defined as the situation that exists when all of the core resources of the machine are not being utilized because of the fact that the available core is not contiguous. This situation was observed to be most detrimental to system performance.

In addition, two problems were observed in relation to the direct access devices: First, contention was found to be serious on those devices needed by all the users (i.e., the output spool device). Secondly, a problem was observed in the apparent non-reenterability of the device allocation routine in cases where a new volume is required.

In the area of user impact on the operation of the system, the most harmful thing observed was overestimation of the core resource—ranging from 50 percent to 150 percent. The difference between the amount estimated and the amount used is a totally wasted productive resource. In addition, a similar problem was noticed in the use of the direct access devices—users would have the system allocate devices for their use, and then never use them—another wasted resource.

The general conclusion reached was that resources were being underutilized—not only due to possible system inefficiencies (the operating system generated for this particular machine possibly was not completely tailored to the workload being studied), but also due to some lack of understanding on the part of the user community of the effects (and requirements) of multiprogramming.

Viewing multiprogramming

A motion picture film has been made from the output display of SPY showing a carefully selected job stream. This film has become an extremely valuable tutorial aid in demonstrating multiprogramming to the unsophisticated. To the more knowledgeable, it presents a real look at the actual occurrence of problems in a running computer, and the reasons that cause such a system to fall below the ideal maximum efficiency.

The film begins with a clear system with only SPY present, and then adds a single program and the system tasks necessary for that program. Slowly, the capability for more programs is added until a three job system is achieved. As jobs move in and out of execution, problems of core fragmentation, contention, unused core, and unused disks are observed.

The technique of using a movie, while not changing the amount of information presented, has allowed that information to be presented to persons outside the immediate environment, and has permitted the display of a controlled selection of jobs with a wide range of characteristics.

CONCLUSION

The SPY system has proven invaluable not only as a means of measuring the performance of a computing system but also as a tutorial aid in the theory of multiprogramming.

The statistics gathering capability, as expected, proved to be vital to the functions of "tuning" a system and validating hardware and software changes. Although the measurements retained for analysis in this first version of the system were rather coarse, they did bring to light some fundamental problems in the use of the operating system, and led to some action being taken towards system optimization.

The monitoring capability proved to be very valuable in giving many observers a true feel for how the operating system works. In addition, the availability of operator information was often useful and showed much potential, depending on the types of jobs being run at an installation, and thus the degree of operator control over jobs entering the system.

The two major design goals were clearly met: no modification whatsoever was made to OS/360 in any of the software development; and it was at all times clear that the effects of SPY upon the operation of the system are negligible. It utilizes less than 1 percent of the CPU time, occupies only 14K of core storage, and uses no secondary storage units.

One extension of the work presented in this paper will be the use of SPY to aid in more serious statistical studies on the operation of computing systems. Now that a means has been developed to extract salient

information from the system, a serious developmental effort can be expended towards expanding the system to produce data which can be used in more valid and meaningful statistical tests on system performance.

In conjunction with this work it is hoped that not only can the capabilities provided by the system be generalized, but also the applicability of the system can be extended. It is felt that this project has shown such a system to be an effective tool in any computing environment, and it is hoped that this experience can be expanded upon by the implementation of the system at other S/360 installations.

REFERENCES

1 J M GROCHOW
Real-time graphic display of time-sharing system operating characteristics
AFIPS Conference Proceedings Fall Joint Computer Conference Volume 35 pp 379-386 1969

2 G ESTRIN L KLEINROCK
Measures, models, and measurements for time-shared computer utilities
Proceedings ACM National Meeting pp 85-95 1967

3 G ESTRIN et al
SNUPER COMPUTER—A computer in instrumentation automation
AFIPS Conference Proceedings Spring Joint Computer Conference Volume 30 pp 645-656 1967

4 P CALINGAERT
System performance evaluation: Survey and appraisal
CACM Volume 10 No 1 pp 12-18 January 1967

5 R A ARBUCKLE
Computer analysis and thruput evaluation
Computers and Automation Volume 15 No 1 pp 12-15 January 1966

6 F D SCHULMAN
Hardware measurement device for IBM system/360 time-sharing evaluation
Proceedings ACM National Meeting pp 103-109 1967

7 J E SHEMER D W HEYING
Performance modeling and empirical measurements in a system designed for batch and time-sharing users
AFIPS Conference Proceedings Fall Joint Computer Conference Volume 35 pp 17-26 1969

8 T B PINKERTON
Performance modeling in a time-shared system
CACM Volume 12 No 11 pp 608-610 November 1969

9 H N CONTRELL A L ELLISON
Multiprogramming system performance measurement and analysis
AFIPS Conference Proceedings Spring Joint Computer Conference Volume 32 pp 213-221 1968

10 A L SCHERR
An anlysis of time-shared computer systems
Research Monograph No 36 MIT Press Cambridge Massachusetts 1967

11 T HASTINGS et al
Conversational system performance and measurement
DECUS Proceedings Fall Symposium pp 191-201 1969

12 W R DENISTON
SIPE: a TSS/360 software measurement technique
Procedings 24th National ACM Conference 1969

13 *IBM System/360 Operating System: System control blocks*
IBM Corporation White Plains NY Form C28-6628

14 *IBM System/360 Operating System: Concepts and facilities*
IBM Corporation White Plains NY Form C28-6535 1968

An efficient algorithm for optimum trajectory computation

by KENTON S. DAY

ESL Incorporated
Sunnyvale, California

INTRODUCTION

This paper describes a variation to the steepest-descent method for generating optimum trajectories. The steepest-descent approach to trajectory optimization was formulated by Kelley,[1,2] Bryson et al., [3–5] for numerically solving a variety of two-point boundary-value problems. The procedure is iterative, requiring repeated forward numerical integrations of the state differential equations and backward integrations of the adjoint equations. In many applications, however, convergence was slow; thus, several techniques for speeding convergence were devised.[6,7,8]

The computational algorithm presented in this paper requires that a perturbation of fixed size along one segment of the trajectory effect the same diminution of the penalty function as along any other segment. This essentially compensates for differences in sensitivity over various parts of the trajectory. An advantage of choosing the direction of uniform sensitivity is that *a priori* knowledge of the system behavior can be used effectively to speed convergence without increasing computational effort.

In the proposed method the nominal trajectory must be computed and stored, and the adjoint equations integrated backward. The uniform sensitivity direction is searched to determine the optimum step size corresponding to the greatest diminution of the penalty function. Only the unconstrained optimization problem is considered in this paper, though a general formulation of optimum programming requires terminal constraints on the state and inequality constraints on both the state and control variables all along the trajectory. (Constraints of this nature are often converted to unconstrained problems by judicious selection of a penalty function.[2,5,6])

The proposed method is compared to the conventional steepest-descent algorithm by an example of the interceptor missile control problem. In all cases studied, the proposed method converged immediately whereas conventional steepest-descent was oscillatory and slow. Neither method, however, can distinguish between local and global minima as with most other iterative algorithms. The usual procedure is to rerun the method with different starting conditions.

PROBLEM FORMULATION

The problem is to minimize

$$\varphi=\varphi[x(T), T] \tag{1}$$

subject to constraints

$$dx/dt=f(x, \alpha, t) \tag{2}$$

$$x(t_0)=x_0 \tag{3}$$

$$W[x(T), T]=0 \tag{4}$$

where $x=(x_1 \cdots x_n)'=$ state vector, $f=(f_1 \cdots f_n)'=$ velocity vector, φ and W are scalar functions, and $\alpha(t)$ is a scalar control variable. The terminal time T, fixed or free, is determined from (4). Constraints at $x(T)$ as well as on $\alpha(t)$ are assumed to be accounted for in (1). Furthermore, f is assumed to be differentiable in x and α.

STEEPEST-DESCENT

The method of steepest-descent requires guessing an initial control $\bar{\alpha}(t)$, and integrating (2) from initial state (3) until $W=0$ to obtain the first nominal trajectory $\bar{x}(t)$. A small control variation $\delta\alpha(t)$ causes a corresponding trajectory variation $\delta x(t)$ that must satisfy (to first order)

$$(d/dt)\delta x=F(t)\delta x+G(t)\delta\alpha \tag{5}$$

where

$$F(t)=\begin{bmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_n \\ \cdots\cdots & \cdots & \cdots\cdots \\ \partial f_n/\partial x_1 & \cdots & \partial f_n/\partial x_n \end{bmatrix} \qquad G(t)=\begin{bmatrix} \partial f_1/\partial\alpha \\ \cdot \\ \cdot \\ \cdot \\ \partial f_n/\partial\alpha \end{bmatrix}$$

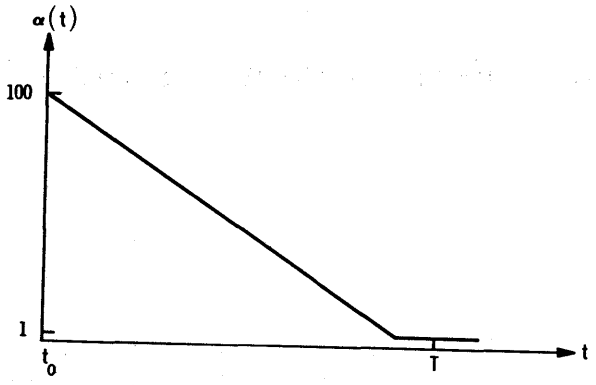


Figure 1—Control perturbations

and the partial derivatives are evaluated along $\bar{x}(t)$. Perturbations in φ and W at the terminal time due to $\delta\alpha(t)$ and dT are

$$d\varphi=[\partial\varphi(T)/\partial x]'\,dx(T)+(\partial\varphi/\partial T)\,dT$$

and

$$dW=0=[\partial W(T)/\partial x]'\,dx(T)+[\partial W(T)/\partial T]\,dT$$

where

$$\partial\varphi/\partial x=(\partial\varphi/\partial x_1\cdots\partial\varphi/\partial x_n)'$$

$$\partial W/\partial x=(\partial W/\partial x_1\cdots\partial W/\partial x_n)'$$

and

$$dx(T)=\delta x(T)+[dx(T)/dt]\,dT$$

These equations combine to give

$$d\varphi=\{\partial\varphi/\partial x-[(\dot{\varphi}/\dot{W})\,(\partial W/\partial x)]\}'_{t=T}\delta x(T)=k'\delta x(T) \tag{6}$$

where k is a vector constant (defined explicitly) for the nominal $\bar{x}(t)$, and

$$\dot{\varphi}=\partial\varphi/\partial T+[(\partial\varphi/\partial x)'(dx/dt)]_{t=T}$$

$$\dot{W}=\partial W/\partial T+[(\partial W/\partial x)'(dx/dt)]_{t=T}$$

In place of (6), the penalty function variation can be expressed as an integral

$$d\varphi=\int_{t_0}^{T}p'(t)G(t)\delta\alpha(t)\,dt$$

where $p(t)=(p_1\cdots p_n)'$ is defined by the differential equation

$$dp/dt=-F'(t)p$$

with boundary conditions

$$p(T)=\partial\varphi(T)/\partial x-\{(\dot{\varphi}/\dot{W})[\partial W(T)/\partial x]\}$$

The usual procedure is to choose $\delta\alpha(t)$ so as to cause the greatest diminution in φ for a fixed $\|\delta\alpha\|$ defined by

$$\|\delta\alpha\|=\int_{t_0}^{T}\delta\alpha^2(t)\,dt \tag{7}$$

The gradient direction is $\delta\alpha(t)=-p'(t)G(t)$ and the optimum step size λ minimizes $\varphi[\bar{\alpha}-\lambda p'G]$ for the present iteration. Thus, the best control for the next iteration is

$$\alpha_{\text{new}}(t)=\bar{\alpha}(t)+\lambda\delta\alpha(t)$$

UNIFORM SENSITIVITY

Uniform sensitivity varies from the method just described in the choice of $\delta\alpha$. Instead of (7), the norm is taken as

$$\|\delta\alpha\|=\int_{t_0}^{T}a(t)\delta\alpha^2(t)\,dt\qquad a(t)>0 \tag{8}$$

and the uniform sensitivity direction is

$$\delta\alpha(t)=-p'(t)G(t)/a(t) \tag{9}$$

It remains to determine $a(t)$ such that the sensitivity of (1) to $\delta\alpha(t)$ is uniform over the entire trajectory.

Assuming that a nominal control $\bar{\alpha}(t)$ and trajectory $\bar{x}(t)$ on $[t_0, T]$ are available, the time interval $[t_0, T]$ is partitioned into small increments of width Δt. Any small control perturbation $\delta\alpha_\tau$ at time τ, $t_0\leq\tau\leq T$, of the type shown in Figure 1, with amplitude $A(\tau)$, duration Δt, and fixed norm $\|\delta\alpha_\tau\|$ is required to effect an equal change in φ as a control variation $\delta\alpha_{\tau'}$ at τ' of the same size (i.e., $\|\delta\alpha_{\tau'}\|=\|\delta\alpha_\tau\|$ implies $d\varphi_\tau=d\varphi_{\tau'}$). The effect of $\delta\alpha_\tau$ on $x(T)$ is

$$\delta x(T)=\int_{\tau}^{\tau+\Delta t}\Phi(T,s)G(s)A(\tau)\,ds$$

where Φ is the transition matrix of (5). The norm of $\delta\alpha_\tau$, from Figure 1 and (8) is

$$\|\delta\alpha_\tau\|=\int_{\tau}^{\tau+\Delta t}a(t)A^2(\tau)\,dt=A^2(\tau)\int_{\tau}^{\tau+\Delta t}a(t)\,dt \tag{10}$$

From (6), φ is uniformly sensitive if $k'\delta x(T)$ is constant, so for $d\varphi_\tau=d\varphi_{\tau'}$, it follows that

$$d\varphi_\tau=\left[k'\int_{\tau}^{\tau+\Delta t}\Phi(T,s)G(s)\,ds\right]A(\tau)=\text{constant} \tag{11}$$

Eliminating $A(\tau)$ from (10) and (11) gives

$$\int_{\tau}^{\tau+\Delta t} a(t)\,dt = (\,||\,\delta\alpha_\tau\,||\,/d\varphi_\tau^2) \times \left[k' \int_{\tau}^{\tau+\Delta t} \Phi(T, s)G(s)\,ds\right]^2 \quad (12)$$

The original steepest-descent gradient direction $-p'(t)G(t)$ is modified by $a(t)$ (see (9)) at $\tau=t$ to produce uniform sensitivity in the penalty function. The optimum step size λ, as before, minimizes $\varphi(\bar{\alpha}-\lambda p'(t)G(t)/a(t))$. The method always generates directions of descent whenever steepest-descent does. Although (12) is an integral equation, the mean-value theorem can be invoked for both sides of the equation giving

$$a(\xi) = (\,||\,\delta\alpha_\tau\,||\,/d\varphi_\tau^2)[k'\Phi(T, \zeta)G(\zeta)]^2\,\Delta t$$

for $\tau \leq \xi$, $\zeta \leq \tau+\Delta t$. It is irrelevant to let $\Delta t \to 0$ since control perturbations vanish in the limiting case. However, the right hand side of (12) is smooth, and in keeping within the framework of practicality, it should suffice for small but finite Δt to substitute τ for ξ and ζ, i.e.,

$$a(\tau) \approx C[k'\Phi(T, \tau)G(\tau)]^2 \quad (13)$$

where C is a positive constant and τ is any time on $[t_0, T]$.

Equation (13) calls for computing a transition matrix, but this is not always necessary for the iterative process to be efficient. It may suffice to know gross behavior of $a(t)$ on $[t_0, T]$. Trends can often be obtained by treating $F(t)$ and $G(t)$ as constants or by estimating $a(t)$ *a priori* as in the following example.

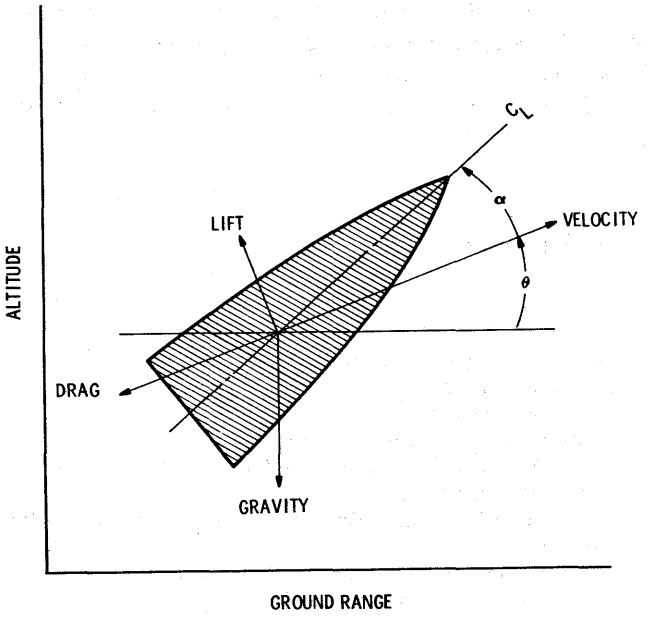


Figure 2—Vehicle nomenclautre and coordinate system

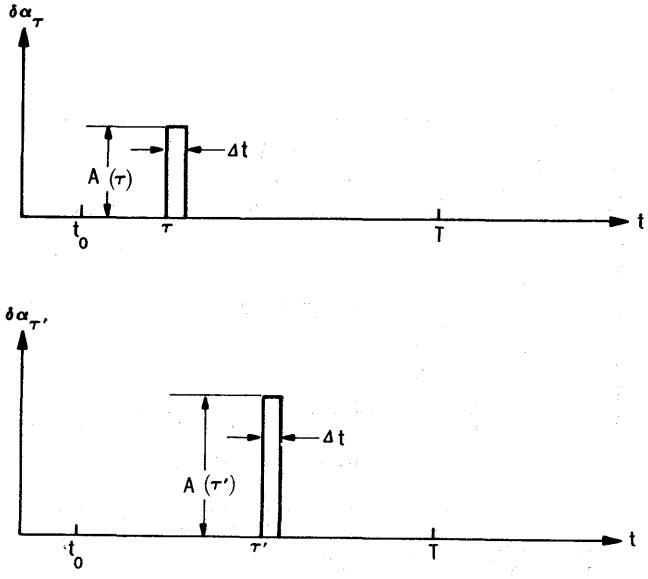


Figure 3—Approximation to equation 12 for the interceptor missile example

Example: the interceptor missile

This example considers optimal guidance of an interceptor missile to a fixed space point in minimum time with the additional terminal constraint that the intercept occur with a specified flight path angle. The dynamic equations for the nomenclature of Figure 2 are

$$dr/dt = V\cos\theta$$

$$dh/dt = V\sin\theta$$

$$dV/dt = [F(t)/m(t)]\cos\alpha - g\sin\theta - [\rho(h)V^2C_D(\alpha, M)S/2m(t)]$$

$$d\theta/dt = [F(t)/m(t)](\sin\alpha)/V - (g/V)\cos\theta + [\rho(h)V^2C_L(\alpha, M)S/2m(t)V]$$

where r=range, h=altitude, V=velocity, θ=flight path angle, α=angle of attack (control variable), g=gravity, F=thrust, m=mass, ρ=air density, M=mach number, and S=reference area. The stopping condition and penalty function are

$$W = \tfrac{1}{2}[r(T) - r_I]^2 = 0$$

$$\varphi = T + \tfrac{1}{2}R_1[h(T) - h_I]^2 + \tfrac{1}{2}R_2[\theta(T) - \theta_I]^2$$

where $(r_I h_I V \theta_I)'$ defines the terminal state vector

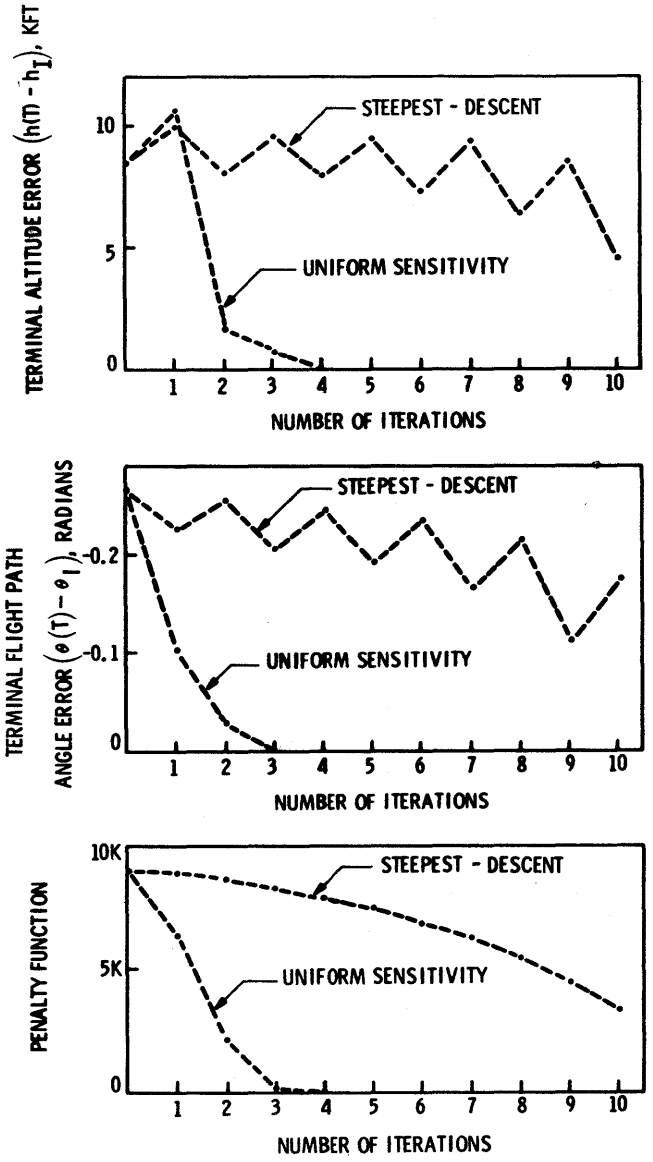


Figure 4—Illustration of convergence for the interceptor missile example

[$V(T)$ is unspecified], and $R_1 = 10^{-4}$ and $R_2 = 10^4$ are constant weighting factors selected to equalize each term in φ when equalities in the convergence criteria are satisfied. Iteration terminated when the following conditions of convergence were simultaneously achieved: $r(T) = r_I$, $|h(T) - h_I| \leq 100$ feet,

$$|\theta(T) - \theta_I| \leq 0.02 \text{ rad},$$

and $\Delta t \leq 0.01$ seconds = difference between successive iterations.

The function for $a(t)$ shown in Figure 3, was estimated from the linearized state equations and used as an approximation to (13). This type of result for $a(t)$ might be expected since a control variation $\delta\alpha$ early in flight, when the velocity and atmospheric density are high, has much greater effect on the terminal state (and consequently φ) than the same $\delta\alpha$ later in flight at a higher altitude. The curves in Figure 4 illustrate convergence for steepest-descent and uniform sensitivity for an intercept at 100,000 feet range, 20,000 feet altitude, and zero flight path angle, and are representative of convergence behavior for other space point intercepts that were studied.

CONCLUSION

A practical extension to the method of steepest-descent has been developed which seems to enhance convergence properties without significantly increasing computational effort. Based on the requirement that the penalty function be equally sensitive to a fixed control perturbation along any segment of the trajectory, the direction of uniform sensitivity was determined. The uniform sensitivity direction essentially represents a modified steepest-descent gradient. Therefore, the basic steepest-descent algorithm is unaltered except at the stage where the control perturbation is computed.

The proposed method always generates directions of descent whenever steepest-descent does, and has convergence properties superior to the latter—at least when applied to optimal interceptor missile control. In fact, the development evolved out of difficulty in forcing steepest-descent convergence in application to optimal missile guidance. Figure 4 typifies the level of improvements in convergence that were realized for two types of missiles and over a wide range of atmospheric intercepts.

REFERENCES

1 H J KELLEY
Gradient theory of optimal flight paths
ARS J Volume 30 pp 947-953 1960

2 H J KELLEY
Method of gradients
Optimization Techniques G Leitmann Ed Academic Press New York New York Chapter 6 1962

3 A E BRYSON et al
Determination of lift or drag programs that minimize re-entry heating
J Aerospace Sci Volume 29 pp 420-430 1962

4 A E BRYSON W F DENHAM
A steepest-ascent method for solving optimum programming problems
J Appl Mec Volume 29 pp 247-257 1962

5 W F DENHAM A E BRYSON
Optimal programming problems with inequality constraints—II: Solution by steepest-ascent
AIAA J Volume 2 pp 25-34 1964

6 L S LASDON et al
The conjugate gradient method for optimal control problems
IEEE Trans on Automatic Control Volume AC-12 pp 132–138 1967

7 W E WILLIAMSON W T FOWLER
A segmented weighting scheme for steepest-ascent optimization
AIAA J Volume 6 pp 976-977 1968

8 R ROSENBAUM
Convergence technique for the steepest-descent method of trajectory optimization
AIAA J Volume 1 pp 1703-1705 1963

Hybrid computer solutions for optimal control of time-varying systems with parameter uncertainties*

by W. TRAUTWEIN and C. L. CONNOR

Lockheed Missiles & Space Company
Huntsville, Alabama

SUMMARY

A hybrid computing scheme is described which economically solves optimal regulator problems for time-varying systems. The variational problem of determining optimal time-varying controller gain schedules is reduced to a sequence of standard one-dimensional parameter searches and high-speed simulations by restricting the class of optimal gain schedules to be piecewise linear.

A major shortcoming of most optimization techniques—the high sensitivity to parameter uncertainties—is mitigated by making the performance index dependent upon two or more different simulations. Choosing highly adverse operating conditions for performance evaluation largely reduces the sensitivity of the optimized system to off-nominal conditions. This is illustrated by an example, the design of a load-relief attitude controller for large launch vehicles. Other practical features of the hybrid optimization include the use of free-form, non-quadratic performance measures to specify design goals in the most direct manner with only minor mathematical constraints concerning their functional form.

INTRODUCTION

Hybrid computers have long found wide use in parameter optimization of time-invariant dynamic systems. Fast-time, repetitive analog simulation of the system dynamics is combined with a functional minimization scheme programmed digitally. A performance index, formulated in accordance with the design goals, is minimized by iteratively updating the adjustable parameters between subsequent analog runs.[1]

A basic advantage of hybrid optimization over linear optimal control theory is that performance measures can be freely selected to best reflect the design objectives, whereas the quadratic criteria required by linear control theory are in general difficult to relate to the design goals. System nonlinearities and constraints imposed upon certain state variables or control parameters can be accounted for in the hybrid approach without difficulty.

These practical features made it desirable to solve another important engineering problem, optimal controller design for time-varying systems, by a similar technique. Optimal controller adjustments in this case become functions of time which in general require variational techniques for their solution. For regulator design without specified terminal conditions the variational problem can readily be reduced to a sequence of parameter optimizations if the optimal gain schedules are restricted to piecewise linear functions of time. In another section the basic approach will be described for completely defined systems. An extension of the hybrid technique to drastically reduce the sensitivity of the optimized system to parameter uncertainties or other off-nominal conditions is given later in this paper. A more detailed discussion of the optimization method and its application to booster load relief controller design is given in Reference 2.

OPTIMAL REGULATOR DESIGN FOR TIME-VARYING SYSTEMS UNDER NOMINAL CONDITIONS

Formulation of the control problem

The dynamics of the system (plant) be described by the first-order vector equation

$$\dot{\bar{x}} = \bar{f}(\bar{x}, \bar{u}, t) \qquad (1)$$

* Work supported by NASA-MSFC Contract No. NAS8-30515, Mod II, Task 1.

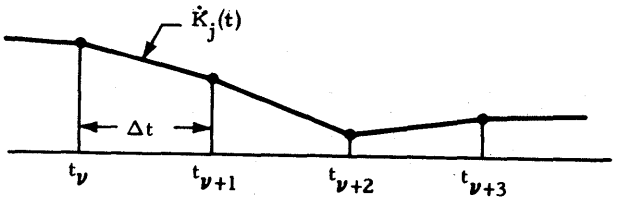


Figure 1—Assumed polygonal form of optimal gain schedules

where $\bar{x}$ is the n-dimensional state vector, $\bar{f}$ an n-dimensional functional vector, $\bar{u}$ an m-dimensional control vector $(m \leq n)$ and $(\dot{}) = d/dt$.

The control law be dictated by available sensors or other limitations in cost and complexity. For this discussion we assume a linear control law

$$\bar{u}(t) = K(t)\bar{x}(t) \tag{2}$$

where K is an $m \times n$ gain matrix. The design problem then is to determine the gain matrix $K(t)$ so that the performance measure

$$J = J(\bar{x}, \bar{u}, t) \tag{3}$$

is minimized. J is a positive definite function of otherwise free form, selected to best reflect the design objectives.

Restriction to piecewise linear gain schedules

The computational load can be greatly reduced with little or no loss in optimality of the solutions if the mn gain schedules of $K(t)$ are assumed to be piecewise linear functions of time as shown in Figure 1. Then the optimization problem is reduced to a parameter search

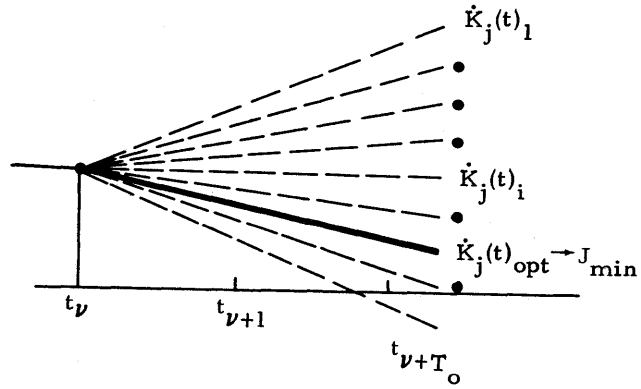


Figure 2—During optimization cycle at time t_ν, a search is performed in K_j parameter space of the mn gain slopes for optimum performance

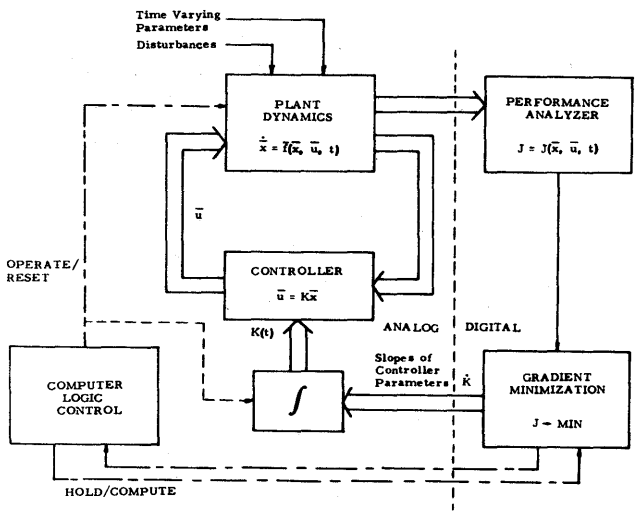


Figure 3—Basic control system optimization scheme. Complete plant and control system dynamics are simulated repetitively on analog console of hybrid computer. Performance is analyzed after simulation in digital computer and optimized by iterative changes in slopes of controller gain schedules.

for the mn gain slopes $\dot{K}_j$ of the gain matrix $K(t)$ as shown in Figure 2 which can be carried out based on standard hybrid techniques. In this regulator problem the performance index does not contain the terminal state. Thus, forward integrations are sufficient to determine the optimum. The basic computing scheme is shown in Figure 3. The analog simulation serves a dual purpose. Performance is repetitively evaluated during an optimization cycle in fast time. Once the optimum set of gain slopes $\dot{K}_j(j=1, 2, \ldots mn)$ is determined the optimum parameters are used during a real time simulation from one update time (t_ν) to the next $(t_{\nu+1})$ using the same analog circuit (Figures 4 and 5).

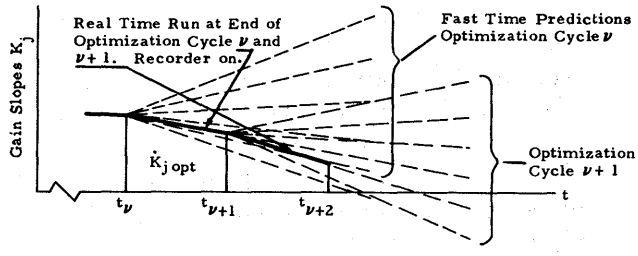


Figure 4—Series of fast time predictions at update time t_ν leads to optimum gain slopes $\dot{K}_{j\,opt}(j=1 \ldots mn)$. Subsequent real time simulation uses $\dot{K}_{j\,opt}$ from t_ν to next update time $t_{\nu+1}$

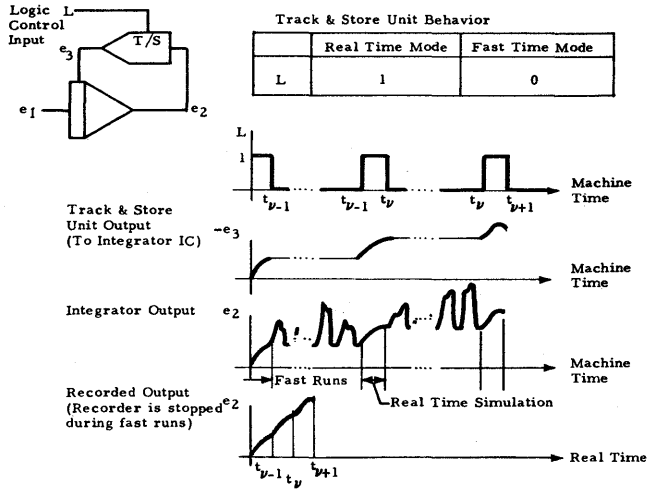


Figure 5—Simple analog circuit for real time simulation and fast time predictions using same integrators

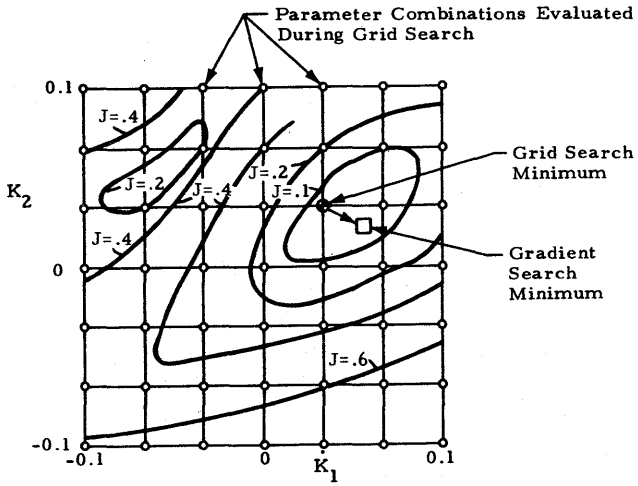


Figure 6—Parameter optimization performed in two phases: (1) Systematic grid search (o) for complete survey of parameter space; Grid point of minimum J(○) serves as starting point for (2) gradient search which locates the minimum more precisely (□). From grid search contour plots (Lines of $J = \text{Const}$) can be displayed at CRT for better insight into J-topology.

Minimization method

In order to distinguish between local minima and absolute minima the search in the $\dot{K}_j$ parameter space is performed in two phases:

1. Systematic Grid Search
   All possible parameter combinations within a region of specified limits and a grid of specified fineness are evaluated for their performance J. Such a complete survey is feasible as long as the parameter space is of low dimension as in present applications (Figure 6).
2. Gradient Search
   The Fletcher-Powell-Davidon gradient method[3] uses the grid search minimum as a starting point to precisely locate the minimum.

Example: Booster load relief controller design

The practical features of the optimization scheme are best illustrated using the following example. The peak bending loads at two critical stations, x_1 and x_2 (Figure 7) of a Saturn V launch vehicle shall be minimized during powered first stage flight while the vehicle is subjected to severe winds with a superimposed gust (Figure 8, top strip) and to an engineout failure at the time of maximum dynamic pressure.

The major variables are:

α	vehicle angle of attack
β	control engines gimbal angle
η	generalized displacement of bending mode at nose of vehicle
ϕ	vehicle pitch attitude error
χ	commanded pitch attitude relative to vertical at time of launch
z	drift, normal to the reference trajectory
ξ	slosh mass displacement, normal to reference trajectory
$Y(x)$, $Y'(x)$	bending mode normalized amplitude and slope at station x

Disturbance terms due to engine failure:

$\ddot{\phi}_F$	pitch acceleration due to engine failure
$\ddot{z}_F$	lateral acceleration due to engine failure
$\ddot{\eta}_F$	bending mode acceleration due to engine failure

Control law

The attitude controller to be optimally adjusted has feedback of attitude error, error rate and lateral acceleration:

$$u \equiv \beta = a_0(t)\phi_s + a_1(t)\theta_s + g_2(t)\ddot{r}_s \qquad (4)$$

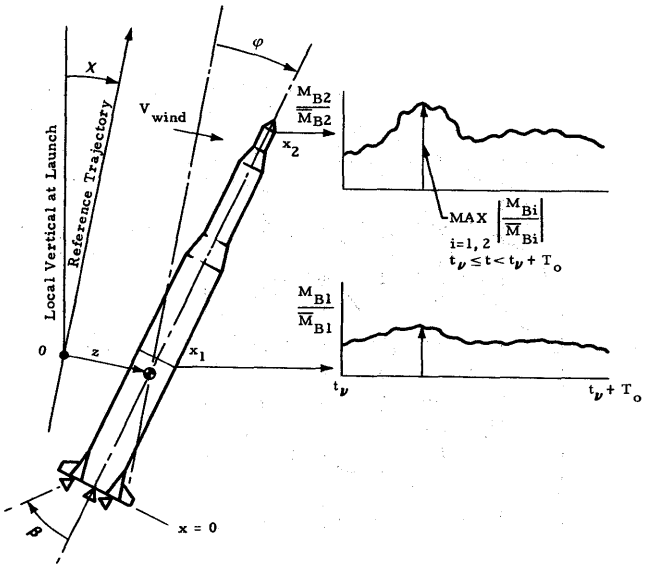


Figure 7—Minimax load relief performance criterion to minimize peak bending loads at two critical locations, x_1, x_2:

$$J=\max_{\substack{i=1,2\\ t_\nu\le t<t_\nu+T_0}}|M_{Bi}(t)/\bar{M}_{Bi}|+q\int_{t_\nu}^{t_\nu+T_0}\theta^2_{RG}(t)\,dt\rightarrow\min \qquad (6)$$

Normalized Peak Bending Loads

Mean Square of Rate Gyro Output to Ensure Trajectory Stability

System equations

The vehicle equations of motion as simulated on the analog console are:

Rotation:

$$\ddot{\phi}=-c_1\alpha-c_2(1-0.25\delta)\beta+\delta\ddot{\phi}_F+G_r\eta+N_r\ddot{\xi}+\bar{N}_r\ddot{\xi}$$

Translation:

$$\ddot{z}=K_1\phi+K_2\alpha+K_3(1-0.25\delta)\beta+\delta\ddot{z}_F+G_t\eta+\bar{N}_t\ddot{\xi}$$

Bending Mode:

$$\ddot{\eta}=-\bar{G}_b\dot{\eta}-G_b\eta+B_b(1-0.25\delta)\beta+\delta\ddot{\eta}_F+\bar{N}_{b1}\ddot{\xi}$$

Sloshing:

$$\ddot{\xi}=-N\dot{\xi}-N\xi-\ddot{z}+\bar{U}_1\ddot{\phi}+K_1\phi$$

Position Gyro Output:

$$\phi_i=\phi+Y'(x_{PG})\eta$$

Rate Gyro Output:

$$\theta_i=\dot{\phi}+Y'(x_{RG})\dot{\eta}$$

Accelerometer Output:

$$\ddot{\tau}_i=\ddot{z}+A_1\ddot{\phi}-A_2\phi-A_3\ddot{\eta}+A_4\eta$$

Gimbaled Engine Dynamics:

$$\dot{\beta}=N_E(\beta_c-\beta)\quad|\beta|\le 5.15^\circ$$

Shaping Filters:

Attitude Error Filter: $\phi_s=\phi_i(P_0s+1)/P_1s+1)$

Rate Error Filter: $\theta_s=\theta_iQ_0/(s^2+Q_1s+Q_0)$

Accelerometer Filter: $\ddot{\tau}_s=\ddot{\tau}_iR_0/(s^2+R_1s+R_0)$

Angular Relationship:

$$\alpha=\phi-\dot{z}/V+\alpha_w$$

$$\alpha_w=\tan^{-1}(V_w\cos\chi/V-V_w\sin\chi)$$

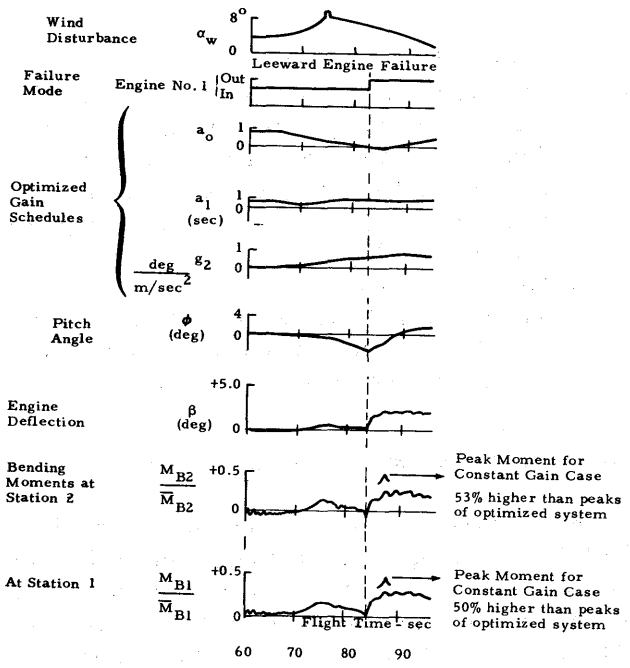


Figure 8—Typical Saturn V S-IC optimization results. Three gain schedules are optimally adjusted to minimize the peak bending loads among two stations (1541 and 3764 inches) for the disturbance and failure history of the top charts. Peak loads are substantially reduced compared with nominal Saturn V performance (without accelerometer feedback). Weighting factor $q=0.05$; floating optimization time interval $T_0=20$ sec in performance criterion (6)

Engine Failure Switching Function:

$$\delta = \begin{cases} +1 \text{windward engine out} \\ 0 \text{ no failure} \\ -1 \text{leeward engine out} \end{cases}$$

Only one bending mode and one slosh mode was included in this study. The bending moment at station x_i is

$$M_{Bi} = M'_{\alpha i}\alpha + M'_{\beta i}\beta + M'_{\ddot{\eta}}\ddot{\eta}$$

Complete numerical data for all coefficients used in the simulation are compiled in Reference 4.

Selection of performance index

In earlier studies[4] quadratic performance criteria such as

$$J = \int_{t_p}^{t_p+T_0} \{q_1 M^2_{B1} + q_2 M^2_{B2}\}\, dt \qquad (5)$$

were used. They allow a straightforward physical interpretation and at the same time can still be loosely related to linear optimal control theory. Neglecting external disturbances ($\alpha_w \approx 0$), Equation (5) can be rewritten in the more familiar form

$$J = \int_{t_p}^{t_p+T_0} \{q_3 x^T a a^T x + (u + q_4 a^T x)^2\}\, dt$$

where a is an n-dimensional coefficient vector, q_3, q_4 are constants depending upon q_1, q_2, M_α' and M_β' and superscript T denotes transpose.

The results from optimizations where Equation (5) was minimized were disappointing insofar as peak bending loads were reduced by a few percent only, whereas the major reductions were in the RMS values of the bending loads.

Since peak loads are of major concern to the designer, a more direct approach was made to reduce peak loads by using the minimax criterion (6) of Figure 7. During each run bending load peaks at both critical stations were sampled and compared. Only the greater of the two peaks was included in J. This peak amplitude normalized with respect to the structural limit $\bar{M}_B$ was the major term in J. The only additional term, the mean square of measured error rate, was included to ensure smooth time histories and trajectory stability. This performance criterion reduced the number of weighting factors to be empirically adjusted to one, whereas n such factors must be selected in linear optimal control theory for an nth order system.

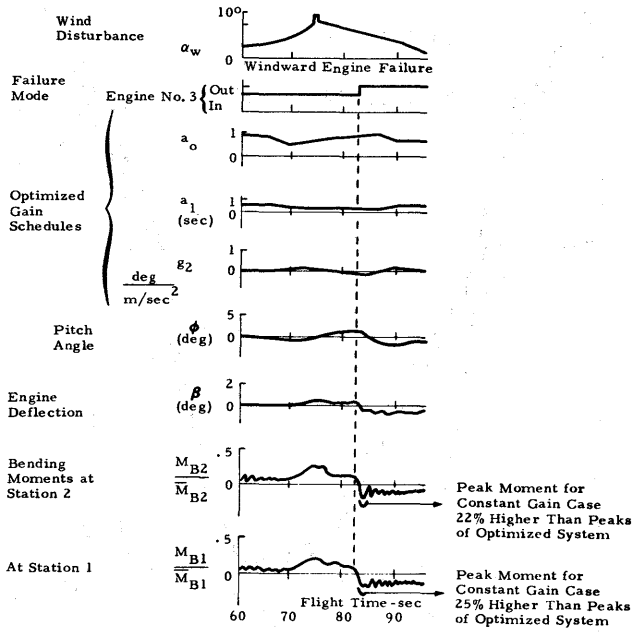


Figure 9—Saturn V S-IC optimization of Figure 8 repeated for assumed *windward* engine failure under otherwise identical flight conditions. Optimal gain schedules are strongly dependent upon assumed failure condition

Results

A typical optimization result is shown in Figure 8. Drastic reductions in bending moment peaks result from the minimax criterion compared with the constant gain case. It should be noted, however, that perfect predictions 20 seconds ahead were used in the optimization including the anticipated failure.

In Figure 9 the results of a similar case are shown. All flight conditions are identical to the previous case except for the failure mode: a leeward engine fails in Figure 8, a windward engine in Figure 9. Again, peak bending loads are substantially reduced in magnitude compared with a case with nominal constant adjustments of the controller. However, two of the three optimal gain schedules ($a_0(t)$ and $g_2(t)$) differ drastically for the two failure modes. In view of the lack of any *a priori* knowledge about time and location of a possible engine failure no useful information can therefore be gained from the two optimizations concerning load relief controller design. This basic shortcoming of all strictly deterministic optimization schemes must be relieved before the method can be applied to practical engineering design problems characterized by parameter or failure uncertainties.

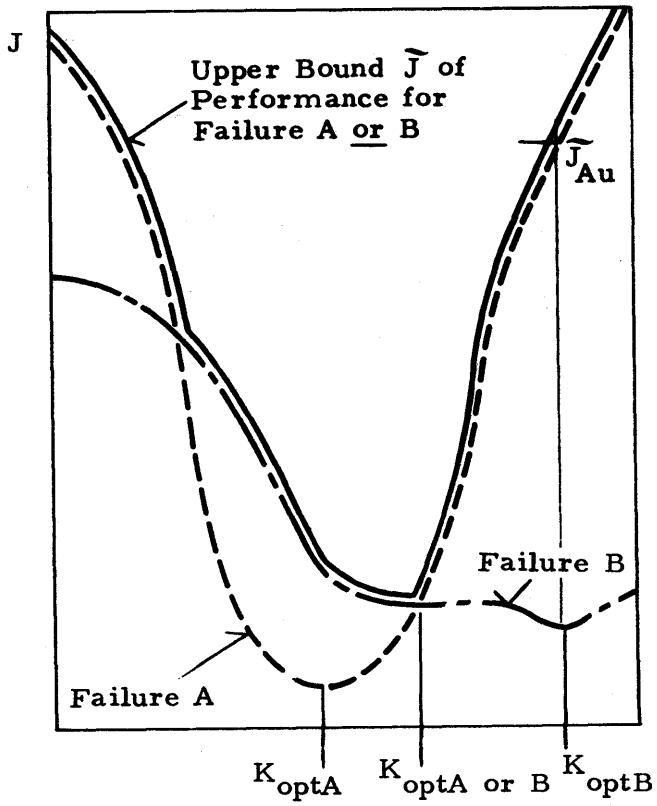


Figure 10—Optimum adjustment of scalar control parameter K considering possible occurrence of failure A *or* B

OPTIMAL REGULATOR DESIGN INSENSITIVE TO FAILURE UNCERTAINTIES

Previous work to reduce parameter sensitivity has centered around inclusion of sensitivity terms $\partial J_0/\partial K$ in the performance index to be minimized, where J_0 denotes performance under nominal conditions and K is the uncertain parameter vector.[5] Substantial additional computational load would arise if such an approach were implemented on the hybrid computer. Moreover, in the case of possible failures the uncertain parameters may vary discontinuously from one discrete value to another like the engine failure switching function in the preceding example:

$$\delta = \begin{cases} 1 \text{ for Failure } A \\ 0 \text{ for Nominal Case} \\ -1 \text{ for Failure } B \end{cases}$$

Another approach is therefore chosen: The hybrid optimization method is extended to account for such failure uncertainties even if no partial derivatives exist. Consider the case of two possible failure conditions, A or B. The performance index evaluated for each failure may be of the form of Figure 10. Neither K_{optA} nor K_{optB} would be optimal in view of the uncertainty concerning the failure mode. Performance might be unacceptably poor at the level $\tilde{J}_{Au}$ if Failure A occurred and the control parameter were adjusted at the optimum for Failure B. The best tradeoff in view of the failure uncertainty is the minimum of the upper bound of J_A and J_B (solid line in Figure 10). In the example of Figure 10 this optimum which is the least sensitive to the type of failure lies at the lower intersection of the two curves.

Extension of the optimum-seeking computing scheme

The most direct way to locate the minimum of the upper performance bound is to simulate all possible failure modes for a given set of control parameters in order to determine the upper bound $\tilde{J}$. A gradient dependent minimization technique can again be applied to seek the minimum. One might expect convergence difficulties around the corners of these $\tilde{J}$-functions. However, only minor modifications were necessary to the basic Fletcher-Powell-Davidon gradient scheme and to the preceding grid search to locate corner-type minima. The changes included a relaxing of the gradient convergence test ($|\nabla J| \leq \Sigma$, where Σ is a small specified number). If all other convergence tests are passed, then Σ is doubled in subsequent iterations. In

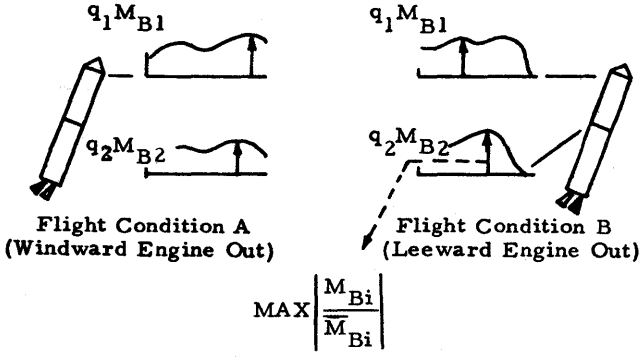


Figure 11—Generalized load relief performance criterion to minimize upper performance bound $\tilde{J}$ for two possible operating conditions

$$\tilde{J} = \max|M_{Bi}/\bar{M}_{Bi}| + \int_{t_\nu}^{t_\nu+T_0} \theta^2_{RG}(t)\, dt \rightarrow \min$$

Case A, Case B
$i = 1, 2$
$t_\nu < t < t_\nu + T_0$

order to keep the number of analog simulations low, only one failure case associated with the local upper bound is simulated for approximate gradient computations based on $\tilde{J}(K+\Delta K)$. During the grid search much computing time can be saved by checking if the upper performance bound $\tilde{J}$ is much larger than the performance index J of the less critical case. Then evaluation of only one failure case is necessary for the neighboring grid points.

Application to the booster load relief problem

In its extended form the optimization technique is ideally suited to minimize the effects of possible failures in the booster attitude control problem. The design goal is to minimize peak bending loads for the worst of several possible failure conditions. To this end the two most adverse flight and failure conditions are simulated to determine the upper performance bound $\tilde{J}$ of both cases for each set of parameters as shown in Figure 11. The performance index $\tilde{J}$, now a function of both adverse cases, is then minimized following much the same scheme as in the basic approach.

In Figure 12 results obtained by considering a single failure mode only (solid lines) as in Figures 8 and 9 are compared with results gained from the extended optimization scheme which considers both adverse failure conditions to minimize the upper performance bound. The former gain schedules (solid lines) are closely tuned to the specific flight and failure condition and therefore, differ drastically for the two adverse failure conditions and cannot be used for controller design.

Simultaneous consideration of both adverse failure modes (dotted lines) leads to a single set of controller

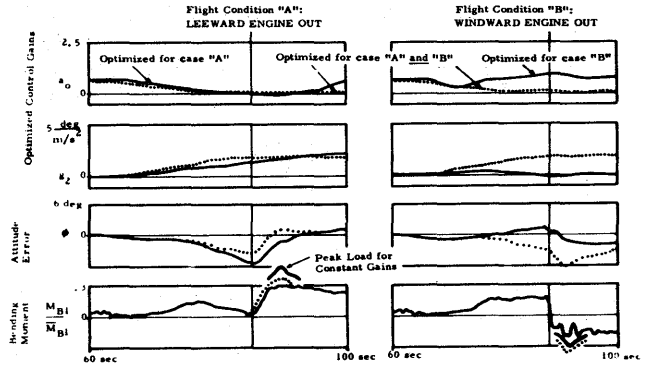


Figure 12—Deterministic optimization for one failure case only (solid lines) compared with optimization subject to failure uncertainty (dotted lines) with consideration of two possible failure modes

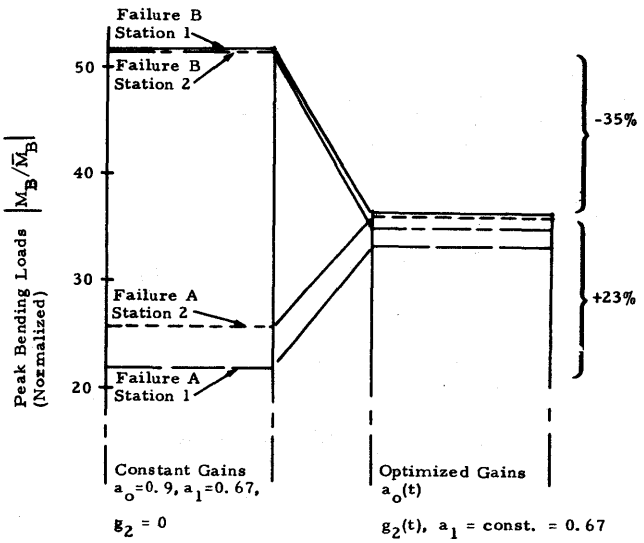


Figure 13—Peak bending loads for Saturn V S-IC powered flight compared for constant gain case and for case optimized for failure of leeward *or* windward engine. Wind profile as shown in Figure 8; Assumed engine failure time 76 sec

gain schedules. The reduction in peak load is smaller than before for Condition A, whereas the milder Condition B even shows some increase in bending load. A closer inspection of the resulting loads in Figure 13 reveals that an ideal tradeoff between failure effects A and B was achieved. All peak loads are brought to a common level of about 33 percent of the structural limit $\bar{M}_B$, whereas the constant gain controller adjustment exhibits loads up to 52 percent of $\bar{M}_B$.

COMPUTATIONAL ASPECTS

Experiments were made on the EAI 8900 computer to determine if acceptable accuracy and repeatability is obtained for a time scale of 1000 times real time. Repeatability was found to be within 2 percent for the time scale 1000 and within 0.5 percent for slower time scales using a circuit and interface similar to the optimization scheme. Two percent repeatability was considered sufficient. Therefore, a time scale of 1000 times real time is used during fast time predictions. In the booster control problem the optimization of two gain schedules takes about 5.5 minutes hybrid time.

Up to 40 analog simulations per second are presently achieved for analog run times of typically 20 milliseconds (Figure 14). Using nine grid points for each parameter during the grid search locates the minimum sufficiently close in the example presented. After 3

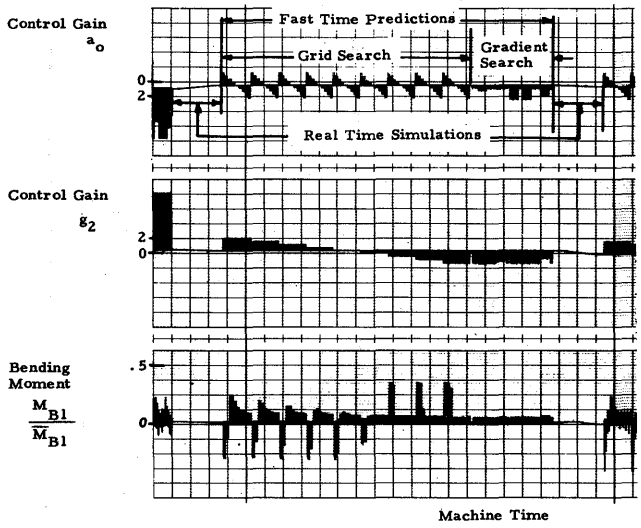


Figure 14—Typical computing speed is 20 to 40 analog simulations per second on EAI 8900 computer including A/D transfer of performance index, D/A transfer of new parameters between simulations. Time scale used during fast runs: 1000 times real time; Typical analog run time 20 millisec. per run

to 5 iterations of the following Fletcher-Powell-Davidon gradient search the minimum is found with sufficient accuracy.

ACKNOWLEDGMENT

The authors wish to acknowledge the contributions of Messrs. Roger Lin (now EAI, NASA Ames) and S. Lo in hybrid programming and computations.

REFERENCES

1 G A BEKEY W J KARPLUS
Hybrid computation
John Wiley & Sons Inc Section 9 New York 1968

2 C L CONNOR W TRAUTWEIN
Control system optimization for Saturn V launch vehicles using gradient techniques
Final Report Contract NAS8-30515 Mod II Task 1
LMSC/HREC D162122-I & III Huntsville Alabama
February 1970

3 R FLETCHER M J D POWELL
A rapidly convergent descent method for minimization
Computer J Vol 6 1963

4 W TRAUTWEIN J G TUCK
Control system optimization for Saturn V launch vehicles using gradient techniques
Final Report Contract NAS8-21335 LMSC/HREC
A791836 Lockheed Missiles & Space Company Huntsville Alabama October 1968

5 P DORATO
On sensitivity in optimal control systems
IEEE Trans Vol AC-8 pp 256-257 July 1963

The role of computer specialists in contracting for computers—An interdisciplinary effort

by ROY N. FREED

Widett & Kruger
Boston, Massachusetts

INTRODUCTION

The complexity of computer-communications technology requires computer specialist involvement in the negotiation and structuring of contracts relating to computer systems if many business arrangements important to the parties are to work out smoothly and successfully. Numerous businessmen and lawyers are not yet sufficiently sophisticated concerning the unique qualities of computers and their uses to set up viable contractual relationships respecting the more complex uses of the technology without substantial guidance from persons acquainted with the vulnerabilities of users and suppliers in particular types of transactions. As users and suppliers now become aware, somewhat belatedly, of the pitfalls in contracting in that area of technology, computer specialists have challenging opportunities for interdisciplinary professional involvement with lawyers. This paper suggests means for making such involvement as fruitful as possible for all parties concerned.

POOR CONTRACTING PRACTICES HAVE PREVAILED

The inadequacy of contracting for computer systems and computer use and the adverse consequences of that inadequacy to user and supplier alike finally are being recognized, if the response to the author's public discussion of the subject is any criterion.[1] Warnings based on sheer professional judgment went unheeded for a number of years. Only when poorly structured transactions started to result in litigation and in large damage awards against major companies did users and suppliers commence to explore ways to achieve sounder contracts. The primary stimulus probably was the verdict for almost $500,000 against IBM's Service Bureau Corporation in April, 1969, for damages found to have resulted from incorrect representations made so carelessly as to amount to legal fraud.*

Computer contracts require special attention for a number of reasons. In many situations, the subject matter is more complex than that normally encountered in purchase or lease transactions. At the very least, for example, the operations of an entity suddenly might become entirely dependent upon the sustained functioning of a machine system. This would be the case whether an in-house system or an external time-sharing service is used. Previously, interruptions of equal magnitude were possible only in the most unusual of circumstances, such as a strike or equally dire development that immobilized large quantities of people. Consider, as a further and more specific example, the use of an external time-sharing service to conduct business record-keeping and information processing. By taking that step, a fundamental segment of a business is carved out and turned over completely to an outsider. Controls that formerly were enjoyed directly now must be maintained vicariously, through the vehicle of contracts, thereby introducing the conflicting interests of a supplier and the need to resort to a legal tribunal for the resolution of disputes. Means must be provided, in such an arrangement, to insure that the supplier will respond adequately to largely unpredictable future needs for increased volume and variety of services, will handle catastrophes with genuine concern, and will protect critical information from unauthorized disclosure.

Frequently, the technology insinuates a particular supplier more deeply and irreversibly into the operations of the user than do other machines or techniques. The time and expense entailed in introducing a computer system often bar switches to other suppliers of hardware or services once the process of computerization

* Clements Auto Co. et al. v. Service Bureau Corporation, 298 F. Supp. 115 (D. Minn. 1969). Decision also reported in The Wall Street Journal, April 19, 1969.

has been started. Long lead times for system design, hardware procurement, and record transformation often make it impossible for an unhappy customer to change suppliers, thereby reducing his bargaining power in an improvident transaction. Even where a short term lease is used in an effort to preserve freedom of action, that freedom is illusory unless it is feasible to switch to a compatible system. It becomes imperative to identify potential failures of suppliers no later than the very outset of contract performance if the customer is to have genuine freedom to bargain and maneuver.

In many cases, the transactions involve financial outlays or commitments substantially greater than those usually assumed. Some system sales prices are substantial. Even the purchase of extensive time-sharing service can involve great sums in many cases.

And many types of transactions are so novel that reports of negative experience with them are not available for guidance in avoiding frictions. This is especially the case with time-sharing services.

The concrete impacts of bad contracts have been great. Suppliers have been subjected to large awards of money damages or have settled for them out of court.* Customers have suffered from severe disruptions of their operations and substantial portions of their true losses cannot be quantified and recovered as money damages.** Much annoying wrangling undoubtedly has occurred in many situations that never ended up in the courts. At the very least, innumerable customers have been extremely dissatisfied because they expected more from their contract transactions than they received.

The improvident transactions frequently impugn the technology and discourage or delay its proper use. Potential users justifiably can become gun shy from some reports of difficulties encountered. It still seems to be especially newsworthy to detail problems encountered with computer systems.

Until recently, the contents of contracts used in the computer industry appear to have been influenced largely by marketing people. Many of the form documents studiously avoided detailed statements of the subject matter involved. A large number of them reflected the greater bargaining power of the supplier. Few of them mainfested the careful draftsmanship, including respect for the rules of grammar, their intrinsic importance would seem to warrant for the reasons just detailed.

PROPER CONTRACTING APPROACH

Proper contracting in the computer area requires at least the application of the best negotiating and contract-drafting techniques found applicable to other subject matter if not probably even some superior ones. There is no mystery surrounding the nature or use of those techniques. Essentially, they require (a) thorough negotiation of each transaction to identify and cover all significant points and (b) careful statement in writing of the precise nature of the transaction worked out by the parties. Effective negotiation entails anticipation of situations that might arise during the course of the transaction and treatment of how the parties will handle them if they happen to occur. There is no substitute for painstaking thoroughness in covering both basic aspects. Normally, those functions are performed by persons knowledgeable in the subject matter and by lawyers, both working together.

Contracting for computer systems or services requires especially careful use of those techniques for the reasons noted in the review of the effects of poor practices. This care demands particular attention to the factual aspects of computer transactions and to the formulation of contract documents to cover them. The facts surrounding the transactions frequently are not obvious, but appear only after application to the transactions of imaginative analysis by technically skilled persons. Moreover, many transactions extend over long periods of time, and the persons involved in carrying them out can benefit from considerable guidance in the form of a clearly written, explicit agreement. This is particularly important because personnel turnover frequently interjects complete strangers into the middle of complex transactions.

Computer specialists must be called upon to identify the pertinent facts in contractual transactions, which might include the nature of the customer's needs, the technical aspects of the products or services considered to fill their needs, and the types of business approaches available to secure those products or services. That function of computer specialists has a number of manifestations. They must prepare specifications covering the supplier's performance and occasionally the customer's environment into which the supplier's product or service will be introduced. They must select ways for determining whether performance of the products or services has been satisfactory, usually through the use of acceptance tests. With respect to software, they must identify such factors as the

---

* See, for example, Clements Auto case, *supra*; Food Center Wholesale Grocers, Inc. v. International Business Machines Corp., (U.S. District Court for Massachusetts) (jury verdict of $53,200), The Wall Street Journal, March 27, 1968; and U.S. v. Wegematic Corp., 360 F. 2d 674 (2nd Cir. 1966).

** See particularly Clements Auto case, *supra*, and Food Center Wholesalers case, *supra*.

possible need for maintenance, the likelihood that a particular program will be enhanced, and the precise nature and form of the items that should comprise a specific software package. They should be able to evaluate the risk that a particular proprietary package will be pirated and thus requires an effort to fence it in legally. They should point out jeopardies to file information in time-sharing applications, propose physical means for preventing unauthorized access to it, and identify remaining needs for legal protections to bolster the physical fences that are utilized.

The possible contributions of the computer specialist along the lines suggested can be identified readily by a technical audience, such as that to which this paper is addressed, with slight stimulation and suggestion. To that audience, in contrast, the role of the lawyer requires elucidation. The lawyer usually functions in contract situations, as in most of his other professional activities, essentially as a craftsman rather than as a specialist in particular subject matter that might be involved. He is skilled in verbalizing the details of relationships, reducing complicated arrangements to writing, and prodding the parties for an identification of potential circumstances that require advance treatment. Lawyers are accustomed to assimilate technical subject matter sufficiently to carry out their professional function, although some of them seem, up to now, to have felt that computer-communications technology is more complex than they are prepared for. This unfortunate state of mind is not limited to lawyers in the United States. A lecturer of the Faculty of Law of the University of Stockholden notes that in Sweden "suprisingly few lawyers work with and have gained insights into the legal technicalities of computer acquisition although the whole area is well suited for legal analysis and involvement."[2]

The experienced contractual frictions adverted to above suggest that lawyers have not been involved in many computer transactions, expecially not lawyers versed in preventive law. Since the publicity on the recent lawsuits, it seems clear that lawyers will be called upon increasingly in that area. Under those circumstances, computer specialists will have an opportunity to achieve an interdisciplinary working relationship with lawyers in the contracting activity. That relationship will be most fruitful and most satisfying to computer specialists professionally if they identify their own role and fulfill it effectively. It might be helpful to consider the nature and scope of that role.

Computer specialists must take substantially full responsibility, frequently entirely on their own initiative, for identifying factual aspects that require treatment in negotiations and contract drafting. They must bring those aspects to light realistically, pointing out the genuine likelihoods that the identified situations will be encountered, the probable consequences if those situations are ignored, and the means for providing for them by contract.

Computer specialists must interpret to lawyers the significance of the factual aspects they uncover. In that effort, they must insure that the lawyers genuinely understand their explanations. This function is fraught with special difficulty because of the tendency to use unique words in the computer area, many of them coined only recently and many of them not enjoying universally accepted meanings.

Also, computer specialists must take considerable initiative in performing their function. They must identify points requiring treatment, frequently without clues or guidelines from the lawyers, and then they must persuade the lawyers, if necessary, that those points are important enough to be treated. They must devote the meticulous, time-consuming effort necessary to work up and state product or service and interface specifications, acceptance tests, and other detailed technical points. Only they can do those things. Moreover, only by preparing and adopting full and clear specifications can both parties recognize fully what is expected and required to be delivered, depending on their points of view. All too much of contractual difficulties and misunderstandings stem directly from sloppiness in that respect and a failure to reach complete agreement on those fundamental matters.

Computer specialists probably would get considerable useful guidance in identifying salient points to be covered by considering what they would want to know if they suddenly were given responsibilities, either as computer specialists or business managers, for a particular transaction, without any prior involvement with it.

It might be helpful to examine a potential time-sharing relationship as a case in point. Assume that a multi-location commercial customer is considering buying recordkeeping, accounting, and other information processing services from a supplier having data processing facilities located strategically over the country. Those services will include order processing, billing, inventory control, payment and financial reporting. The user's primary contact with the supplier will be through the input-output terminals on its premises. The complete records will be kept in binary code on disks in the supplier's system. The software will be prepared at the supplier's expense and will be used to serve other companies as well, some of which are competitors of the customer. What general points would a computer specialist alert the customer's lawyer to for treatment in the contract? He probably

would include at least the following:

1. Descriptions of the nature of the services to be provided, along these lines:
   a. Specific types of outputs to be provided, by the supplier, including precise formats and frequencies.
   b. Response and cycle times for various kinds of output.
   c. Permissible error rates in output.
   d. Permissible downtime for different kinds of services.
2. Statements of important characteristics to be included in the system to be used, such as the following:
   a. Measures to prevent unauthorized disclosure of customer's information.
   b. Means for recording usage of services for billing purposes.
3. The need for flexibility in satisfying the customer's unpredictable requirements for increased quantities and new types of services at reasonable prices.
4. Some way to assure the customer of genuine freedom to seek another source of services when a switch is permissible contractually.
5. Assurance that the customer will suffer minimal interruption even if a catastrophe strikes the computer facility that usually serves him.
6. Protection against financial exposure in case of patent infringement.

What detailed items would the computer specialist seek to have included pertinent to those general points? He would want, for example, to have exact output forms created in advance and appended as exhibits to the contract. He also would want the entire array of specific services that make up the ultimate service listed and described. They might include the furnishing and maintenance of terminals, the use of communications lines, training, the supplying of manuals, and software consultation, as well as the delivery of output papers that result from the processing of input supplied by the customer, the function on which most attention normally would be concentrated.

Why would the software specialist ask for treatment of some of the items included in the list? If the variety of system features intended to preserve privacy of customer data are specified, periodic system audits could be conducted to see whether they still are in use. The discovery of deficiencies by that measure hopefully would make it possible to prevent leaks by remedying their potential causes. Companies offer specialized services that include computer system security audits.[3] Similarly, if charges are imposed based on usage of services as determined by the supplier's computer, it might be advisable to provide for periodic audits of that portion of the system. The writer's personal experience with a telephone company persuades him of the merits of such a precaution.*

Hopefully, the customer's needs for the data processing services will increase. Since he is delegating responsibility for data processing activities to the supplier and is relinquishing his opportunity to see to it directly that those needs are filled, the customer must secure an open-ended commitment on the quantity of services he may call for as time goes on. Procedures must be worked out to make it likely that the supplier can, in fact, fill those needs. These might include the furnishing to the supplier periodically of information that would indicate the customer's expanding needs sufficiently in advance for the supplier to meet them. Arrangements for adding new data processing services at fair prices are essential, since data processing systems must be dynamic to provide new functions that become apparent to imaginative users.

A critical factor is the need to make the customer truly independent of the supplier after a firm-commitment period of reasonable length and even during that period if the supplier falls down on his contractual obligations. Otherwise, the customer is placed at a serious disadvantage in bargaining for future prices and in forcing the supplier to live up to promises he has made. At the very least, the customer should arrange to receive a signal of the advisability of switching at the end of the initial period long enough in advance to work out the change in the source of data processing, possibly even to an in-house set up. Some consideration might be given to the feasibility of securing rights to use the supplier's software after termination of the supply contract, as a means of achieving that freedom. However, that approach is a most complex one that must be examined extremely carefully from the points of view of economics and technical feasibility.

Finally, means to avoid the adverse consequences of always potential catastrophes must be included. At the very least, frequent dumps of records for off-premises storage must be required and provisions must be made for recreation of the records from interim transaction input source documents. Also, the availability, on short notice, of back-up facilities and alternate communications means must be insured.

* Many years ago, the writer's informal, personal challenges of the accuracy of a series of bills for telephone service based on the quantities of local calls made resulted in a substantial refund covering all charges for excess calls.

These examples could be enlarged substantially. However, they should be sufficient to suggest the contributions the computer specialist will make.

We have explored what the computer specialist can and should do in an interprofessional contracting effort. But his ability to contribute can be without value unless it is made available effectively. How should the computer specialist undertake to perform his function? Essentially, he must develop a genuinely mutual relationship with the lawyer on the team. That requires that he both acquire an understanding of the legal role and interpret his own role to the lawyer. For the professional comfort of the people involved, each of them must make it clear in individual situations that a true synergism exists between them and that neither will attempt to dominate the situation or otherwise place the other in a professionally unsatisfactory light.

The computer specialist should inform the lawyer, where appropriate, of the desirability to the specialist of explanations of subtleties of legal phraseology. For example, it frequently would be instructive to be aware of the true significance of a stated "warranty against defects in materials or workmanship" when it is followed by language imposing time limitations on the customer's enjoyment of the "warranty" and particularly that specifying what action the supplier is obligated to take when given timely notification of a claimed defect.

He also might consider whether expressions the lawyer attempts to use to describe situations in the computer industry or technology have any real meaning. For example, he might encounter a statement such as "an on line, real time, teleprocessing environment, as those terms are commonly accepted in the data processing industry." Do those terms really have a regularly accepted meaning anywhere? Is there actually something that might be called "the data processing industry"?

The computer specialist should furnish pertinent advice in that respect, as well as in others, to avoid the creation of pitfalls that can have serious adverse consequences in the future. It is easy to pass such errors by during the contracting process because they have no obvious adverse impact at that time. This aspect is reminiscent of the observation made to the writer by a man whose will was patently deficient in that it would deprive his heirs of substantial sums of money because of a failure to avoid unnecessary taxes. He blandly declared, "I've had this will for five years and it's been perfectly all right for me."

The computer specialist also can perform a very useful function in many cases by stimulating the lawyer to look for unique legal exposure in the transaction. For example, it is important to take whatever precautions are possible to provide that the information itself stored in a time-sharing system, as distinguished from the records of that information that include relatively expensive media owned by the supplier, are the property of the customer and to prevent loss of that information to outsiders through legal process used to levy on the recording media or to secure information for litigation through so-called discovery procedures.

In fulfilling his responsibilities to inform and guide the lawyer, the computer specialist must present his advice with scrupulous regard for professional niceties. Preferably, the advice should be given to the lawyer discreetly without providing any basis for embarrassment. Of course, the computer specialist is entitled to no less courtesy and should insist upon receiving it.

In fact, the interdisciplinary effort entails mutuality of such explanations, in both subject matter and manner of presentation. The lawyer should request and the computer specialist should supply graciously explanations of words and expressions stated in computerese. As a matter of practice, such inquiries by the lawyer provide useful tests of the meaningfulness of the mode of expression for the somewhat technical aspects of a contract. It is essential that substantially all of a written agreement, if not the entire agreement, be readily understandable by non-technical people. Frequently, such people have major roles in the performance of agreements. Those people might include purchasing agents, contract administrators, and billing clerks, for example. And it is necesary to be prepared for the fortunately rare, but extremely important, occasion when an agreement ends up in court. There, understandability could be critical, and the understanding of technical material frequently experienced is at a relatively low level and is rarely aided by the awkward procedure provided for exploring the facts. As in other respects, if the lawyer does not question, on his own initiative, the meaningfulness of expressions on technical aspects, the computer specialist should test his understanding of them and remedy them if necessary. If the material is deficient in that regard, getting it by the lawyer is a false victory.

A CHALLENGE AND AN OPPORTUNITY

As this discussion indicates, poor contracting practices have created a general situation with serious adverse financial and other consequences for contracting parties in the computer area. The solution seems to involve the adoption of contracting techniques whose soundness has been demonstrated in other areas.

Achieving that solution entails an interdisciplinary effort by computer specialists and lawyers, working as a team. That essential approach presents significant challenges to identify the distinct, complementary roles of the participants and to establish working relationships that are professionally mutually satisfactory.

In undertaking to meet those challenges, computer specialists have great opportunities to expand not only their professional involvements but also their knowledge of the functioning of the legal process.

It is hoped that computer specialists will take whatever initiative is necessary to insure that they will be called upon to meet those challenges and enjoy those opportunities. Such initiative might include preparation for the role and demonstration of an ability to fulfill it. Undoubtedly, many lawyers will welcome the opportunity to team up with computer specialists so qualified.

REFERENCES

1 R N FREED
Get the computer system you want
Harvard Bus Rev pp 99-108 Nov-Dec 1969

2 P SEIPEL
Introducing law students to computers: Swedish experiences
Rutgers J of Computers and the Law Vol 1970 Spring pp 88-93

3 B ALLEN
Danger ahead: Safeguard your computer
Harvard Bus Rev pp 97-101 Nov-Dec 1968

4 J J WASSERMAN
Plugging the leaks in computer security
Harvard Bus Rev pp 119-129 Sept-Oct 1969

Editor's Note

Pages 149 through 158 have been deleted from this volume.

Pages 149 - 158 deleted from volume

Selected R&D requirements in the computer and information sciences*

by M. E. STEVENS

National Bureau of Standards
Washington, D. C.

INTRODUCTION

Under the provisions of the Brooks Bill, PL 89–306, enacted in 1966, the Center for Computer Sciences and Technology of the National Bureau of Standards has been authorized to sponsor and to conduct research and development work in the computer and information sciences and technologies, especially where the problems are unique to Government or where the results are likely to have wide applicability in Government operations. In addition the CCST attempts to: (1) maintain awareness of advances in the field of automatic data processing and related sciences and technologies (a truly broad interdisciplinary spectrum); (2) disseminate information on advanced developments, especially with a view toward the cross-fertilization of ideas, and, (3) identify areas where breakthroughs, either theoretical or pragmatic, are needed in order to achieve more effective or efficient use of ADP techniques or in order to anticipate future requirements for standardization efforts.

In this context, a first and necessary approach to R&D requirements analysis is the apparently obvious, (but often neglected) one of fact-finding. As a cliche has it, we must find, fix, focus, and face the facts. To find, fix, and focus the facts, the CCST has been engaged over the past several years in selective reviews of the literature in the computer and information sciences and technologies, and in related fields, for the purpose of identifying areas of continuing R&D concern. A series of reports, resulting from such reviews, is in process of preparation and publication.

To date, the first three volumes of NBS Monograph 113, *Research and Development in the Computer and Information Sciences,* have been issued. They cover, respectively, information acquisition, sensing, and input (Volume 1); information processing, storage, and output (Volume 2), and overall system design considerations (Volume 3), including, for example, requirements and resources analyses in network planning, software/hardware requirements, and advances in hardware technologies.

Further reports in the series are planned to include such topics as the maintenance of the integrity of privileged files, the domain of information selection, storage and retrieval (ISSR) research, ISSR and library automation system development requirements, natural language processing, and problems of system evaluation. In effect, the present paper presents a preview, an overview, and a summary of the projected series of reports, prepared in advance of the completion of several proposed further volumes.

The difficulties of undertaking such tasks of reviewing and reportage should be obvious. The scope and the spectrum of areas of concern range quite literally from A to Z—automata theory to zero-crossings in speech recognition, automatic abstracting to Zip codes, and Abelian groups to Zipf's Law, for example.

Marvin Minsky's "Steps Toward Artificial Intelligence"[1] was a Renaissance-Man's approach to pattern recognition, language processing, question-answering and machine-aided inference, CAI (Computer Assisted Instruction) and CAD (Computer Aided Design) and CAC (Computer Assisted Classification). But what of adaptive modems, computer generation of kinoform representations of three-dimensional objects (some of which may be purely conceptual), photochromic or thermoplastic data storage, laser pipes for communication systems, large-scale display screens with half-tone or color capabilities? What of the automatic segmentation of very large and complex computer programs? What of the behavioral effectiveness of men in the man-machine interactive situation? It is obvious, therefore, that the authors of the review-series reports (and the author of the present advanced

* Contribution of the National Bureau of Standards

summary paper in particular) will be Jacks-of-all-trades, and masters of none.

As Punchard[2] points out with considerable cogency, no one, in 1939, could have predicted the invention of the transistor (1948) and certainly not the first steps of man upon the moon in 1969, which could not have been achieved with vacuum-tube technology. Who, then, would be rash enough to guess the inventions and innovations likely to occur in the next 30 years, from 1970 to 1999 (assuming, of course, that man will be able to adequately control his societal relationships, his environment, and his machines)?

Nevertheless, in terms of R&D requirements that can be immediately foreseen, certain trends would appear to be indicative of probable early progress and accomplishment. Among them are the following:

- First, there is increasing recognition that we are faced with what Warren Weaver, as early as 1948,[3] called "the problems of organized complexity".
- Second, a critical emphasis is beginning to emerge upon the *man* in man-machine relationships, including not only behavioral questions but (even more importantly) problems of the enhancement of individual creativity and of the protection of individual privacy.
- Third, data compaction and redundancy reduction requirements are beginning to be recognized in all fields of information processing and teleprocessing, from remote data collection by means of automatic sensors to the systematic purging of libraries and files. Filtering of telemetered data, for example, is crucial to effective analysis and interpretation. Image enhancement in pattern recognition, reduction of synonymity in natural language processing, and data compression in speech or pictorial data transmission are other prime examples.
- Fourth, there is a growing convergence of cross-disciplinary efforts in a number of significant problem areas.

A common attribute of the typical problem areas is that of steadily increasing complexity—advanced hardware in processor and system design requiring new and unprecedented programming techniques to take full advantage of parallel processing, associative memory access and organization, and multiprogrammed system control, for example. Further, it is becoming increasingly difficult to isolate and insulate communication and computing functions in the emerging teleprocessing networks. It will be increasingly necessary to deal effectively with hierarchies of language, hierarchies of storage devices, and hierarchies of systems. The challenges of communication at all levels— man to man, man to machine, and machine to machine— demand our immediate and committed attention as interdisciplinary specialists *now*, and for the foreseeable future.

The state of the art and the state of the practice of the computer and information sciences and technologies pose enormous and virtually unprecedented challenges for the Seventies, and beyond. These challenges are truly interdisciplinary. They are indeed the problems of organized complexity. They intimately and interdependently involve major problems of both systems and society.

Facing these facts involves the recognition that, as of the beginnings of the Seventies, we are giving Topsy (who "just growed") the initiative. Many, many interests (from suppliers to users) are primarily concerned with the status-quo-ante-bellum, yet many developments with the most serious socio-economic implications are, in fact, on-going. In many fields, technological developments often outstrip requisite social awareness and control. In our field of the computer and information sciences, the question of social responsibilities should be, but seldom is, of vital concern to all of us.

SOME REQUIREMENTS IN TERMS OF GENERALIZED INFORMATION PROCESSING AND TELEPROCESSING FUNCTIONS

Considering first a generalized information processing system, we see in Figure 1 the functional areas in which R&D requirements arise. With respect to Box 1, for example, we can identify more specific requirements as shown in Figure 2. Similarly, some of the areas of R&D concern for Boxes 2 and 3 are shown in Figure 3.

It is impossible in a short summary, of course, to do more than to sample, often quite randomly, a few of the R&D implications shown. For example, we may merely note that image enhancement operations in pattern recognition (such as stroke thinning, contrast enhancement, and noise elimination) are complemented in the case of natural language processing by KWIC-indexing "stoplists" and by synonym reduction and homograph resolution.

However, certain illustrative trends towards future development, improvement, and exploitation requirements may be emphasized. First, with respect to information acquisition, sensing, and input, the fol-

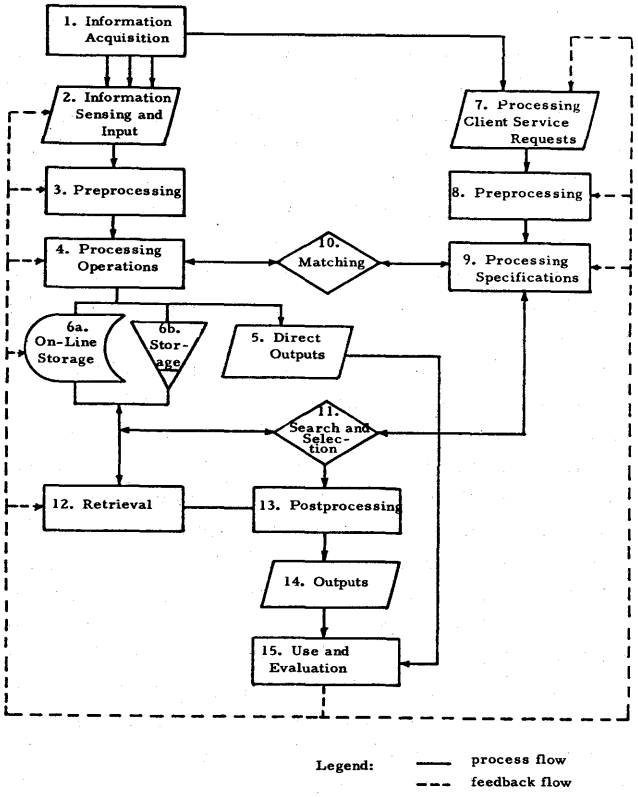


Figure 1

lowing major areas of concern are indicated:

- Data filtering, compaction, and compression—in general, the reduction of redundancy, but with reserve capability for reproduction or regeneration of reasonable facsimiles as required,
- Teleprocessing networks—their design, implementation, use, and regulation as necessary,
- Advanced communications techniques—further development of CATV, domestic satellite networks, laser pipes,
- More versatile, efficient, and economical OCR—especially with respect to open-ended character sets, exotic alphabets, true page reading, graphic-symbol recognition, two- and three- dimensional graphic inputs and identifications, and
- Audio inputs, including speaker identification and speech recognition.

Secondly, with respect to pattern recognition generally, continuing empirical and theoretical developments will be required in such areas as the following:

- Determination of membership in classes especially in the case of non-linearly separable functions,
- Recognition of cursive handwriting, continuous and intonated speech, and fingerprints,
- Scene analysis, and
- Automatic categorization and classification, in general, including text in the case of automatic subject content analysis.

Communications system challenges include: global facsimile transmission, automatic international telephone and data transmission networks, and the extension and practical application of optical data transmission techniques (for example, as of today, lasers are useful primarily at the opposite ends of the distance spectrum—that is, for only a few miles or for millions of miles[4]). Other communications hardware requirements have been predicted for much higher speeds of switching and signaling systems, for high-velocity four-wire transmission facilities, for common controls in switching systems, and for equipment capable of handling different bandwidths.[5]

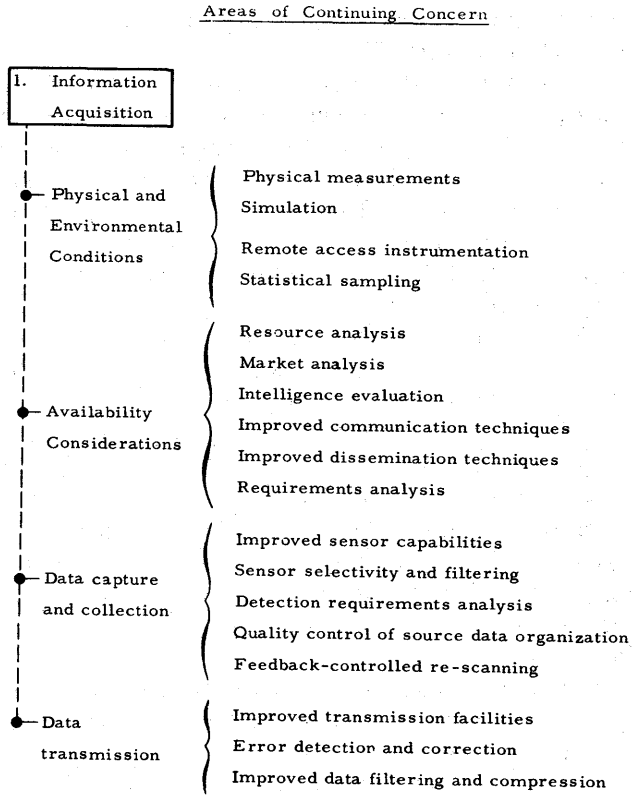


Figure 2

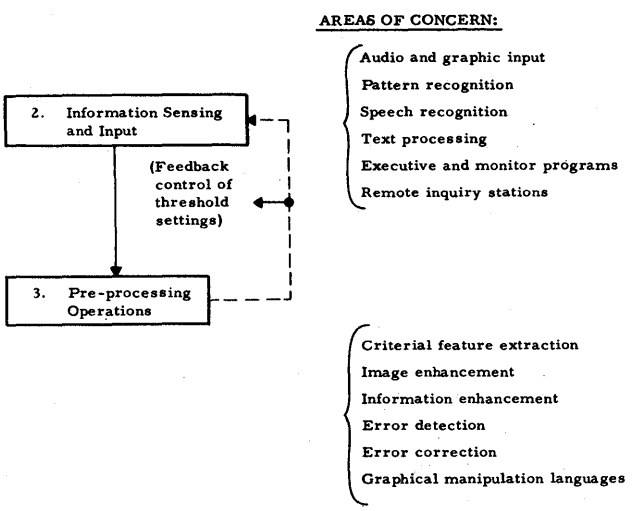


Figure 3

Even more importantly, it is to be noted that: "The introduction of Data Communications to an established data processing operation is an essentially revolutionary development: it changes the lives of everyone concerned. It places new demands on the manager, the systems analyst, the programmer, and the computer operator, and presents them with a new set of problems."[6]

In particular, the problem of man-machine interfacings, in terms of various levels of terminal design, user-oriented languages, human engineering, and behavioral or attitudinal factors deserve intensive and continuing attention. Terminal keyboards, text-editing facilities, graphical communication devices, and other services or facilities should not only be easy to use, but easy to learn how to use.

Considering next the information processing and storage functions of Figure 1, we note in Figure 4 some of the R&D requirements affecting these operations. For example, in the area of processing system planning and management, we must learn more about realistic needs for modularity, replication, and reconfiguration of system and network components; for system self-checking and self-repair, and for on-line diagnosis and instrumentation. Then, as problems and needs are defined, the appropriate R&D efforts must be undertaken.

Can the techniques of input-output economics be applied to systems design of information processing and information utilization operations? If so, as Tell suggests, a measure comparable to dollar value for the effectiveness of information flow must yet be developed.[7]

Requirements for handling a variety of input and output modalities and for processing more than one I/O channel simultaneously clearly indicate the need for continuing R&D efforts in parallel processing, multi-access scheduling, and multiprogramming, as well as in the sensing and display techniques as such.

Priority scheduling, dynamic allocation and reallocation of system and network resources, and protection of access to only authroized users, are all problems presenting relatively unprecedented challenges not only to the technologies but to the underlying sciences as well.

On the theoretical side, we note that advances in automata theory may provide means for the realization of desired matching functions in hardware with minimal component requirements; that iterative circuit computer designs (such as Holland machines[8]), might be effectively applied to problems of pattern detection and identification, and that "an important goal of automata theory is a basic understanding of the computational process for various classes of problems. The theory must go beyond the notion of computation and how that difficulty is related to the organization of the machine that performs the computation."[9]

Other examples of R&D requirements affecting Figure 4 include the following:

- Logical design for effective utilization of large interconnected circuit arrays,[10]
- Multivalued logic systems permitting more effective use of advanced opto-electronic elements,[11]
- More flexible pagination mechanisms, especially for graphical data processing,
- Design and use of multilevel storage systems,
- Maintaining file integrity, from the protection of privileged information to the provision of adequate roll-back and recovery procedures in the case both of minor failures and also of catastrophes,
- More effective on-line debugging, diagnostic, and instrumentation techniques, and
- More accurate and more sophisticated simulation models.

Next, typical R&D requirements with respect to output are shown in Figure 5. Some specific questions of concern are as follows: How adequate are remote console and keyboard designs? What about large-scale, high-resolution displays for group viewing? How important is color or three-dimensional projection or highlighting of selected features of a display? These and related questions raise R&D questions of both hardware and human engineering.

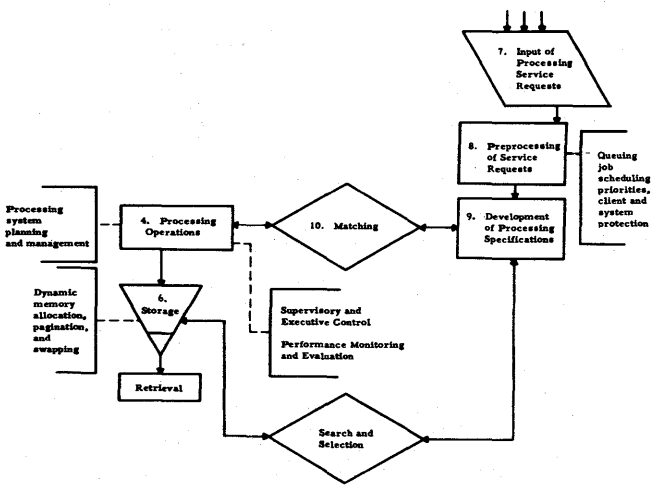


Figure 4

It will have been noted that certain rather obvious and important areas of research and development have been omitted from Figures 1–5. They are considered to be of broader applicability, affecting more than one Box or function. Volume 3 of our series, therefore, separately reviews such overall design considerations as requirements and resources analysis; problems of system networking; input/output, terminal design, and characters sets; advanced software; advanced hardware, and debugging, on-line diagnosis and simulation. Here, we will consider for examples of pertinent R&D problems only the areas of programming and advanced hardware technology.

Continuing R&D concerns in the area of programming languages represent several conflicting requirements. On the one hand, there is recognizable need for common-purpose languages that will be compatible with a wide variety of systems and types of applications; on the other hand, there are needs for hierarchies of language systems. In addition, a number of specialized requirements exist for new and more versatile languages, geared to the user who is relatively unfamiliar with computer systems, especially in such areas as on-line problem-solving and graphical data processing.

For example, Newman points to "the difficulty of specifying new graphic techniques using the available programming languages"[12] and Sutherland states: "The lack of precise ways to formulate and represent graphical language fundamentals impedes the use of graphical techniques in many problem areas."[13] Direct input and manipulation of two- and three-dimensional graphic data is of major interest in terms of machine-aided design (architecture, automotive styling, highway engineering, and ship design, among many examples) and pattern recognition (including the automatic analysis and interpretation of ecological, agricultural, geological, and other "patterns).

Another R&D need indicated in the literature is that of automatic segmentation of very large programs.[14] Again, a challenge posed by Burge in 1966 appears as fully pertinent today, namely: "Can we get a computer program to scan a library of programs, detect common parts or patterns, extract them, and re-program the library so that these common parts are shared?"[15]

Typically, the spectrum of available and desired programming languages ranges from a wide variety of relatively natural, but constrained, conversational uses of remote consoles interacting with processor centers to highly sophisticated executive control, monitoring, and simulation languages. More and more complexity is being required in executive, control, and protection programs, while at the same time typical users are less and less likely to have had programming or computer training.

Continuing progress (as Perlis noted in the first ACM Turing Lecture) will depend upon the balances to be achieved "between efficiency and generality".[16]

Finally, we note that: "Work on programming languages will continue to provide a basis for studies on languages, on the concept of grammar, on the relation between actions, objects and words. . . ."[17]

Hardware areas of R&D concern include, first, the increasing interdependence of hardware and software and the emergence of "firmware" (or wired-in microprogramming).[18] Next, there are advanced technologies (opto-electronics, holography, laser technology, photochromics, large-scale integration, or LSI, and

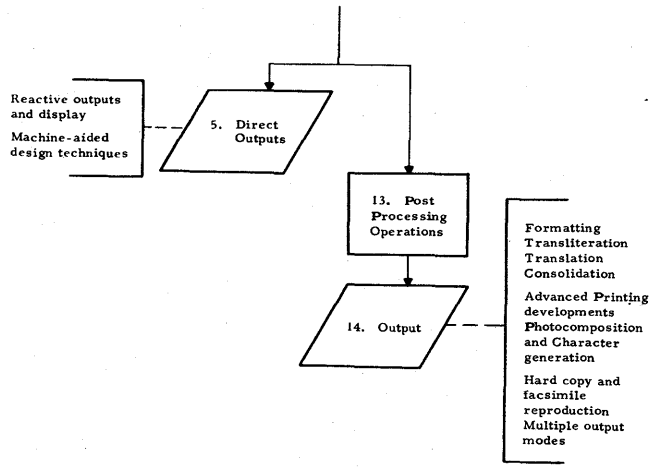


Figure 5

the like) for such applications as the following:

- Interconnection problems in integrated circuit design, aggravated by the advent of LSI,[19]
- High-density, multi-bit data storage, including color-coding of a single "bit" position in advanced storage media,[20]
- Efficient recording and regeneration of three-dimensional and color images, and
- Improved design of scanning and image enhancement techniques.

Some more specific examples are as follows:

- Laser beam recording does not require a vacuum, by contrast with electron beam recording, which does. However, for many applications, there appear to be continuing problems with laser deflection techniques.[21,22]
- "At today's pace of innovation, holography may be outmoded before it approaches being practical. One of the latest competitors for 3-D display, storage, and wave conversion applications is the kinoform".[23] More particularly, "the kinoform is a new, computer-generated, wavefront reconstruction device which, like the hologram, provides the display of a three-dimensional image. In contrast, however, the illuminated kinoform yields a single diffraction order and, ideally, all the incident light is used to reconstruct this one image".[24]
- LSI permits the wiring-in of "spares" on the same wafer, so that diagnostic logic and special programs can be used to effect self-repair;[25] further, advances in LSI should make associative memories of reasonable size practical.[26]
- "Photochromic high-resolution films coupled with proper light sources and optical systems can provide the storage of millions of bits to the square inch. A micro-holographic indexing system used with such storage devices may revolutionize data storage and retrieval".[27]

A final illustrative area of hardware R&D concern is that of storage developments both with respect to high speed main memories and also with respect to very large storage capacities. Little more than a year ago, the race for increased speed was in terms of nanoseconds, but today we are beginning to talk of picoseconds (trillionths of a second).[28]

There have been numerous predictions in the past as to the skyscraper proportions of the equipment that would be required to store all the information contained in the collections of the Library of Congress or the U.S. Patent Office. These proportions are rapidly shrinking, with data cells, ultrafiche, and other advanced technologies, and they will become entirely outmoded if planned research in molecular storage can be brought to fruition. However, the greater problem is: how can men organize the information to be placed in such vast machine stores so that it can be effectively manipulated, correlated, selected, and retrieved?

The growing interdependence of systems and society demands the symbiosis of men and machines;[29] but with a far greater emphasis upon the *man* (as transmitter, receiver, editor or monitor, guardian, publisher, and user of machine-processed information) than has prevailed up to now. Thus we may preview other reports in the R&D requirements series primarily in terms of communication problems: man-with-machine, machine-with-machine, and man-with-man.

COMMUNICATION PROBLEM AREAS

Three major communication problem areas that involve computer and information sciences research and that emphasize the man in man-machine relationships can be identified as follows:

- The human factor problems in system planning, use, and evaluation,
- The "man-machine communication gap . . . the greatest obstacle to the wider enjoyment of the services of the computer",[30] and
- Man-to-man communication as aided by machine, especially with respect to the improved utilization of recorded knowledge.

An important approach to the determination of R&D requirements in the computer and information sciences is the identification of handicaps currently being encountered in reductions-to-practice. The first and most obvious problem is that of problem definition itself. In particular, the area of requirements analysis in system design and system planning presents many current difficulties. While the introduction of information processing techniques has not changed the *kind* of fact-finding, analysis, forecasting, and evaluation operations required, it has changed the *degree* of the efforts required.

Yet with respect to such critical factors as user behavior, user reactions, and especially user effectiveness, there is very little objective data available. Similarly, with respect to questions of loadings of teleprocessing networks new methodologies of requirements and resources analysis are needed, and better

communication between the potential users of systems and the systems designer must be developed.[31]

The handicap of handicaps is that, in general, we do not know enough about human behavior. What is available, or what can be developed, with respect to user-need and market analyses? The airlines, with operational teleprocessing networks today, can keep their operations effective because their markets are constrained and therefore their typical user requirements can be forecast with reasonable accuracy. So, too, with command and control systems within a specific area of defense responsibility. The case is not so, however, with commerical time-sharing services in general (although some such services may be offered only for restricted types of applications such as banking tansactions) and certainly not for nationwide (and even international) information acquisition, announcement, dissemination, and retrieval services.

Why should behavioral research with respect to man-machine interactions be required in the planning and design of advanced information processing and teleprocessing systems? Many problems of unknown human behavioral factors in the design, use, and evaluation of such systems have, in effect, been "swept-under-the-rug"—perhaps because of the very obvious difficulties of fact-finding, and undoubtedly because of the even more obvious difficulties of arriving at effective and valid solutions.

Some specific examples of R&D requirements cited in the literature are as follows:

- "The crucial regions for research and development seem to lie on both sides of the literal interface".[32]
- "Very little is known about individual performance differences, user learning, and human decision-making, the key elements underlying the general behavioral dynamics of man-computer communications".[33]
- "Human errors and the education of human beings are, therefore, two systems design factors which must receive significant attention".[34]
- "Quite often the most important parameter in a system's performance is the behavior of the average user. This information is very rarely known in advance. . ."[35]

Other critical factors largely neglected both in system design and in system use are the questions of the accuracy and reliability of the content of the information in the system. Thus, the development of improved reliability indicators and automatic inference techniques should be of serious R&D concern.[36]

Another type of handicap indicative of continuing R&D requirements with respect to communication problem areas is that of current lack of compatibility and convertibility between systems and between system components, in the sense that we need convertibility by means of interface equipment and procedures, compatibility at least to the extent of conventions for interchange and common practices, and standardization wherever feasible. Voluntary American National Standards already exist for various information-interchange requirements, including bibliographic reference data formats on magnetic tape and ASCII, the American Standard Code for Information Interchange, but much more work is needed on extended character sets and on transparent-mode procedures, and these are only the beginnings of the machine-to-machine communication problems.

By contrast, however, it may be said that not even the beginnings have been made on the problems of man-machine communications. In particular, a major goal of on-line responsiveness and interactive feedback is to "free the man-machine interface from the need for letter-perfect information representation by the man".[37]

The problems are aggravated by the plethora of time-sharing languages ("the number of time-sharing installations is roughly equal to the number of languages offered among them"[38]); by the "glaring experimental lag" in systematic studies of man-machine communication techniques and procedures[33], and by recognition that: "The potentials of information systems that adapt to the user's response patterns are yet to be realized . . . [and] "we have been . . . humbled and challenged by our ignorance of how a dialogue should be structured, how we should mold the machine to fit the man".[39]

Three years ago, J. C. R. Licklider pointed out that the real difficulties of networking for teleprocessing purposes are not likely to be problems of hardware but rather that they will be predominantly those of ". . . social and software organization, of conventions, formats, and standards, of programming, interaction, and communication languages".[29]

Beyond the individual man-machine communication problems are those of society (men-at-large) and machine interactions. The protection of privileged information is the most obvious example of the socio-political problem. How far should the machine possibilities for rapid data accumulation, correlation, analysis, and summation (of vital importance to national, regional, and local planning, to law enforcement, to nationwide services in medical care, to public welfare programs) be allowed to proceed at the expense of some loss of personal privacy? What penalties (pro-

fessional, social, legislative, juridical) are to be imposed, and what sanctions incurred, for abuses?

Requirements for extensive training and retraining[40] of potential users (scientists, engineers, and practitioners in a wide variety of professional fields; executives, managers, legislators, judges; businessmen and insurance men and bankers; computer programmers, scholars and students, and perhaps even housewives) for effective use of man-machine facilities raise the even more important, and more difficult, problems of man-to-man communications, especially as they may be machine-assisted.

In terms of general-purpose network planning (and one man-machine combination to another man-machine interaction in particular), we are faced with "the vast possibilities and intricacies of direct human communication".[41] More generally, we may note the importance of oral communications, especially in the sense of the "Invisible College".[42]

However, much of the thrust of R&D requirements in the computer and information sciences is toward effective utilization of *recorded* knowledge, and, particularly, the efficient handling of reported scientific and technical information.

In a technological sense, the device that launched the first major revolution in communication and documentation since the invention of written alphabets, the printing press, is today obsolete. Increasingly, techniques of photocomposition and electronic generation of characters and other graphic symbols are replacing those of movable-type. The more cogent question, however, is: will the printing press and its latter day equivalents become obsolete in the societal sense? Has the book had its day; will scientific and technical journals fade away? Will libraries, as we know them be no more?

As of the beginnings of the '70s, printing and paper are still with us, and this is likely to continue to be the case for some time to come. Nevertheless, new techniques are revolutionizing our traditional methods of preparation, publication, and dissemination of recorded information: Machine-assisted editing, computer-generated preparation of camera-ready copy or direct output to microform, publication in microfiche form, mechanized indexing and selective dissemination, and on-line text search together portend a second major revolution in communication and documentation.

There has been no one Gutenburg (or his Chinese predecessor); instead, there have been a number of interdisciplinary contributions. In the review-series reports not yet completed, we have stressed ISSR, natural language processing, and machine-aided inference, as in question-answering systems or "cognitive economy".[43]

The contributions of the computer sciences to the more traditional library and information sciences lie in such areas as the following:

- The automation of the production cycle, from author to typesetter to user, often via selective dissemination systems,
- The development and utilization of machine systems for information selection, storage, and retrieval,
- The design and implementation of teleprocessing networks for the interchange and cooperative processing of bibliographic information, including the effective use of facsimile transmission,
- The continuing advancement of storage techniques and media, both graphic and digital, including not only increasing capacities at steadily decreasing costs but also the development of high-order-relational schemes of association, selection, and recall,
- The further development of natural language processing techniques, with emphasis upon moving from syntactics to semantics, from context dependency to context expectancy, from extraction to abstraction, and
- The advancement and exploitation of question-answering, error-detection and consistency-checking, and decision-making capabilities of machines.

However, our knowledge of the principles and mechanisms of information transfer in man-with-man communications has remained discouragingly primitive by comparison to the mastery of technologies for information handling (meaningful communication as opposed to mere transportation or transmission).

Natural language processing requirements arise both in the handling of scientific and technical information (machine-compiled and machine-generated indexes; information search and retrieval as in the case of the legal literature for example) and man-machine dialog situations (CAI, question-answering systems). On-line searching of subject-content-indicators, other selection criteria and full text (simultaneously, if desired) is already practically realizable, and, by the beginnings of the '70s, respectably large and diversified data bases are beginning to become available on a commerical service basis. We need, however, to experiment (as Salton[44] has on a test-basis) with varying search strategies and with the identification of the very real problems of synonymity, multiple meanings of a given character string, and noise (such as garbles on input). Such R&D efforts may provide us with sensible

clues to the solutions of problems of misspelled, misleading, and even missing data.

The major handicap in the area of natural language processing, nevertheless, remains the lack of adequate R&D efforts in semantics, upon which rock the initially promising MT efforts have nearly all floundered. Only the mere beginnings have been made, for example, in the use of contextual information for homograph resolution.

The use of contextual information for error detection and correction can hardly be overemphasized, whether it is for OCR input, misspelled or mispronounced names (both people and pharamaceuticals), or the recovery of an encrypted message garbled in transmission. Context *expectancy*, rather than merely context dependency, is a likely clue to improved techniques of automatic syntactic analysis, classification or indexing by computer, and pattern recognition generally.

Many R&D challenges remain in these areas, from the specific (in automatic categorization or statistical association, for example, there are severe problems of matrix manipulations when the collections of data or objects are realistically large) to the truly fundamental—the problems of human perception, learning, concept formation, knowledge, and communication, as such.

The proper study of mankind, even in an age of accelerating machine sophistication, is man. Thus, for the long-range, the social scientists, the psychologist, and the epistemologist are likely to play increasingly crucial roles in the basic computer and information sciences. What we know, how we know, and how machines can help us know, are likely to be the themes for AFIPS conferences in the '80s. Hopefully, for science, for systems, and for society, the truth shall make and keep us free.

REFERENCES

1 M MINSKY
Steps towards artificial intelligence
Proc IRE 49 No 1 pp 8-30 Jan 1961

2 J C R PUNCHARD
What's ahead in communications
IEEE Spectrum 7 No 1 pp 51-54 Jan 1970

3 W WEAVER
Science and complexity
Amer Scientist 36 pp 536-544 Oct 1948

4 J VOLLMER
Applied lasers
IEEE Spectrum 4 pp 66-70 June 1967

5 T B WESTFALL
Global cables and satellite communication
IEEE Spectrum 3 No 10 pp 64-66 Oct 1966

6 G O'TOOLE
Preparing for data communications
Computers & Automation 18 No 5 pp 33-35 May 1969

7 B V TELL
Auditing procedures for information retrieval systems
Proc 1965 Congress FID 31st Meeting and Congress Vol II Washington D C Oct 7-16 1965 pp 119-124 Spartan Books Washington D C 1966

8 J HOLLAND
A universal computer capable of executing an arbitrary number of sub-programs simultaneously
Proc Eastern Joint Computer Conf Vol 16 Boston Mass Dec 1-3 1959 pp 108-113 Pub by Eastern Joint Computer Conf 1959

9 J HARTMANIS P M LEWIS II R E STEARNS
Classifications of computations by time and memory requirements
Information processing 1965 Proc IFIP Congress 65 Vol 1 New York N Y May 24-29 1965 Ed W A Kalenich pp 31-35 Spartan Books Washington D C 1965

10 L C HOBBS
The impact of hardware in the 1970's
Datamation 12 No 3 pp 36-44 Mar 1966

11 E M RING H L FOX L C CLAPP
A quantum optical phenomenon: implications for logic
Optical and electro-optical information processing Ed J R Tippett et al pp 31-43 MIT Press Cambridge Mass 1965

12 W M NEWMAN
A system for interactive graphical programming
AFIPS Proc Spring Joint Computer Conf Vol 32 Atlantic City N J Apr 30-May 2 1968 pp 47-54 Thompson Book Co Washington D C 1968

13 W R SUTHERLAND
Language structure and graphical man-machine communication
Information System Science and Technology Papers prepared for the Third Cong Scheduled for Nov 21-22 1966 Ed D E Walker pp 29-31 Thompson Book Co Washington D C 1967

14 M N PERRY
Handling very large programs
Information Processing 1965 Proc IFIP Congress 65 Vol 1 New York N Y May 24-29 1965 Ed W A Kalenich pp 243-247 Spartan Books Washington D C 1965

15 W H BURGE
A reprogramming machine
Commun ACM 9 No 2 pp 60-66 Feb 1966

16 A J PERLIS
The synthesis of algorithmic systems
Proc 21st National Conf ACM Los Angeles Calif Aug 30-Sept 1 1966 pp 1-6 Thompson Book Co Washington D C 1966 Also in J ACM 14 No 1 pp 1-9 Jan 1967

17 A CARACCIOLO DI FORINO
Special programming languages
Centro Studi Calcolatrici Elettroniche 21 p Universita di Pisa Italy 1965

18 A OPLER
New directions in software 1960-1966
Proc IEEE 54 1757-1763 Dec 1966

19 B R SHAH K L KONNERTH
Optical interconnections in computers
1968 WESCON Technical Papers 16/5 Aug 1968 6 p

20 A FLEISHER P PENGELLY J REYNOLDS R SCHOOLS G SINCERBOX
An optically accessed memory using the Lippman process for information storage

Optical and electro-optical information processing Ed J R Tippett et al pp 1-30 MIT Press Cambridge Mass 1965

21 L C HOBBS
Display applications and technology
Proc IEEE 54 pp 1870-1884 Dec 1966

22 W A GROSS
Information storage and retrieval, a state-of-the-art report
Ampex READOUT special issue 9 p Ampex Corp Redwood City Calif 1967

23 Datamation 15 No 5 131 May 1969

24 L B LESEM P M HIRSCH J A JORDAN JR
The kinoform: a new wavefront reconstruction device
IBM J Res & Dev 13 No 2 pp 150-155 Mar 1969

25 E C JOSEPH
Computers: trends toward the future
Proc IFIP Congress 68 Edinburgh Scotland Aug 5-10 1968 Vol 1 invited papers pp 145-157 North Holland Pub Co Amsterdam 1968

26 R F GRAHAM
Semiconductor memories; evolution or revolution
Datamation 15 No 6 pp 99-101 June 1969

27 *R and D for tomorrow's computers*
Data Systems pp 52-53 Mar 1969

28 Datamation 15 No 4193 Apr 1969

29 J C R LICKLIDER
Interactive information processing
Computer and information sciences II Proc 2nd Symp on Computer and Information Sciences Columbus Ohio Aug 22-24 1966 Ed J T Tou pp 1-13 Academic Press New York 1967

30 M HALPERN
The case for natural-language programming
AFIPS Proc Fall Joint Computer Conf Vol 29 San Francisco Calif Nov 7-10 1966 pp 639-649 Spartan Books Washington D C 1966

31 L C CLAPP
Some brainware problems in information systems and operations analysis
Information System Science and Technology Papers prepared for the Third Cong scheduled for Nov 21-22 1966 Ed D E Walker pp 3-6 Thompson Book Co Washington D C 1967

32 J C R LICKLIDER
Man-computer interaction in information systems
Toward a national information system Second Annual National Colloquium on Information Retrieval Philadelphia Pa Apr 23-24 1965 Ed M Rubinoff pp 63-75 Spartan Books Washington D C 1965

33 H SACKMAN
Current methodological research
Position paper for sessions on managing the economics of computer programming Proc 23rd National Conf ACM Las Vegas Nev Aug 27-29 1968 pp 349-352 Brandon/Systems Press Inc Princeton N J 1968

34 C S PEDLER
New variables in the data processing equation
Computers & Automation 18 No 5 pp 28-30 May 1969

35 E YOURDON
An approach to measuring a time-sharing system
Datamation 15 No 4 pp 124-126 Apr 1969

36 R M DAVIS
Information control in an information system
Draft of lecture delivered to the Washington D C Chapter The Institute of Management Sciences Oct 18 1967 49 p

37 R G MILLS
Man-machine communication and problem solving
Annual review of information science and technology Vol 2 Ed C A Cuadra pp 223-254 Interscience Pub New York 1967

38 T MARILL L G ROBERTS
Toward a cooperative network of time shared computers
AFIPS Proc Fall Joint Computer Conf Vol 29 San Francisco Calif Nov 7-10 1966 pp 425-431 Spartan Books Washington D C 1966

39 G T UBER P E WILLIAMS B L HISEY R G SIEKERT
The organization and formatting of hierarchical displays for the on-line input of data
AFIPS Proc Fall Joint Computer Conf Vol 33 Pt 1 San Francisco Calif Dec 9-11 1968 pp 219-226 Thompson Book Co Washington D C 1968

40 G W BROWN J G MILLER T A KEENAN Eds
EDUNET
Report of the Summer study on Information Networks Conducted by the Interuniversity Communications Council (EDUCOM) 440 p Wiley New York 1967

41 A M CAIN I H PIZER
The SUNY biomedical communication network: implementation of an on-line, real-time, user-oriented system
Levels of interaction between man and information Proc Am Doc Inst Annual Meeting Vol 4 New York N Y Oct 22-27 1967 pp 258-262 Thompson Book Co Washington D C 1967

42 D J PRICE
Some theories about scientific papers and their authors
Proc Sixth Annual Meeting National Federation of Science Abstracting and Indexing Services Washington D C Mar 20-22 1963 Ed R A Jensen pp 47-56 Pub by NFSAIS Washington D C Nov 1963

43 A M HORMANN
Planning by man-machine synergism a characterization of processes and environment
Rept No SP-3484/000/00 83 p
System Development Corp Santa Monica Calif Mar 31 1970

44 G SALTON
Automatic information organization and retrieval
514 p McGraw-Hill Inc New York 1968

Development of the LOGICON 2 + 2 system

by ALBERT L. DEAN, JR.

LOGICON, Inc.
San Diego, California

INTRODUCTION

LOGICON is a systems and software engineering firm that has been heavily involved during the past nine years in developing special systems and software to be used primarily for military and space flight applications. In March of 1969, LOGICON began to investigate the feasibility of using a time-sharing system in its operations. Because a large portion of LOGICON personnel were involved in software development, it was felt that a time-sharing system used as a program development and documentation environment could be of significant aid to these activities.

An examination of other LOGICON operations revealed that many activities such as contract administration, project planning and control, personnel management, financial planning, and marketing could also make effective use of a time-sharing system. Not surprising was the fact that many of LOGICON's business activities are very similar to activities of firms of the same size but operating entirely different businesses. A key capability required by nearly all of these activities was a need to interact with files and data bases while performing concurrent computational operations.

A survey of time-sharing systems available indicated that none of these systems really provided interactive file and data-base management capabilities. This result seemed to indicate the need for such a product; therefore, LOGICON embarked upon the development of a time-sharing system called the LOGICON 2+2 System.

The 2+2 System has been developed specifically to provide very powerful computational, file, and data-base management capabilities on an interactive basis. The 2+2 System is unique—not only because of its file and data-base management capabilities, but also because it was developed for the most part with "off the shelf" fixed-wire hardware, microprogrammed hardware and software designs and/or components. These components have been organized to form an effective environment in which processes may be run in a multiplexed fashion for many simultaneous users. Each user has access to a virtual machine which executes one or more processes in his behalf free from undesired interaction with other users.

This paper describes an analysis of the LOGICON 2+2 System mission, the system requirements derived from this analysis, the development methods employed in creating the system, and the operational and organizational characteristics of the resulting system.

SYSTEM MISSION ANALYSIS

The mission analysis for the LOGICON 2+2 System began with a delineation of system objectives. Four major objectives were defined for the 2+2 System. The 2+2 System must:

- provide capabilities that will allow it to be used as a software development facility by LOGICON in its various system development activities.
- be capable of supporting a variety of applications in the general business environment, such that it can be used by LOGICON or other business firms as an in-house general purpose system or as a vehicle for supplying time-shared services to outside users.
- be capable of supporting a wide class of users, from the novice user to expert programmer, over a performance spectrum ranging from interactive to batch processing.
- be organized in a way that will allow it to be easily modified or extended so that it can be used to form a variety of special purpose systems that may be useful as products.

Given the above objectives, a mission analysis was developed for the 2+2 System. This mission analysis was broken into three major categories: (1) marketing, (2) application, and (3) user requirements.

Marketing requirements

The marketing requirements were developed according to type of customer. Three types of potential customers were identified for the **2+2** System. These are:

- Customers who need one or more general purpose computer systems with interactive shared-access capabilities to meet in-house information processing requirements.
- Customers who are or are planning to provide and market interactive remote access computer services in one or more application areas.
- Customers who need, either for themselves or as contractors to other organizations, a general purpose system that can be easily specialized to form the primary component of a dedicated application system.

An analysis of the potential market represented by these customers produced two major results.

First, a number of potential **2+2** System customers were currently subscribing or were considering subscribing to a time sharing service to meet their interactive processing needs. If the **2+2** System was to replace these services through in-house installation or to replace the equipment providing the service, it must meet two basic requirements: (1) the **2+2** System had to be able to demonstrate a significant cost/performance advantage over existing services and systems, and (2) the facilities provided by the **2+2** System must be compatible with the facilities offered by existing systems (at least on a subset basis), while providing increased capabilities.

Second, five major areas of application were identified: (1) scientific computation, (2) general business data processing, (3) education, (4) data bank operation, and (5) specialty applications such as health and construction. The area of scientific computation, while the largest market, is currently becoming saturated, and as a result—very competitive. The general business data processing market is largely unserved and seems to have even greater potential than scientific computation, particularly in the sectors involving small manufacturing and large wholesaling. The education, data bank operation, and specialty application markets all have good potential but require significant development. As a result, it was decided that the **2+2** System must primarily provide those capabilities required to serve the scientific computation and general business data processing markets, and where possible, provide those facilities needed by the education, data bank, and specialty markets.

Application requirements

A study was carried out to identify the capabilities required to support the preceding areas of application. The results are summarized as follows:

Scientific computation covers the activities of desk calculation, statistical computation, engineering design, simulation, and general mathematics. To support scientific computation, the **2+2** System need not provide extensive data-base management capabilities. The software requirements for this application area are almost exclusively connected with the development of high level problem-oriented languages and their processors. Fast computational processors and high-speed batch, graphic display, and printing keyboard terminals constitute the major hardware requirements for this application.

General business data processing covers the traditional activities of accounting, payroll, inventory control, production control, and internal financial and sales analysis. This application area requires that its software provide extensive file and data-base management capabilities coupled with a set of application procedures for handling: payroll and labor distribution; order entry, shipping, billing, and sales statistics; purchase order preparation and accounts payable; inventory control, production control, and cost accounting.

The hardware required for this application consist of:

- keyboard printing terminals with print speeds from 10 to 50 characters per second for order entry, billing, and small report preparation.
- graphic display terminals for data entry and inquiry.
- remote batch terminals for high-speed data input and output.
- mass storage devices with extensive capacity for files and their associated data bases.

Data bank operation covers the application of an operator-established and maintained data base that is referenced by his customers on a "read only" basis to secure some item of information such as credit ratings, checking account balances, stock quotations, etc. The software required for this application also involves extensive file and data management capabilities. The hardware requirements are equivalent to those of general business data processing, except that special terminals are probably required.

Education covers the application areas of computer aided instruction, test scoring, student counseling, and

reference retrieval. The software requirements for this area include extensive data-base management functions and simple interactive program preparation facilities to allow the user to easily construct course presentations, aptitude tests, and reference inquiries. The hardware required is again similar to that needed for general business data processing, with the addition of specialized terminals for instruction.

Specialty applications include areas typified by a highly specialized orientation to work functions unique to a particular occupation. The software requirements for the construction industry include the need for PERT, cost forecasting, and accounting packages that utilize a data base for required parameters. In the medical field, software packages are required to accommodate patient management, inventories of drugs, personnel scheduling, patient billing, clinical history maintenance, insurance coverage, and patient/bed scheduling. Hardware requirements can be met with current equipment, with the exception that special terminals will probably be needed.

User requirements

User requirements for the **2+2** System were developed according to user types, functions, and system loading.

Types of users

The number and types of computer users are expanding at a high rate. To provide facilities that will accommodate all of these users, it is necessary to categorize them into a few major categories.

Novice. A novice user produces no programs; he simply supplies parameters to programs produced by others. Included in this category are clerks, secretaries, foremen, librarians, teachers, and managers. They will need no knowledge of the application, and in fact—they will resist learning to use a system if it significantly detracts or absorbs time from their primary work assignments.

Technician. A technician is a professional in some field other than programming, such as engineering, finance, marketing, manufacturing, etc. The term technician is used in the dictionary sense ("a specialist in the technical details of an occupation") rather than in the more limited sense. He prepares programs to solve his own problems or those of a novice. He is not aware of the hardware features or capabilities. In general, he uses only a small sub-set of the hardware and software features of the system. He is aware of only a "virtual machine" much like the current BASIC systems; with additional facilities for report composition, text editing, and data-base management.

Programmer. A programmer produces programs to serve the computational or data processing needs of an organization. He programs largely in standard procedural languages such as COBOL, FORTRAN, ALGOL, PL/1 and RPG. He will, on occasion, use special system development languages. He uses the software supplied by the system manufacturer, which includes job control facilities, sort, and general utility programs. The fundamental system software commands represent his interface with the system.

User Functions

In order to further identify the **2+2** System requirements, a description was prepared the way each type of user would use the system. The following represents a summary of this analysis.

Novice. The novice user interfaces with the system on an interactive level that permits him to select and control the operation of the system as his commands are being executed. The kinds of functions utilized by the novice user include data entry, inquiry, display, report generation, and function invocation; and in some cases, file and data-base definition.

Technician. The technician works at a programming level that will provide the functions necessary to write programs that then become new operations at the interactive level. The kinds of functions used by the technician include file and data-base definition, data entry, procedure and process definition, editing, and simple program debugging.

Programmer. The programmer works at a defining level that will allow the creation of new commands and programming facilities for use at the novice and technician levels. To accommodate these activities, the system must provide the facilities needed to: define hierarchies of named files; assign access attributes to allow controlled sharing of files; enter programs, data, and text into files; translate source programs to form executable code; test the resulting object programs and insert these programs into the system to form new commands and programming facilities.

User loading

To provide some basic guidelines for developing the **2+2** System, it was essential to estimate the system load produced by the various users. The system loading, as given below, is based upon data from current systems and includes the load from all interaction types.

TABLE I—User Attended Terminal Load per Terminal

Load Component	Application Type		
	Computational Time Sharing	Remote Data Processing	Inquiry
Processing instructions per second	3,500	3,500	250
Data-base accesses per second	.14	.226	.25
Instructions per data-base access	25,000	15,500	1,000
Disc type data-base capacity (megabytes)	1	1	1
Archival type data-base capacity (megabytes)	10	10	10

User-attended terminal load. The load that a user applies to a computer system is heavily influenced by the logistics and scheduling of jobs to and from the computer system. This load is essentially independent from the speed of the user's terminal. The user adjusts his output requirements so that the time he spends waiting for output is tolerable. In current systems with 10 cps teletypes, about 35 to 40 percent of the user's total terminal time is spent in printing for all purposes.

Loads for three different application types are given in Table I. The processor load value in instructions per second, per terminal, is essentially independent of computer speed and compiler software efficiency. It can be shown that this processing load is primarily determined by the scheduling parameters used by the system. For systems having the same response time, those with larger quanta will serve fewer users and do more processing per user than those with smaller quanta. The value of 3,500 instructions per second, per terminal (Table I), is typical of systems designed to do user attended terminal work.

The data-base access load is expressed in terms of processor instructions executed per each access, and it is independent of any scheduling influences. The data-base load expressed in accesses per second follows the processor load instructions per second, as the scheduling method is changed.

Batch and remote batch load. If the user is not at a terminal waiting for results while his job is being run, that job is called a batch job. The chief difference between batch work and user attended terminal work is that the user does not receive his results immediately and he may not submit another job until after he has received and studied his results. The man-machine interaction cycle is much longer than with a user attended terminal, and the rate at which users submit successive jobs is much less dependent upon the length of that user's job.

User jobs are done in a batch mode when they are too long to wait for in a user attended terminal mode or when they require too much output for an individual user terminal and a faster, shared, batch terminal is required.

Table II shows some typical characteristics of a batch or remote batch system load.

SYSTEM REQUIREMENTS

Using the preceding analysis of the LOGICON **2+2** System mission, requirements for the basic organization of the system and its hardware and software components were developed.

Hardware requirements

The **2+2** hardware system was conceived to consist of four major components (1) an application subsystem, (2) an extended memory subsystem, (3) a communications—I/O subsystem, and (4) a control subsystem.

Application subsystem

The application subsystem was defined to consist of an arithmetic processor connected to a high-speed memory. The requirements developed for the applica-

TABLE II—Batch and Remote Batch Load Characteristics

Characteristic	Quantity
Mean executed processor instructions for one program execution (activity)	35 million
Mean lines-of output for one program execution (on-line high-speed printer)	2,000
Mean executed instructions per data-base access	15,500
Mean executed instructions per line of output (on-line high-speed printer)	17,500
Remote batch-restricted output, executed instructions per line of output	35,000

tion subsystem were:

- that it be able to sustain an instruction rate of 3500–7000 instructions per second, per active user.
- the processor should be microprogrammed to allow specialization as needed for a given application area.
- it must be able to support fragmentation of programs and data, through either paging or segmentation, and further provide access control over these program fragments.
- the processor should provide extensive multi-level interrupt capabilities.

Extended memory subsystem

The extended memory subsystem was defined to consist of a drum storage unit and a controller connected directly to the application subsystem's memory. The requirements derived for the extended memory subsystem were:

- the drum storage unit must have a capacity equal to at least 10 times the amount of virtual memory allowed for each user.
- when the latency and transmission rate for a block of data is considered, the "average" access time for a word in the extended memory subsystem should be no greater than 10 times the cycle-time for application memory.

Communications—I/O subsystem

The communications—I/O was defined to consist of a processor with communications and peripheral adapters connected directly both to the high-speed memory of the application subsystem and to its own high-speed working memory. Attached to the communications—I/O subsystem are a variety of peripherals and communication devices. The requirements developed for the communications—I/O subsystem were:

- it must be capable of supporting multiple communication lines, either duplex or half-duplex, with bandwidths ranging from 110 to 300 baud for the low-speed lines and 2,000 to 19,200 baud for the high-speed lines.
- all device connections for the low-speed devices on the communications—I/O subsystem (including the low-speed peripherals) must conform to EIA RS 232 interface standards.
- it must support ASCII codes.
- the processor should be micro-programmed and have sufficient speed to allow line management and device control to be accomplished by the processor, versus special device adapters.
- it must be capable of storing .5 million bytes of information on-line, per user of the system.
- when the seek time and transmission rate are considered, the "average" access time for a word in the disk storage units should be no greater than 10 times the access-time of the extended memory subsystem.
- at least some portion of the disc storage capacity should exist on removable media.

Control subsystem

The control subsystem was defined to consist of a processor connected to the communication—I/O memory and the application memory. The requirements derived for the control subsystem were:

- it must be connected to the other subsystems via interrupt and acknowledge lines to form a control network within the system.
- it must be microprogrammed to allow specialization as required for its control functions.

Software requirements

Three major requirements were defined for the **2+2** software. The software must:

- be based upon a proven design.
- when integrated with the hardware, provide users with a uniform environment free of hardware idiosyncrasies.
- provide a graduated set of capabilities in response to different types of users.

With these requirements in mind, the **2+2** software was conceived to consist of two major systems; the operating system and the application system.

The operating system

Requirements developed for the operating system specified that it must:

- be organized in a modular fashion to facilitate modification or extension as required.

- allow for a distributed type of operation where a user can select, use, and pay only for those modules required by his job.

The operating system was defined to consist of four major components: (1) the file manager, (2) the I/O manager, (3) the process manager, and (4) the data-base manager.

The file manager must provide the capabilities required to:

- form hierarchies of named files.
- allow access attributes to be assigned to these files, and enforce these access definitions.
- enter or extract information in or from these files.
- enable file backup and recovery in the event of a failure.

The I/O manager must provide the facilities required to:

- control the communications and peripheral devices of the system.
- support two modes of data access; random and sequential.
- provide the functions of read, rewrite, and append; where rewrite refers to the ability to write over existing data, and append refers to writing on an "unwritten" portion of data.
- supply format control over both physical and logical streams of data; where physical format is dictated by the device, and logical refers to a user supplied structuring.

The process manager must provide the capabilities required to:

- create processes consisting of procedures and data.
- allocate resources required to execute these processes.
- schedule the execution of processes, both system and user.
- monitor the execution of processes.
- maintain an accounting of the usage of the system resources by processes.

The data-base manager must supply the capabilities required to:

- define multi-level data structures and access attributes.
- associate defined data with one of four access methods; sequential, indexed-sequential, linked list, or multi-list as defined for a designated file.

The applications system

Requirements defined for the application system were that it must:

- provide a uniform interface to the users of the **2+2**; i.e., interactive commands and batch control statements for equivalent functions are the same.
- provide "total" functional capability in each of its subsystems so that all the facilities required by a user for the solution of his problems are available to him with the subsystem.
- provide a graduated set of capabilities that match the need of the various types of users; novice, technician, programmer, operator, and "owner."

The application system was defined to consist of three major subsystems or classes of subsystems; an executive, language subsystems, and application library.

The executive subsystem must provide the facilities needed to:

- establish an interface between the user and the system in the form of a command language and command language interpreter.
- interpret user commands and invoke the appropriate system functions.
- allow for the modification and extension of the command language.

The language subsystems were divided into three categories, corresponding to type of user: (1) novice, (2) technician, and (3) programmer.

The language subsystems supplied for the novice user must:

- operate on an interactive command level basis.
- provide the functions required to do data entry, inquiry, display, and updating of data bases.
- provide the functions to carry out simple desk computations using formatted forms and reports.

The language subsystems supplied for the technician must:

- provide the functions needed to accomplish file and data-base definition, data entry, procedure

and process definition, editing, and simple program debugging.

The language subsystems supplied for the programmer must allow him to:

- define hierarchies of named files.
- assign access attributes to allow controlled sharing of files.
- enter information into these files consisting of programs, data and text.
- translate source programs to form files of executable code.
- select, edit, merge, and link files to form processes that are ready for execution.
- load linked processes and invoke execution.
- monitor and retrieve the results of an executed process.
- select, edit, and format source language, flowchart, and text files to form the documentation for a new command or system.

The application library must provide a series of scientific, engineering, and general business packages. In particular for business—these packages should be concerned with payroll, inventory control, accounting, production control, and internal financial and sales analysis.

DEVELOPMENT METHODS

The development methods employed for the LOGICON **2+2** System merit discussion for several reasons. *First*, while a number of time-sharing system descriptions have been published, only fragmentary information has been presented concerning the methods used to develop these systems. *Second*, the methods used to develop the **2+2** System represent a systematic approach to the construction of a time-sharing system.

The discussion of the **2+2** System development is divided into two sections. The first section describes the general development scheme, while the second section describes the development tools utilized to produce the system.

General development scheme

The general scheme used for development of the **2+2** System can be called an "outside-in to inside-out" approach. This process is illustrated by Figure 1.

The notion of outside-in to inside-out development is analogous to the evaluation of an expression, in that one starts at the outside and proceeds inward identifying the primary operands and operators and their ordering. Once the primary operands are identified, the process is reversed and the constituents of an expression are combined, according to the applicable operator rules, to produce the value defined by the expression. Similarly, the development of the **2+2** System proceeded from the "outside-in to inside-out."

Outside-in

The outside-in path is divided into two phases; specification and design. During the specification phase, a system requirements specification was produced that identified the types of service to be supported by the system, the classes of users to be served, the functions to be supplied to these users, and the possible major components required to provide these functions. Using the system requirements specification as a guide, the functional, organizational, and operational characteristics of the **2+2** System were defined. From these definitions, a system architecture specification was developed.

Given the system architecture specification, effort was then directed toward producing detailed designs for subsystems and their major components as called out in the system architecture specification. It should be made clear, if not already so, that one does not proceed directly through the transformation of requirements—to the architecture—to the detailed design—without some iteration and intuitive knowledge of the feasibility of the resulting object.

Inside-out

At this point, the detailed designs for each subsystem and major component were transformed into executable

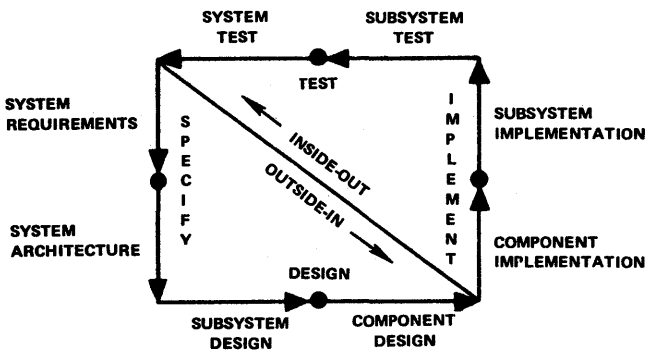


Figure 1—Outside-in to inside-out approach

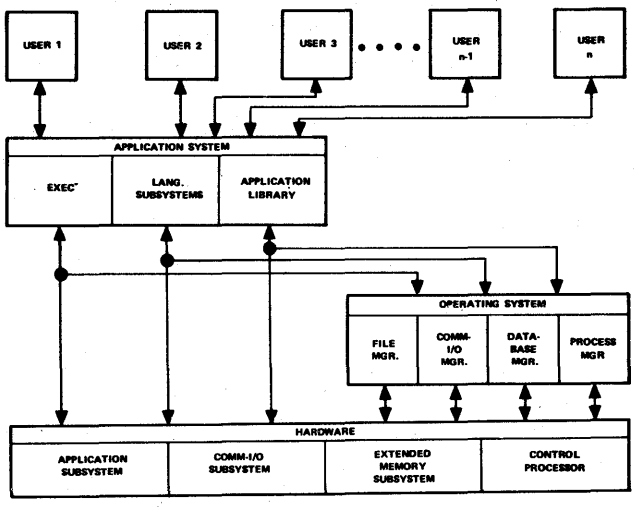


Figure 2—2+2 system logical structure

hardware/software logic. This transformation was accomplished by translating the design for each module of a component or subsystem with the implementation language appropriate to that module; i.e., wire-wrap lists, microprogram assembly language, or system assembly language.

As each module was implemented, an effort was made to ascertain that a module met the functional and operational requirements defined for it. This was accomplished by building a skeleton for the next higher structure in which the component was to reside; and then, supplying the appropriate input parameters and calls to the module, verifying the correctness of the output parameters and returns, and measuring critical execution time and frequency of use. This process was repeated at successive levels of integration until a complete subsystem was developed. The subsystems were then tested, and upon successful test completion, integrated into the system. As each subsystem was integrated into the system, an iterative testing procedure was applied. This procedure basically started at the innermost modules, insuring their continued integrity, and proceeded outward until the newly integrated modules or subsystems were checked.

Development tools

The tools utilized to develop the **2+2** System may be grouped into two categories; support system facilities and target system facilities.

The *support system facilities* were established on a time-sharing service system on which LOGICON purchased time. The facilities provided by the support system consisted of a simulator for the target machine; an assembler, to produce code for the target machine or its simulator; a program debugging package running under the simulator, for aiding in the checkout of system software; and the standard file and editing facilities of the time-sharing system on which the support system resides.

The *target system facilities* consisted of a simple run-time executive with a simple dump and trace capability for loading, executing, and monitoring the execution of software modules developed for the **2+2** System. In addition, a simple file system was developed to allow modules to be placed on the **2+2** System mass storage. Finally, a linking loader was also placed in the development executive for aiding in the generation of system programs.

SYSTEM DESCRIPTION

The LOGICON **2+2** System is composed of six major components: (1) the application subsystem, (2) the extended memory subsystem, (3) the communication—I/O subsystem, (4) the control subsystem, (5) the operating system, and (6) the application system. The logical organization of the LOGICON **2+2** System is illustrated in Figure 2.

These six components have been integrated to form an environment containing the facilities necessary to concurrently process a wide range of user applications. This concurrent processing is accomplished by multiplexing the system's facilities against the requirements of the system's users. In addition, these facilities have been regularized to eliminate hardware inconsistencies. As a result, each user is provided with a virtual machine containing the facilities required for his application, free from undesired interaction with other users and hardware idiosyncrasies.

The fundamental unit of user activity in the system is a process. A process is a related sequence of operations executable on the processor(s) comprising the virtual machine. These operations are executed in one of two modes; user and system. User mode operations are executed directly while system mode operations are indirectly executed. System mode operations are hardware instructions or extensions to these hardware instructions implemented as software procedures, which require protected and controlled execution in order to assure effective and uniform usage.

To utilize the **2+2** System, a user sends a connect signal to system via his terminal device. The system responds by establishing a process called the executive for the user. The executive first validates the user's access to the system and then operates as an interface

to the user, interpreting the users commands and invoking system functions in response to these commands.

Through commands, the user is able to create and cause the execution of one or more processes on behalf of his application. These processes may be system supplied as in the case of the language subsystems or user supplied applications or combinations of both.

In the remaining portion of this paper, the components comprising the **2+2** System are described in more detail.

The **2+2** *hardware*

The **2+2** hardware is composed of four major subsystems: (1) the application subsystem, (2) the communications—I/O subsystem, (3) the extended memory subsystem, and (4) the control subsystem. The physical organization of the **2+2** System is illustrated in Figure 3.

Application subsystem

The application subsystem has been constructed using a microprogrammed processor augmented with a Virtual Address Translator (designed by LOGICON) and a 900 nano-second, 16-bit word core memory ranging from 32 to 64 thousand words. The instruction set for the application processor was developed by LOGICON specifically for the **2+2** System using the microprogramming capability of the processor.

The Virtual Address Translator, or VAT, allows virtual addresses consisting of a page number and offset to be translated into physical addresses. The core memory provides for four-way interlace and page protection facilities of read, write, and execute. The instruction execution rate of the application subsystem is approximately .5 million instructions per second, which equates to a capability of supporting between 64 to 128 users—depending upon core memory size.

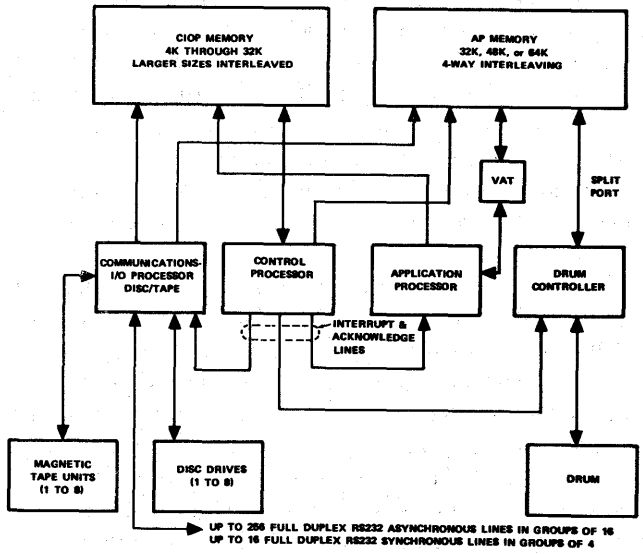


Figure 3—**2+2** physical organization

The communication—I/O subsystem

The communications—I/O subsystem was also constructed with a microprogrammed processor. This processor is connected to the application subsystem's memory and has its own high-speed working memory of 900 nano-seconds; a 16-bit word core memory, ranging in size from 8 to 32 thousand words; a set of asynchronous line adapters for 16 to 128 lines, in groups of 16; with bandwidths ranging from 110 to 300 baud; a set of synchronous line adapters for 4 to 16 lines, in groups of 4, with bandwidths of 2,000 to 19,000 baud; and peripheral adapters for the disk and magnetic tape storage devices. Normally, one synchronous line is used to attach a remote batch terminal consisting of a 300 cpm card reader and a 400 lpm line printer.

The disk storage units have removable media, 28 million-byte storage capacity, and an average access time of 45 milliseconds. Up to eight disk units can be attached to the mass storage controller.

Tape storage consists of from one to eight tape drives working 25 ips; 800 bps with 7 or 9 channel recording. Based upon the use of eight disk storage units, the **2+2** System can support 348 users having minimal storage requirements and up to 64 users requiring heavy storage needs. The "access-time" to disk storage is less than 10 times the access-time to the extended memory subsystem.

The extended memory subsystem

The extended memory subsystem for the **2+2** System was constructed using a drum having a 1 million word capacity, expandable to 4 million words, with a 1 million word-per-second transfer rate and an average access time of 8.5 milliseconds. The extended memory subsystem is connected to the application subsystem memory. The drum controller was designed and

developed by LOGICON to provide the required interface.

The control subsystem

The control subsystem was constructed using a microprogrammed processor. The instruction set for the control processor is almost identical to that of the application processor.

The control processor performs all system scheduling and I/O management tasks. It therefore controls the information flow between the application processor memory, extended memory, and mass storage. To accomplish these control functions, it is connected to the communications—I/O processor, the application processor, and the drum controller with interrupt and acknowledge lines. The control processor is connected to both the communications—I/O subsystem's and the applications subsystems's memory.

The 2+2 *operating system*

The 2+2 operating system is composed of four major subsystems: (1) the file subsystem, (2) the I/O subsystem, (3) the control subsystem, and (4) the data-base subsystem.

The file manager

Files in the 2+2 System are organized in a hierarchical fashion or tree structure of finite length (five levels) with a root directory origin. The structure contains pointers or entries for all nodes at the next lower level. These nodes may be directories themselves or point to some next level. The terminal nodes for the structure are files.

The file manager provides the operations needed to allow directories of files and files to be created, linked, modified, opened, closed, and deleted. The file manager consists of four major components: (1) the directory control module, (2) the access control module, (3) the storage mapping module, and (4) the backup and recovery module. The *directory control module* provides those functions required to create directories, create files and as a result create an entry in the appropriate directory, delete directory entries thereby deleting a file, and deleting directories. The *access control module* provides the capabilities needed to specify and enforce the use and access allowed for shared files. The access attributes provided are read, write, append, and execute. The *storage mapping module* maintains maps of all storage devices and the allocation of the space represented by these devices. The *backup and recovery module* provides the functions required to periodically dump files and to reestablish files in the event of a mishap.

The I/O manager

The I/O manager provides operations such as the reading and writing of sequential and random files and enabling the execution of physical input-output operations expressed in terms of logical parameters.

The I/O manager is organized into three major components: (1) the device control modules, (2) the device strategy modules, and (3) the I/O control module. The *device control modules* are particularized to manage a specific type of device and to control its operation, both in normal and error mode. The *device strategy modules* are responsible for the logical management of the device in terms of the data stream being transmitted to or from the device. The device strategy modules provide the function needed to access data in a random or sequential fashion, to format data, and for the logical operations of read, write, or append. The I/O *control module* is responsible for scheduling and managing the flow of data streams from one device to another, from one process to another, etc.

The process manager

The process manager provides the functions necessary to create, activate, block, and terminate processes, and also allocate resources and schedule their usage. In addition, it provides a mechanism for interprocess and interuser communication. It allows a process to create an "event," to be notified when an "event" occurs, and to delete "events." The process manager is composed of four major modules: (1) the scheduler, (2) the resource allocator, (3) process control, and (4) the accounting module.

The *scheduler module* is responsible for ordering the execution of processes in the 2+2 System. The scheduler accomplishes this ordering by maintaining a queue or process execution requests that are selected through a multi-level priority algorithm. The *resource allocator* provides the functions required to allocate and manage the resources of the 2+2 System. The *process control* is responsible for process creation, loading, execution, unloading, and destruction. During execution, the process control acts as the interface between the system and a user's process. The *accounting module* is responsible for metering, pricing, and billing for the facilities of the system used by its users.

The data-base manager

The data-base manager is responsible for the management of records in a data base and consists of three major modules: (1) the record access control module, (2) the record content module, and (3) the record format module. The record access control module is concerned with two types of access control. The first type of access control is involved with the logical operations of extracting or inserting a record in a data base through one of four access methods; sequential, indexed sequential, linked list, and multi-list. The second type of access control is involved with security conventions defined for a record, and insuring that these conventions are not violated.

The record content module is responsible for associating and maintaining lists of the elements that comprise a record. The record format module is responsible for providing the facilities for mapping contents of a record from one format to another for input-output operations. The record format module provides the facilities for associating a data base with the descriptions of the formats to be applied to that data base.

The 2+2 *application system*

The **2+2** application system is composed of three major components: (1) the executive, (2) the language subsystems, and (3) the application library.

The executive

The executive consists of three major elements: (1) a command language, (2) the command interpreter, and (3) the command library. The command language is the vehicle by which the user and the system communicate with each other. Through the use of the *command language*, the user is able to issue instructions to the system, directing it to perform some desired operation or set of operations. The *command language interpreter*, upon receipt of a command, scans the command and its arguments, calls the needed command subroutines from the command library, links these procedures, and transfers control to the linked procedures for execution. The *command library* contains a series of subroutines that enable the operation of the commands defined for the **2+2** System. Users of the **2+2** System have the capability to augment this command library through the use of commands supplied for that purpose by the system.

The language subsystems

The application system consists of a number of subsystems, each of which support a given language. In general, each language subsystem consists of a subsystem executive, command processor, language processor, and routine processor. There are three levels of language subsystems. Each level equates to the type of system user involved; novice, technician, or programmer.

For the novice user, a desk computation and data-base entry/inquiry command system has been developed called DESK-DATA, which combines the properties of a desk calculator and a simple data-base management system. The DESK-DATA language allows the novice user to establish data base, enter data into or make inquiries against the data bases, generate reports from these data bases whether on hard copy or a graphic display device, and define and perform simple computations on the data bases.

For the technician, three language subsystems are supplied: (1) an extended form of BASIC that provides extensive file and data-base management capabilities, (2) APL, also augmented with file and data-base facilities, and (3) a powerful text editing language.

For the programmer, FORTRAN IV, a subset of PL/1, and assembly language are provided.

The application library

The application library is composed of a large and growing collection of programs designed to perform explicit functions on behalf of its users. The application system is perhaps the most important component of the **2+2** System in the sense that it is directly concerned with the user's requirements.

SUMMARY

The **2+2** System is in the final stages of development and it is now clear that it will meet the functional and operational requirements set out for it. The success of this effort was in large part due to the project team involved. Special acknowledgment is given to R. E. Wolfe and John Mallory for the microcode design and development; Frank J. Rosbach, Allen Ginzburg, and Jan C. Bartlett for the design and development of the **2+2** Operating System; Glen S. Jerpseth, Robert A. Thompson, Paul K. Richards, Leland S. Purrier, and James A. Craig for design and development of the **2+2** Application system; and to George P. Futas and Donald F. Lacy for their design and development of the

Extended Memory subsystem. Special thanks are given to Frank C. Robbins for his aid in editing this paper.

REFERENCES

1 A BENSOUSSAN C T CLINGEN R C DALEY
The multics virtual memory
Second ACM Symposium on Operation System Principles Princeton New Jersey October 1969

2 R C DALEY J B DENNIS
Virtual memory processes and sharing in multics
First ACM Symposium on Operating System Principles Gatlinberg Tennessee October 1967

3 B W LAPSON
Scheduling and protection in interactive multi-processor systems
Project Genie Doc P-11 University of California Berkeley February 1967

4 B W LAMPSON
Time-sharing system reference manual
Project Genie Doc R-21 University of California Berkeley August 1966

5 B W LAMPSON W W LICHTENBERGER M W PIRTLE
A user machine in a time-sharing system
Proceedings IEEE December 1966

6 J H SALTZER
Traffic control in a multiplexed computer system
MAC-TR-30 (thesis) MIT Cambridge Massachusetts July 1966

7 M J SPIER E I ORGANICK
The multics interprocess communication facility
Second ACM Symposium on Operating System Principles Princeton New Jersey October 1969

System Ten—A new approach to multiprogramming

by R. V. DICKINSON and W. K. ORR

The Singer Company
San Leandro, California

INTRODUCTION

Historically, the creation of a multiprogramming computer system, one capable of concurrently operating a number of independent programs, has been viewed largely as a programming task, the task being that of creating an executive program which allocates resources, schedules tasks, and manages input/output. The problems involved in the implementation and operation of such programs have been well documented in the literature.[1]

System Ten, shown in Figure 1, embodies a new approach to the design of multiprogramming systems. The system, while capable of concurrently operating up to twenty independent programs, has no software executive. All executive functions are performed by hardware. This paper describes the architecture of System Ten with particular emphasis on the hardware executive and the system design decisions which made its implementation straightforward and inexpensive.

SYSTEM DESIGN PHILOSOPHY

As System Ten evolved several fundamental decisions were made to simplify the tasks of the hardware executive. These are briefly discussed below under the categories of resource allocation and system control.

Resource allocation

Main memory allocation is done on a fixed partition basis, that is, a contiguous area of fixed size is allotted to each user. Further, while the executive must be responsive to changes in memory allocation, it is not responsible for making these changes.

User device assignments are made through simple hardware connections. Since device assignments are generally made with a particular task, or class of tasks in mind, it is the responsibility of the user to respond to device reassignments.

The foregoing comments on device assignments pertain to peripheral devices such as card readers, card punches, printers, and operator oriented terminals. File storage devices such as magnetic tapes and discs are shared facilities, accessible to all partitions.

A fixed amount of processing time is allotted to each user program using a round-robin discipline.

System control

The user has complete control over the initiation of an IO operation; however, processing time is not again allotted to the user program until the system has completed the operation. Thus, an IO operation started in one partition is overlapped with processing in other partitions but not with processing in the partition issuing the IO instruction.

A supervisory console is not required by the system. The occurrence of a Program Check is handled by the executive through the simple expedient of requesting that the user load another program, generally an error analysis routine.

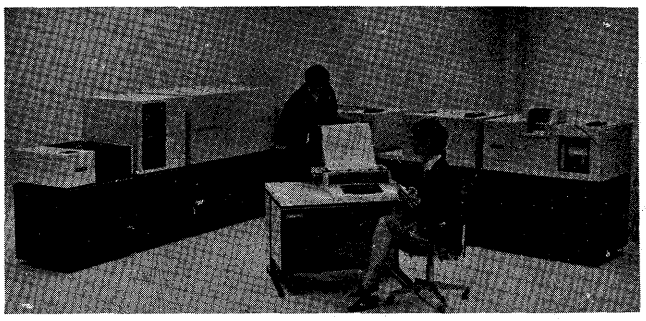


Figure 1—System Ten

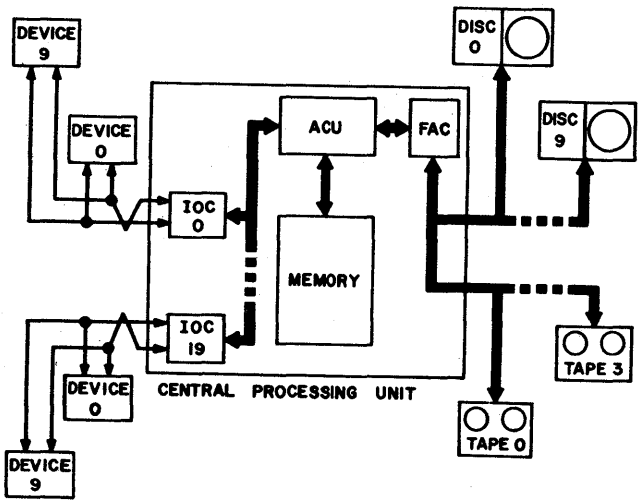


Figure 2—System structure

Those familiar with other systems offering multiprogramming facilities will recognize that the above decisions regarding resource allocation are very similar to those implemented in many executive programs. The significant difference is that reallocation of resources in System Ten is accomplished by changing a few simple hardware connections, whereas in other systems reallocation is accomplished by passing a "few simple" instructions to the executive program. Moreover, in System Ten, system generation consists of making the same hardware connections; in other systems it requires something quite different.

SYSTEM ORGANIZATION

System overview

The major components of System Ten are depicted in Figure 2. The Central Processing Unit (CPU) includes those components enclosed within the solid line. The peripherals and file storage devices are separately packaged. In contrast to many computer systems, the peripherals need not be located in close proximity to the CPU; instead, they may be distributed as dictated by application requirements.

All activity in the system is controlled by the Arithmetic/Control Unit (ACU) which includes a hardware multiprogramming executive. In addition to instruction fetch and execution, the ACU schedules tasks and manages input/output.

Memory is ferrite core with a 3.3 μs cycle time. It can be divided into as many as twenty partitions, each capable of supporting an independent user program. Unlike most multiprogramming systems, there is also a common partition of memory which can be shared by several user programs permitting the exchange of information at memory speed and the use of common subroutines. Maximum memory size is 110K (decimal) six bit characters. Each character in memory is addressable.

Associated with each user partition is an Input/Output Channel (IOC) which can control up to ten peripheral devices. These devices are accessible only to the associated partition. The peripherals and the IOC are interconnected with a single, shielded twisted pair which can be up to two thousand feet in length. Transmission between the peripherals and the IOC is bit serial, character serial at fifteen hundred characters/second.

System Ten peripherals include an operator oriented workstation with a 25 CPS printer and full alphanumeric keyboard, a 300 CPM card reader, a 100 CPM card punch, a 450 LPM line printer, a 300 CPS paper tape reader, and a 100 CPS paper tape punch.

Communications capability is provided by the Synchronous Communication Adapter (SCA) which can be substituted for a standard IOC. It enables the System Ten to communicate with remote terminals or computers at up to 1200 characters/second. The SCA can operate either on four-wire dedicated circuits or on switched circuits and has dialing and automatic answering capability. The SCA is designed to operate with the standard USASCII code and communication procedures.

The System Ten can also support up to ten disc drives and four magnetic tape drives. These devices are available to all user programs and are controlled by the File Access Channel (FAC). The FAC can accommodate data transfer rates of up to 250,000 characters/second.

Each disc drive has a storage capacity of ten million six bit characters stored in one hundred character records. Average access time is seventy-five milliseconds and average latency is 12.5 ms. A tape drive can accommodate either seven or nine channel tape with a density of either 556 or 800 bpi. USASCII, EBCDIC, and other standard codes can be accommodated. Tape speed is 25 ips.

Memory organization

Memory is allocated in one thousand character segments. Each partition, including common, may contain from one to ten segments. Shown in Fig. 3 is a maximum memory configuration with 10K characters allocated to the common partition and 5K characters allocated to each of twenty user partitions.

Note that a system with memory allocated as shown would also include twenty IOC's, one for each user partition, since the number of IOC's in a system determines the number of user partitions that the system will support. In fact, the amount of memory allocated to the partition associated with a given IOC is determined by a jumper block setting in the IOC itself. A minimum System Ten consists of 10K core, 1K of which must be allocated to common, and one IOC.

The common partition is divided into three areas. The first three hundred characters are reserved for the executive. User programs may read from but not write into this area. The executive uses this area to store the program status work (PSW), channel command word (CCW), and file access mask (FAM) for each user partition.

The PSW for a partition includes the partition size, program counter, condition codes, and status flag for that partition. The status flag identifies the currently active partition. The CCW contains the IOC operation code, the residual character count, and the current data address when an IOC operation is in progress for that partition.

The FAM defines the degree of file access permitted that partition. A partition may be denied all access, allowed access only through a subroutine resident in common, allowed read access, or allowed full access to the disc drives. Disc access control applies to all drives in the system. In contrast, tape access control is specified for each individual drive. A partition may be allowed access to any combination of the four tape drives. Write protection for the first area of common can be temporarily inhibited, permitting initialization of the FAM for each partition.

The second area of common is available to all user programs and is referred to as nonprivileged common. The third area, privileged common, is accessible only to those user partitions which have been designated (with a jumper on the associated IOC) as privileged partitions. Nonprivileged partitions may not access this area.

A base register and a limit register are used by the executive to establish boundaries for the currently active partition. Addressing within user partitions is relative to the base address of the partition. Figure 3 illustrates the relationship between absolute and relative memory addresses. A user program can address only that portion of memory within its own partition and the common partition.

Within each partition there are three four-character fields which can be used by the user program as index registers. A five-character field is reserved for storage of the PSW when a Program Check occurs.

SYSTEM OPERATIONS

Multiprogramming executive

The System Ten multiprogramming executive consists of four control routines implemented in hardware: Instruction Fetch, Partition Switch, IO Control, and Interrupt Service. Flow charts of these routines are shown in Figures 4a, 4b, 4c, and 4d.

The allocation of ACU time to user partitions is handled on a round-robin basis. Each partition is allotted 40 ms of processing time. When this interval is exhausted and a successful branch is executed processing time is then allotted to the next partition. This condition (branch timeout) is detected at the end of the Instruction Fetch routine as illustrated in Figure 4a. The steps involved in partition switching are shown in Figure 4b.

Partition switching also occurs whenever a partition initiates an IOC operation or if disc head motion is required for an FAC operation. In these instances, switching is initiated by the IO Control routine as shown in Figure 4c.

IOC interrupts are detected by the Instruction Fetch routine which transfers control to the Interrupt Service routine (cf Figure 4d). Interrupts are also detected and serviced during an FAC operation (cf Figure 4c).

Whenever a LOAD REQUEST is detected, whether initiated by the user or the ACU, the program counter is cleared and a READ CONTROL command is issued to the IOC. This enables entry of one instruction into location zero of the partition. When the READ CONTROL has been terminated, the instruction just entered is executed. This feature provides the capability to initiate program loading.

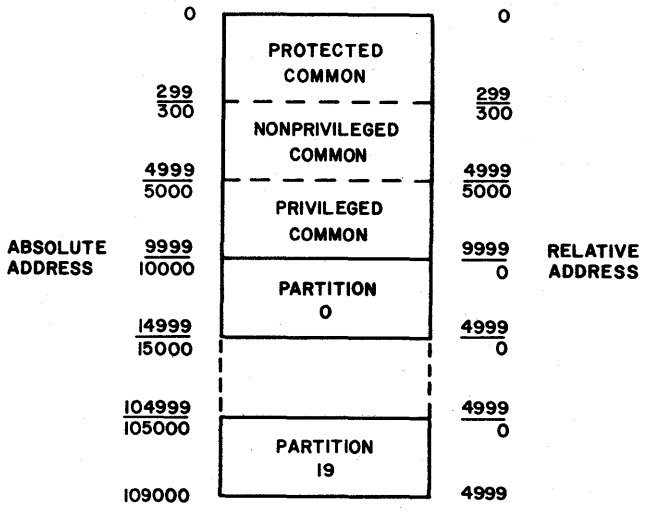


Figure 3—Memory organization

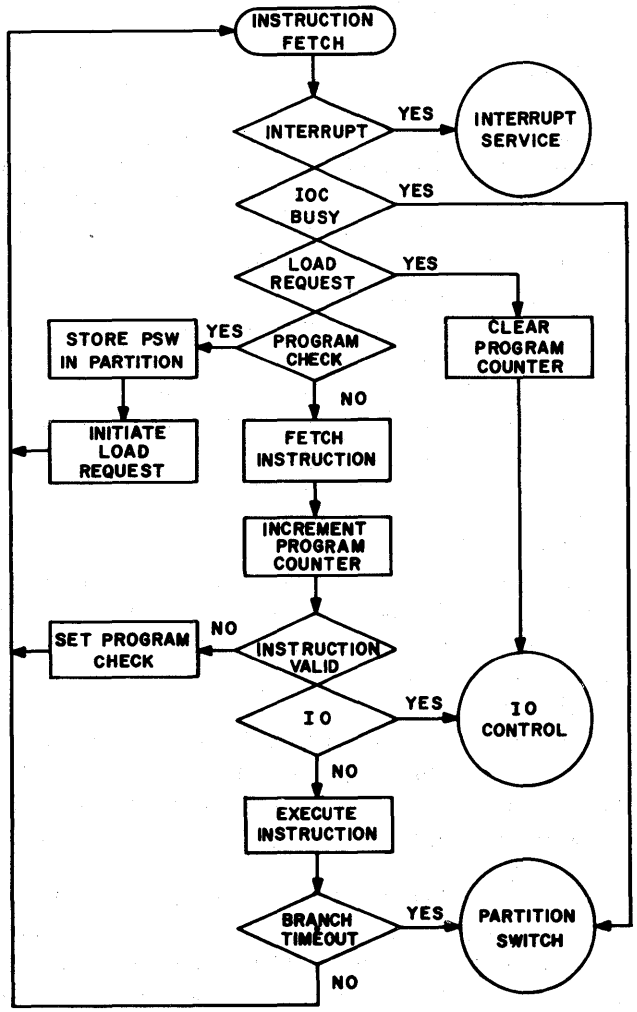


Figure 4a—Instruction fetch

The ACU initiates a LOAD REQUEST whenever it detects a Program Check. A Program Check can be caused either by an attempt to execute an invalid instruction or by an illegal memory reference. When a Program Check occurs, the PSW is stored within the partition.

The IO Control routine initiates all IOC and FAC operations and posts status following their completion. IOC operations are overlapped with processing in other partitions. FAC operations are not overlapped; processing is halted during an FAC operation in order to provide immediate response to FAC memory requests. IOC interrupts are serviced, however.

Access to FAC devices is granted on a first come, first served basis. If a disc seek is required, it is overlapped and the disc is unavailable to other partitions until the operation has been completed. Having once accessed a disc, a user partition is not allowed to initiate a seek until the other partitions are given an opportunity to access that disc. This prevents a partition from monopolizing a disc file.

User/system interaction

Interaction between a user and System Ten generally begins with program loading. To accomplish this, the user must assign address zero to an input device such as a card reader, paper tape reader, or workstation. Each input device is equipped with a LOAD switch; operation of this switch interrupts processing in his partition and permits him to enter one instruction. This instruction is loaded into location zero of his partition and executed the next time the partition is active.

Normally, this instruction is the first of a bootstrap loader. While any input device may be used for program loading, only the device assigned address zero can be used to initiate program loading.

There is one variation of the initial program load procedure described above. If the first character

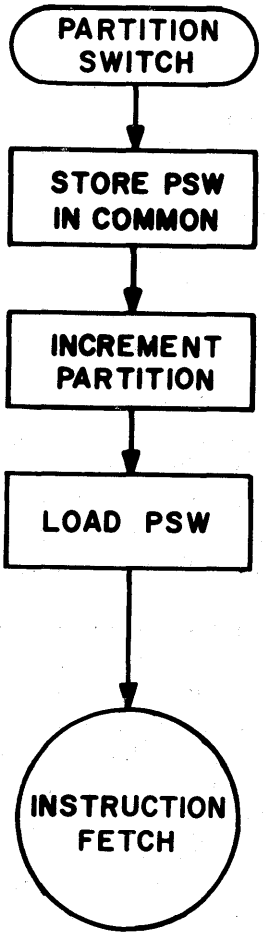


Figure 4b—Partition switch

entered is a unit separator, the input operation is terminated, and a disc read instruction is loaded into location zero. This instruction when executed will load the first 100 character sector from drive zero into the first 100 locations of his partition. The program thus loaded will generally be the first part of a disc oriented program loader.

When a program check occurs, the system will indicate this condition by lighting the LOAD lamp on the device zero. The user may then initiate loading of a new program. This program, generally an error analysis routine, can examine the PSW stored in the partition to determine the address of the offending instruction and perform other checks to determine what went amiss.

Online program debugging is most conveniently accomplished if device zero is a workstation. Using the program loading procedure described above, the user can enter one or more instructions to selectively change or display the contents of his partition thus monitoring the progress of his program.

In addition to the LOAD switch, the workstation has a SERVICE REQUEST switch. Operation of this switch sets a flag in the IOC which can be tested under program control, permitting the user to interact with his program without interrupting its execution.

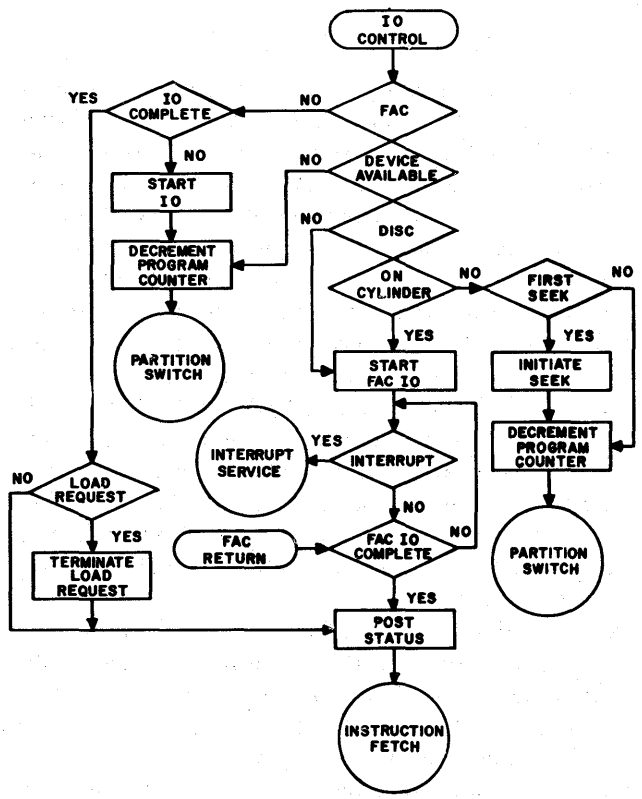


Figure 4c—IO control

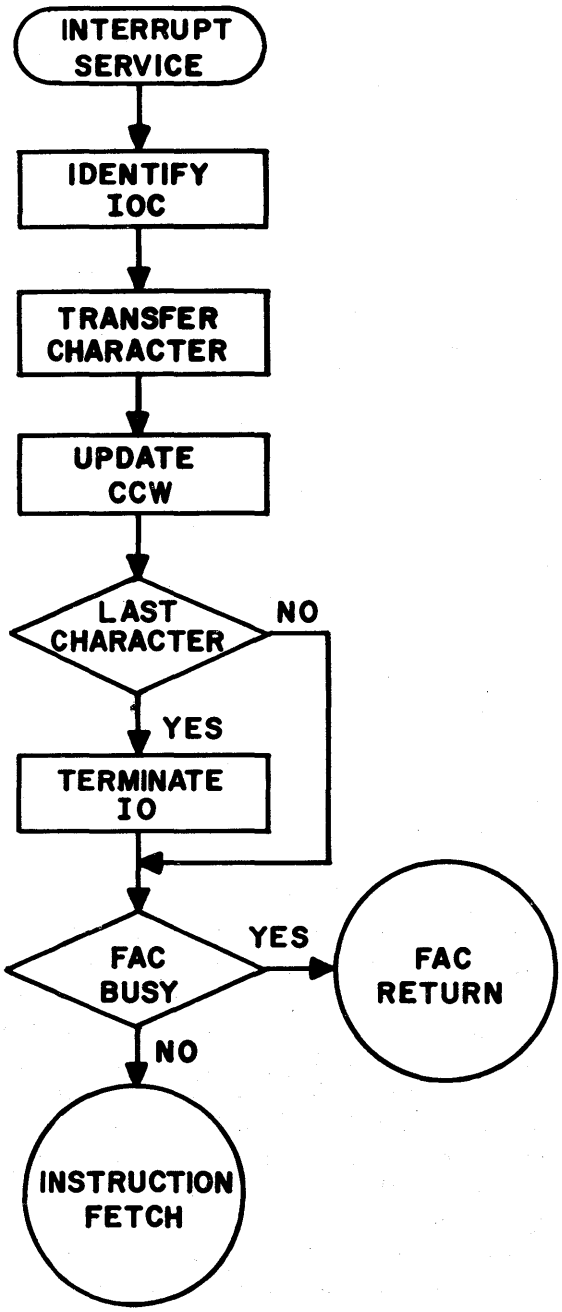


Figure 4d—Interrupt service

PROGRAMMING CONSIDERATIONS

The System Ten character set is a six bit subset of the USASCII character set obtained by omitting the sixth bit (b6) of the USASCII code. (See Figure 5). Although control characters (columns 0 and 1) cannot be stored internally, they can be generated by a WRITE CONTROL instruction for output.

System Ten instructions are designed to operate on fields of variable length, up to ten thousand charac-

TABLE I—System Ten Instruction Set

INSTRUCTION	OPERATIONS	REMARKS
ADD	(B)+(A)→(B)	
SUBTRACT	(B)−(A)→(B)	
MULTIPLY	(A)×(B)→(B)*	(B)* is a field of length LA+LB, whose most significant digit is addressed by B.
DIVIDE	(B)*÷(A)→(B)	See above
TRANSFER CHARACTER	(A)→(B)	The fields are of the same length, 10LA+LB.
TRANSFER NUMERIC	(A)→(B)	Only the numeric portion of (B) is changed. The fields are of the same length, 10LA+LB.
EXCHANGE	(B)⇄(A)	The A and B fields are exchanged. The fields are of the same length, 10LA+LB.
EDIT	(A)→(B)	The transfer is controlled by a *Mask* located at B.
FORM NUMERIC	(A)→(B)	The numeric information in the A field is right-aligned in the B field.
COMPARE		|(A)| is compared with the |(B)|.
BRANCH		A conditional branch is made to the instruction addressed by A or B.
READ		LA specifies the device address, LB selects the channel (IOC/FAC), A is the address of the first character in the input area and, B is the character count.
WRITE		Instruction fields are as defined for READ.
WRITE CONTROL		Same as write, except that characters appearing in columns 4 and 5 (figure *5*) are translated, and output as the control characters of columns 0 and 1.

ters in certain instances. The instructions, however, are of fixed length, namely, ten characters as illustrated by the instruction format shown in Figure 6.

System Ten is a decimal machine in that all arithmetic instructions operate on decimal fields. A decimal digit is stored in binary coded decimal form using the low order four bits (b1-b4) of a character. Operand addresses, the A and B fields of the instruction, and operand lengths, the LA and LB fields are also decimal quantities.

The AC and BC bits indicate whether the fields addressed by A and B respectively, are located in the common partition or in the user partition. Thus, a maximum of 20K characters of memory are available to the System Ten programmer, 10K in common and 10K in his partition.

The IA and IB fields of the instruction are used for index register selection. They are two bit binary fields; IA=0 indicates that the A address is not indexed, IA=1 indicates that the effective address is A plus the contents of the first index register, and so on.

The System Ten instruction set is given in Table 1. (A) denotes the contents of the field address by A.

When a branch instruction is encountered, control passes to the instruction addressed by A if the condition specified by LA is met. If this condition is not met the condition specified by LB is checked and if met, control passes to the instruction address by B, otherwise the next instruction in sequence is executed.

In addition to specifying conditions related to the outcome of arithmetic and input/output operations, the LA and LB fields may specify that a subroutine branch is to be taken or that a branch is to be taken when a device has a pending SERVICE REQUEST. In this latter case, the address of the device requesting service is stored at the location specified by B.

One form of unconditional branch allows the programmer to give up a portion of his allotted processing time. This is the branch and switch instruction. When this instruction is encountered, a branch is taken and partition switching occurs. For example, if a program is waiting for a request for service from a terminal, it can be designed to relinquish processing time to other partitions until the request occurs.

In disc input/output instructions, the B field is the address of a six character disc address rather than a character count. No character count is required as

disc records have a fixed length of one hundred characters.

A disc file has fifty records per track. Record addresses are interleaved by five so that there are four sectors between sequentially addressed sectors. This permits the programmer to modify disc addresses and do a limited amount of housekeeping between sequentially addressed sectors. Thus, random access file organizations which employ some form of scrambling can be implemented very efficiently. There is, however, a penalty when purely sequential access methods are used.

CONCLUSION

System Ten demonstrates the feasibility of providing multiprogramming capabilities, without the need for a complex software executive. Adoption of a system design philosophy oriented toward application requirements rather than unlimited generality made implementation of the hardware executive a straightforward and inexpensive task.

				b7	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1
b4	b3	b2	b1	Column / Row	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figure 5—Character set

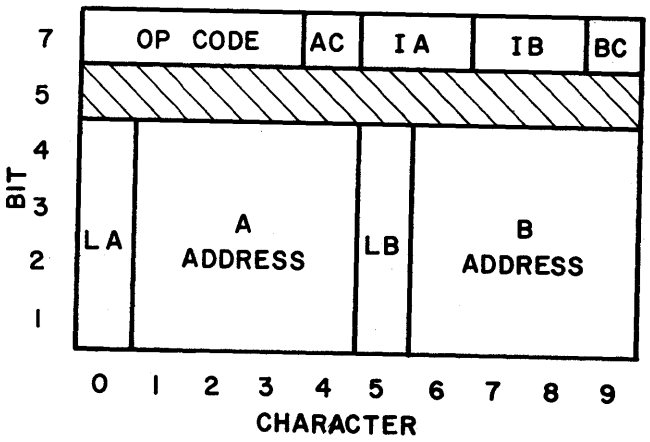


Figure 6—Instruction format

ACKNOWLEDGMENTS

We would like to thank the many individuals who contributed to the development of System Ten. Worthy of particular mention are D. Neilson, E. Poumakis, and H. Schaffer whose ideas provided the framework for the system as it exists today.

REFERENCES

1 T B STEEL JR
Multiprogramming—Promise, performance, and prospect
Proceedings FJCC Vol 33 p 99 1968

On automatic design of data organization

by WILLIAM A. McCUSKEY

Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

A number of research efforts have contributed to the beginning of a methodology for the automatic design of large-scale information processing systems (IPS). See for instance Nunamaker.[1] One facet of study in these efforts is the design of data organization.

Such a study was undertaken in the context of Project ISDOS,* now at the University of Michigan. The purpose of the study was to develop a model of the data organization design process and to create from this model a method for generating specifications of alternative data organizations. The first step of the study was to obtain a view of data organization uncomplicated by data usage. To this end the design of data organization (DOD) was divorced from the total IPS design process. A method for decision-making, which relates data organization to data usage and a measure of effectiveness, was to be a second phase of the study.

The purpose of this paper is to outline some initial results and implications of a set-theoretic approach to DOD which was developed for ISDOS. The assumed framework of the DOD process is described briefly. Within this framework concepts of data are defined in terms of sets. The DOD process can then be described in terms of set-theoretic operations. Finally some implications of the approach are given.

ORGANIZATION OF DATA—A FRAMEWORK

The term *data* is used here to mean the IPS representation of objects which are used as a basis for decision or calculation. The term *data organization* is used here to mean the set of relationships among data established by the problem definer or created by the system designer, as well as the representations of these relationships in the IPS. A *design of data organization* is a specification of these relationships, of their representations in the IPS, of the representation of the data in the IPS storage, and of the logical access and storage assignment processes which will operate on the data organization. The term *process* is used here to mean an operation or set of operations on data, whether that process is described by the problem definer or defined by the system designer. The system design procedure is itself a process and will be referred to as such.

The procedure for organizing data for an IPS may be thought of ideally in terms of four operations. First, a problem definer interprets a problem in his environment and defines a set of requirements which are as complete and concise as possible and which any solution of the problem, manual or automatic, must satisfy. A problem definition is complete if, in order to solve the problem, a system designer needs no further information from the problem definer. The problem definer defines the information processing problem in terms of sets of data, membership relationships among these sets of data, processes operating with the data, time and volume requirements on the processing, other constraints, and a measure of effectiveness for the solution. In order that the best possible design be produced, relative to the given measure of effectiveness, the problem definer should place as few restrictions as possible on the number of alternatives the system designer may consider.

Second, the system designer develops a specification of logically ordered structure for the data and the logical access processes which may be used to find any element in the structure. This structure will be called the logical organization of the data. An example is a binary tree, array, or any directed graph.

Third, the system designer specifies for these logically structured data the corresponding representations in the storage and the strategies for storage assignment. The resulting structure will be called physical organization of the data.

And fourth, the implementor of the system converts

* Information System Design and Optimization System

the actual data from its present form to a form which meets the new specifications.

Within this framework the approach was to view all concepts of data in terms of sets and then to define the design process, steps one through three above, in terms of set-theoretic operations on these sets. The set-theoretic details may be found in McCuskey.[2] The following attempts a more narrative description.

CONCEPTS

The concepts of data organization described below must be viewed in the context of an ideal automated design system such as ISDOS. The problem statement, written in a formal problem statement language, is input to a system design program. This program specifies how processes should be organized into programs, how data should be structured logically and physically, and how the programs and data should be managed as a complete system. The system design is then automatically implemented.

The goal of this description of data concepts is to provide a framework within which to formulate a precise, simple algorithm. The algorithm must operate on a problem definition of data to produce a specification of IPS storage organization for the actual data.

Because of this goal the sets of data which the problem definer describes are viewed here as set-theoretic sets related by unordered cross-product relations. The algorithm must then establish what redundancies to keep, specify how the data should be ordered and then specify how this logical structure should be represented in storage.

The goal requires that the distinction between logical and physical data organization be defined precisely. The logical structure discussed below is the structure which is directly represented in storage. It incorporates some features, like redundancy specification, which are generally considered in the realm of "storage structure".

Problem description

From the problem definer's point-of-view an IPS operates on symbolic representations of conceptual or physical characteristics such as name, age, address, etc. The elementary object used to build such IPS representations will be called a *symbol.* The problem definer must specify an alphabet, the set of all symbols which are valid for the problem he is defining. One such alphabet is the EBCDIC character set.

Each occurrence of a characteristic, such as sex, amount, or title, may be thought of as an ordered pair of symbol sequences. The first component of the pair is the *data name*; the second component is the *data value.* The ordered pair will be called, generically, a *data item.* A data item will be denoted by its associated data name. An instance of a data item is a specific data name/data value pair. Thus ⟨NAME, JONES⟩* is an instance of the data item NAME. Common usage abbreviates this statement to "JONES is an instance of NAME". A data item has sometimes been referred to as an attribute, data element, or datum. In common high-level programming language usage the data value is the "data" stored while the data name is "data about data" which appears in the source program and enters a symbol table during compilation.

From the problem definer's point-of-view the IPS at any point in time will contain representations of many different occurrences of a given characteristic, say warehouse number. Disregarding how warehouse numbers are associated with other data in the IPS, one can describe a set of all distinguishable instances of a data item, named WHNO, existing in the IPS at the given time and having the same data name. Instances are distinguished by data value. The set WHNO contains no repeated occurrences of warehouse number. Such a collection will be called a *data set at level* 0 (henceforth, data set/0). The data set is referenced, like a member data item, by the data name common to all its elements. Context determines whether a data name refers to a data item or a data set.

Associated with a data set/0 is a number, called the *cardinality* of the set, which specifies the anticipated number of elements (unique data item instances) in the data set. Among data sets/0 exist cardinality relationships such as:

> "at any given time approximately three unique instances of ITNO and exactly one unique instance of CITY will be associated with a unique instance of WHNO".

The anticipated cardinality and cardinality relationships among data sets, as defined here, are characteristics of the information processing problem and must be specified by the problem definer. The elements of a data set represent unique occurrences of an object, such as warehouse number, used in the problem as a basis for decision or calculation. What objects are used and how many unique occurrences of each must be represented in the IPS at any one time depend on how the problem definer interprets the problem.

These cardinality specifications eventually will help the system designer determine how much storage space

* A pair of right-angle brackets, ⟨ ⟩, will be used to indicate an *ordered n*-tuple (here a 2-tuple).

may be required for any data organization design which he considers.

The concept of data set may be extended to higher levels. Data sets/0 may be related by a named set membership association. The problem definer then describes processes in terms of operations on these associations as well as data items. For example, an updating process might be defined for INV (inventory) where INV is the data name associating the data items WHNO (warehouse number), ITNO (item number), and QTY (quantity). Nothing is said about ordering or logical structure on INV except the specification of set membership. In set-theoretic terms INV is a subset of the *unordered* cross-product of the three data sets/0. INV names the data set/1 (data set at level one), the next level above its highest level component.

Such set membership relationships may be visualized in a non-directed graph as a tree in which data set names are associated with vertices and dotted arcs represent membership relationships. A graphic representation of the data set/1 INV is given in Figure 1.

A data set/n (data set at level n) ($n \geq 1$) may be thought of as a set of (distinguishable) ordered pairs. Each ordered pair is unique within the data set/n. The first component of the pair is the data name of this data set/n. The second component of the pair is an unordered m-tuple. Each component of the unordered m-tuple is an element (itself an ordered pair) of a data set/j ($0 \leq j \leq n-1$). At least one component of the unordered m-tuple is from a data set/$(n-1)$. The term *data set component* refers to a component of this unordered m-tuple. A data set component is referenced by its data name. *Data set element* refers to a unique member element of the data set/n. *Component instance* refers to the instance of a data set component in a given data set element. Figure 2 gives an instance of the data set

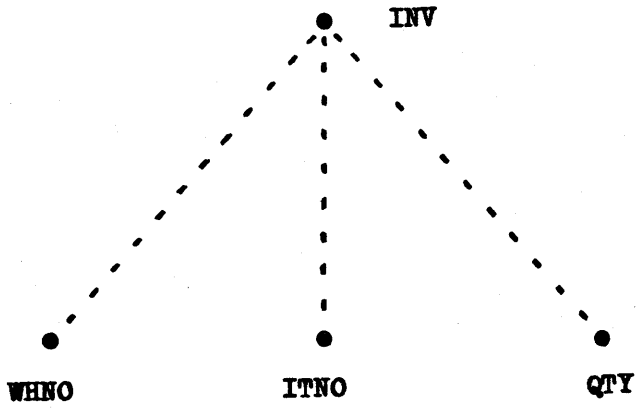


Figure 1—Graph representation of data set INV

⟨INV, { ⟨WHNO,3⟩ , ⟨ITNO,2⟩ , ⟨QTY,2⟩}⟩

⟨INV, { ⟨WHNO,1⟩ , ⟨ITNO,3⟩ , ⟨QTY,7⟩}⟩

⟨INV, { ⟨ITNO,1⟩ , ⟨WHNO,3⟩ , ⟨QTY,1⟩}⟩

⟨INV, { ⟨QTY,2⟩ , ⟨WHNO,2⟩ , ⟨ITNO,2⟩}⟩

⟨INV, { ⟨WHNO,3⟩ , ⟨ITNO,3⟩ , ⟨QTY,7⟩}⟩

Figure 2—Instance of data set INV

INV.* The data set contains five data set elements. The data set components are WHNO, ITNO, and QTY. ⟨WHNO, 3⟩ is a componentinstance of WHNO in three data set elements.

The concepts of cardinality and cardinality relationships, described above for data sets/0, are extended to data set/n. As with data sets/0 cardinality specifications for data sets/n must be given by the problem definer.

According to the above definitions a data set/n element is unique within its data set. However, multiple instances of the same data set element may appear as component instances in a data set at a higher level. In Figure 2 ⟨WHNO,3⟩ is a unique data set element of WHNO but is a component instance in three data set elements of the data set INV. This multiplicity of occurrence of the same data set element is referred to here as *redundancy*. The amount of redundancy—the multiplicity of occurrence of the same data set element —in a data set/n is determined by cardinality relationships among the component data sets, by the cardinality of each component data set, and by the association of data sets defined by the problem definer.

The design of logical data organization may be viewed as a specification of the amount of redundancy and ordering of data set elements and component instances. For the design process to consider as many alternative logical structures as possible, as little structure—redundancy reduction and ordering—should be implied by the problem definition. The above view of data sets admits as much redundancy and as little ordering as the problem definition can allow and still be complete and concise.

Logical data organization

The first problem for the system design process is to take a specification of these data sets and, by performing

* A pair of braces { }, will denote an unordered m-tuple.

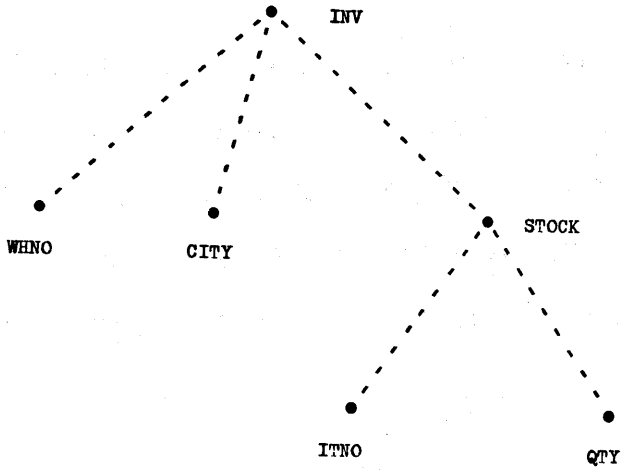


Figure 3—Graph representation of revised data set INV

a sequence of operations, obtain a specification of logical data organization for the data set. Logical structure is provided for two reasons. First, the logical structure maintains in some form the membership associations established and referred to by the problem definer in his problem statement. Second, the logical structure provides a path or sets of paths to any element of the structure. Logical access processes, for example binary search, depend on such paths.

The logical structure of data may be visualized as a directed graph and will be called a *data structure*. Each vertex of the graph represents either a data item or a data structure. A data item or data structure represented by a vertex will be called a *data structure component*. An arc of the graph then represents a logical ordering relationship between two data structure components. Such a directed arc is an ordered pair of data structure components and will be called a *connection*. The logical connection described here is the connection which will be represented directly in storage by a fixed or variable distance in the address space of the storage. A data structure can then be viewed as a set of connections—that is, a set of ordering relations among its data structure components. A series of contiguous connections, called a *logical access path*, may be formed between two data structure components. Logical access processes use these paths to access components in the structure. A specification of data structure is a pattern which when applied to an instance of a data set yields an instance of the given data structure.

Consider the data set INV, revised and described by the non-directed graph given in Figure 3. INV has been redefined to be a data set/2. An instance of data set INV is given in Figure 4. To avoid the confusion of multiple brackets, the depiction of the data set instance in Figure 4 omits the bracket symbols of Figure 2 and factors the data names to the heading of the figure. Each circled single data value represents a data item instance. Data set membership relationships are represented by bounding lines in the heading. Each entire row represents a data set element of INV. Each column represents instances of the specified data item. While a horizontal ordering of data items has been introduced in the figure for ease of reading, it must be remembered that this ordering is only artificial: the data set components WHNO, CITY and STOCK actually form an unordered 3-tuple and ITNO and QTY form an unordered 2-tuple.

In the development of a data structure from the data set INV the system designer might specify the connections ⟨WHNO,CITY⟩, ⟨CITY,STOCK⟩ and ⟨WHNO,STOCK⟩. Similarly the connections ⟨ITNO, QTY⟩ and ⟨QTY,ITNO⟩ might be specified within the data structure developed from STOCK. The data structure components of the data structure developed from INV are WHNO and CITY, which are data items, and STOCK which is itself a data structure. The structure indicated so far is depicted in Figure 5a. For convenience, INV and STOCK will temporarily be the names given to the data structures developed from the data sets INV and STOCK.

Consider now the connection from WHNO to STOCK. This connection creates an ambiguous reference because there are two data structure components in STOCK. If a logical access path is to be constructed from, say, WHNO to the data structure

INV			
		STOCK	
WHNO	CITY	ITNO	QTY
3	A	2	2
1	B	3	7
3	A	1	1
2	C	2	2
3	A	3	7
1	B	2	4
2	C	1	2

Figure 4—Instance of revised data set INV

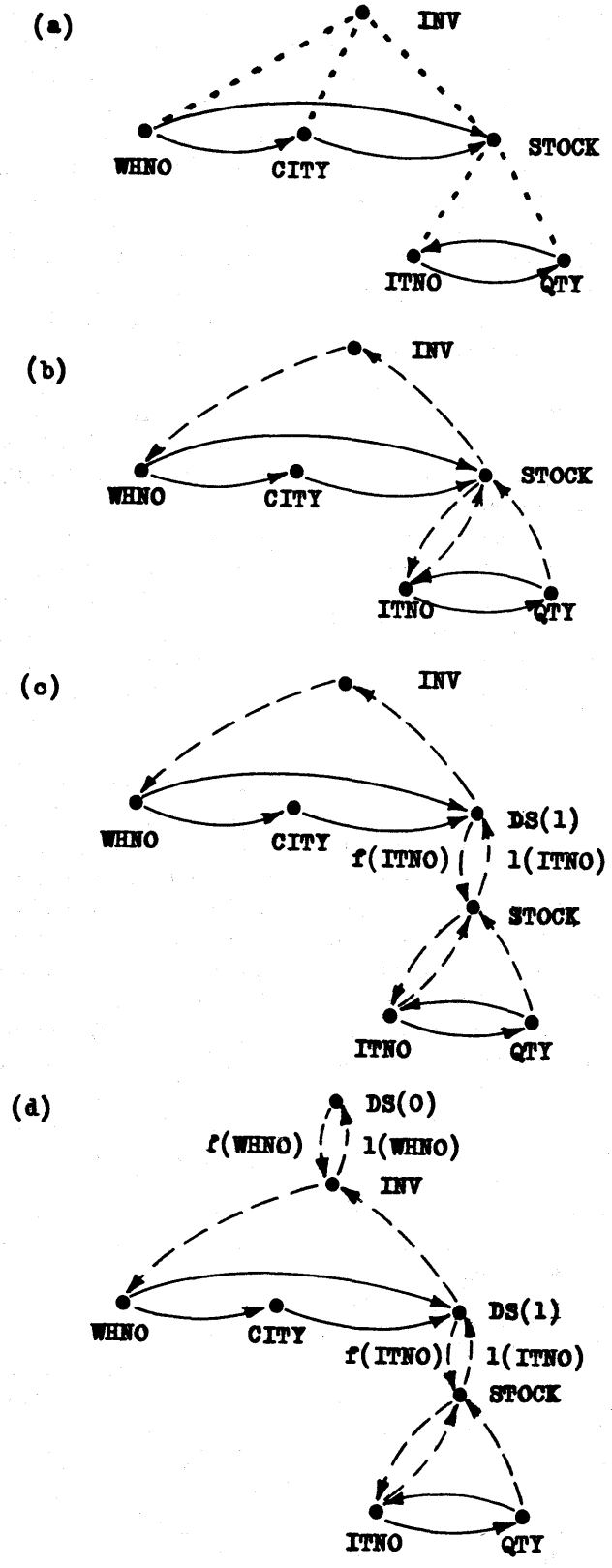


Figure 5—Development of a data structure for INV

STOCK, then through STOCK to QTY, the questions can be raised: At what point or points, ITNO and/or QTY, can the path enter the data structure STOCK and at what point or points can the path exit STOCK? What is the precise path from WHNO to QTY and out to another component?

It is important that this ambiguity be resolved. When the data structure is represented in storage, the programs which represent the logical access processes will operate on the storage representations of the logical access paths in order to access a given component representation. The ambiguity in the path from WHNO to QTY must be resolved if the program representing the logical access process is to reference the representation of QTY from the representation of WHNO.

The ambiguity is resolved here by designating one or more of the data structure components as entry components and one or more components as exit components of the given structure. A data structure component may be an entry component, an exit component, both, or neither. The set of entry and exit components will be called the *boundary* of the data structure. Since a data item may be considered an elementary data structure, the boundary of the data item is the data item itself. A data item is its own entry and exit component.

Thus, the connection to a data structure means a logical reference to its boundary; that is, to each of its entry components. A connection from a data structure means a logical reference from each of its exit components. This interpretation of connections makes no assumptions about the storage representation of the connection or of the boundary. When the boundary consists of multiple entry and exit components the logical access process must contain the logic for deciding through which entry and exit component the logical access path should go.

In the graph depiction of a data structure the boundary may be represented by broken arcs from the vertex representing the data structure to the vertices representing entry components; and by broken arcs from the vertices representing exit components to the vertex representing the data structure. The graph representation of the data structure then has a vertex denoted by the name of the data structure and a sub-graph representing the logical relationships among the data structure components. The arcs representing the boundary specify which subgraph is associated with which data structure vertex.

In the data structure INV, Figure 5b, WHNO has been designated as the entry component of the data structure INV and STOCK as the exit component. ITNO has been designated as both an entry and an exit component of STOCK. QTY occurs also as an exit component of STOCK. The boundary of INV is the component set consisting of WHNO and STOCK.

One piece is still missing from the picture of a data structure. An instance of a data set may contain multiple instances of its components. For example, for each WHNO there may be one CITY but many STOCKs. In the data set INV the same WHNO instance and CITY instance, for example ⟨WHNO,3⟩ and ⟨CITY,A⟩ in Figure 4, were associated redundantly with each of three different STOCK instances. The logical design of the data may specify that for each STOCK instance the corresponding WHNO and CITY instances will be repeated and maintained in the logical structure. In other words full redundancy of data will be maintained. If this design is implemented in storage, the same values of WHNO and CITY will be stored for each related instance of STOCK. On the other hand the logical design may specify that only one occurrence of the redundant WHNO and CITY will be maintained and with that occurrence of WHNO and CITY will be associated a new data structure each of whose components is one of the related instances of the data structure STOCK. The redundancy of WHNO and CITY has been reduced. This structure is depicted in Figure 5c. A structure of multiple instances of the same data structure is sometimes called a repeating group.

Within this newly created structure the instances of STOCK for a given WHNO/CITY combination are given some ordering, e.g., ascending numerical order by ITNO value. In addition, a boundary is specified for this new data structure; for instance, the entry component is the numerically first STOCK (f(ITNO)) and the exit component is the numerically last STOCK (l(ITNO)). In the graph these ordering and boundary specifications can be attached to the arcs to and from the STOCK vertex. The system designer may give a name to this new structure, as DS(1) in Figure 5c.

Assuming the given redundancy reduction, one can apply similar reasoning at the level of INV. According to the cardinality relationships given earlier, several instances of the data structure for INV will occur in an instance of the data structure, one for each instance of WHNO. Each instance of INV will have the logical structure described in Figure 5c. This new structure has three components: WHNO, CITY, and DS(1). In each instance of INV, WHNO and CITY appear once and are connected to a data structure whose components are instances of STOCK. The data structure, DS(0), combining all instances of INV structure will be an ordering of instances and will have a specified boundary.

The complete specification of the data structure is given in Figure 5d. The design gives a specification of both ordering and redundancy and establishes the

network by which a data structure component may logically be accessed. Note that the membership relationships given by the problem definer have been maintained.

Associated with a data structure is one or more logical access processes which function to find or access the relative logical position of a component instance in an instance of the data structure. A logical access process uses contiguous connection instances to construct a path to the relative position of the desired component instance. For example, to find the data value of CITY for a given data value of WHNO, an access process for the above structure might create a path which starts at the entry component instance of the data structure DS(0) and courses through each instance of INV until it finds the given WHNO value which connects to the required CITY value. In each instance of INV the path leads from WHNO to DS(1) and exits via the DS(1) vertex. The access path does not course logically through the instances of STOCK.

From the point of view of the system designer a logical access process is like a problem-defined process. The system design process must incorporate it with the problem-defined processes into the program structure. Any data which the logical access processes need in order to function properly are designer-defined data sets which themselves must be organized logically and physically. At this point the system designer becomes a problem definer.

Physical organization

Physical organization of data means here the IPS storage representation of the given data structure. Two degrees of physical organization should be recognized: *relative* organization and *absolute* organization. Relative organization involves the *manner* in which the data structure components, connections, and boundary of a data structure will be represented in IPS storage. Such a specification involves questions of numbers of cells of memory required, sequential or non-sequential storage of components, header cells, etc., but not the actual locations of the data in storage. Absolute organization involves the *actual locations* of the components, connections, and boundary representations in storage. Absolute organization is specified by a set of storage assignment processes and must maintain the specified relative storage organization. In the following discussion major consideration is given only to the relative organization.

For the design of relative physical organization a relative storage area is defined here. This conceptual storage area is an ordered set of cells. Each *cell* is

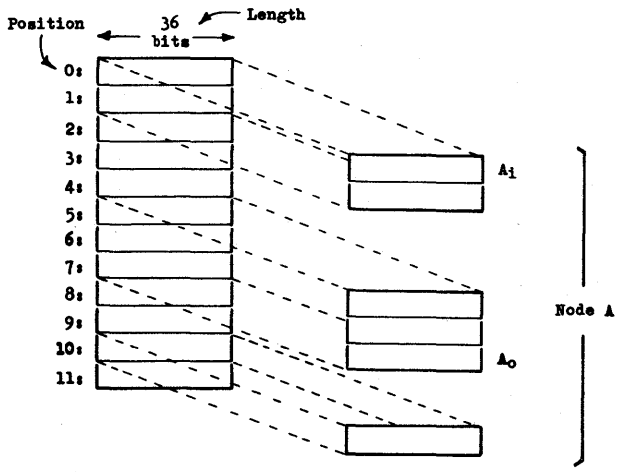


Figure 6—Storage node A in relative storage

uniquely identified by an order-relative position and a length. Cells are non-overlapping. The length of a cell is measured in common units, such as bits.

Looked upon as elements of a set, and regardless of what they will represent, the cells in relative storage may be grouped together conceptually in many different ways. A *storage node*, or simply *node*, is defined to be a collection of elements each of which is a cell or a storage node. The cells in a storage node need not be contiguous relative to the ordering. In Figure 6 node A consists of three elements: two nodes and a cell. A single cell may be regarded as an elementary node.

For convenience in referencing a node relative to another node a measure of separation between the two nodes is defined. A *separation* between two nodes will be defined as the number of units of length, bits or words for instance, which, according to the given order-relative positions of the cells in relative storage, separates a reference cell in one node from a reference cell in the other node. The cell from which reference is made in the given node to another node will be called an *exit cell* of the given node. The cell to which reference is made in the given node from another node will be called an *entry cell*. The reference cells of a node will be called the *node boundary*. An elementary node or single cell is its own entry and exit cell.

Specification of entry and exit cells for a node is required for much the same reason that entry and exit components are specified for a data structure. If particular boundary cells were not specified, then reference to or from a multi-cell node would be ambiguous. In Figure 6 cell 0 has been designated the entry cell of node A (denoted by A_i). Cell 7 has been designated the exit cell (denoted by A_o).

It should be noted that the choice of the boundary

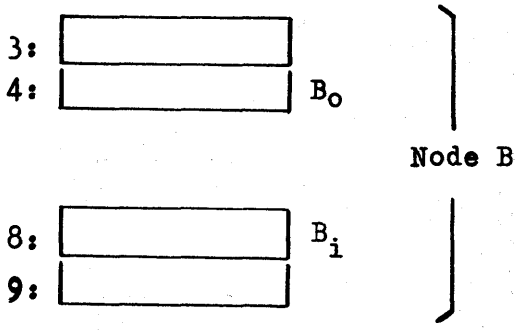


Figure 7—Storage node B

of a node in the conceptual relative storage is arbitrary. Multiple entry and exit cells may be designated in a node. Several different separations can occur between two nodes if one or both have multiple entry and exit cells. In Figures 6 and 7 only a single separation has been defined between nodes A and B. This separation is one cell—A_o to B_i. Figure 6 and following assume that all cells have the same length and that separation is measured in number of cells.

The system designer must specify first how to represent a data structure by a node in the conceptual relative storage and then specify a storage assignment process for mapping the node into absolute storage. The relative storage representation of the components, connections, and boundary of a data structure will be called here a *storage structure.*

A data structure component is represented in the relative storage by a node. If the data structure component is a data item, this node may be a set of cells which are contiguous relative to the ordering of relative storage. In Figure 8a the designer has decided to represent the data item WHNO by a two-cell node with the first cell being both the entry cell, $WHNO_i$, and the exit cell, $WHNO_o$. The system designer has decided that the first cell of the node will be the reference cell in any future separation specifications. The specific order-relative position of this node in relative storage is unimportant. Only the separation between it and other element nodes of the given storage structure is important. Figure 8a also represents data items CITY, ITNO, and QTY. The number of cells required to represent the data item is determined from the problem-defined size of the data value. This representation assumes that only the data value is to be represented in the node.

If the data structure component is a data structure itself then the storage structure may be defined recursively by representing the components, connections, and boundary of this component.

A connection in a data structure may be represented in one of two ways:

1. by a fixed separation between an exit cell of the node representing the first component and an entry cell of the node representing the second component;
2. by a variable separation which will not be given actual value until storage assignment takes place.

In either case the IPS will maintain a record of the separation. In common practice the fixed separation is recorded as a fixed address offset in program instructions. To handle variable separation the system designer may define another data item, a pointer, to be associated with the data structure component from which the connection is defined. The system designer also defines maintenance processes to update the pointer and perhaps other associated data sets, such as headers and trailers, to aid in maintenance. In Figure 8b the connection ⟨WHNO,CITY⟩ has been represented by a variable separation in the form of a pointer. A fixed

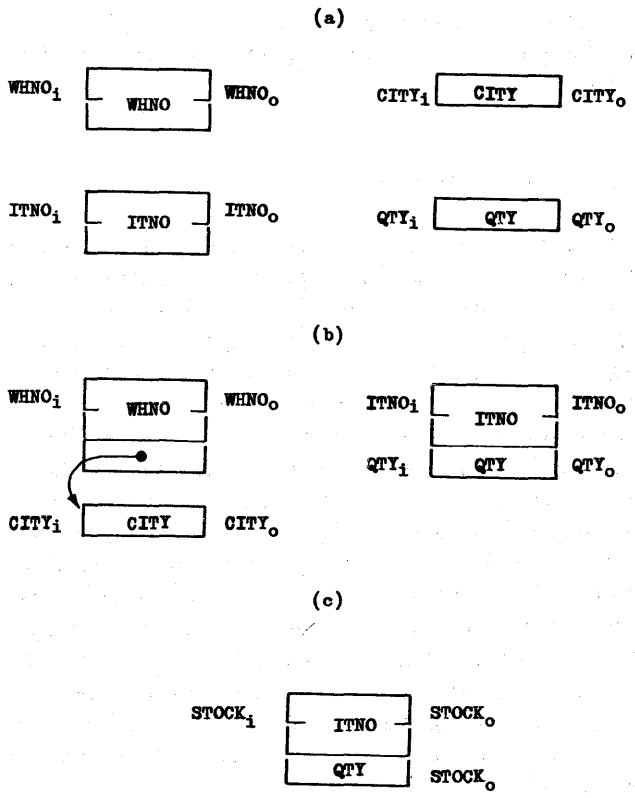


Figure 8—Development of storage structure

separation of two cells has been specified to represent the connections ⟨ITNO,QTY⟩ and ⟨QTY,ITNO⟩.

The data structure boundary is represented by a node boundary developed in the following way. The designer may specify that the boundary of the node representing the whole data structure consists of the entry cells of nodes representing the data structure entry components and the exit cells of nodes representing the data structure exit components. Alternatively, the designer may incorporate additional cells in the node and define them to be the entry and exit cells of the node. He then defines a fixed or variable separation between these cells and the respective boundary cells of nodes representing the data structure entry and exit components. The additional cells and the boundary cells of nodes representing the data structure entry and exit components together represent the data structure boundary.

In terms of the graph representation of a data structure, for instance Figure 5d, the use of additional cells corresponds to treatment of the broken arc, say from DS(1) to STOCK, as a true connection; DS(1) is represented by the additional cells and the connection is represented by fixed or variable separations between these additional cells and the ENTRY and exit cells for the first and last instances of STOCK, respectively. If no additional cells are used, the broken arc is not viewed as a true connection and is therefore not represented explicitly in relative storage.

In Figure 8c the data structure boundary of STOCK has been represented by the entry cell of the entry component ITNO and the exit cells of the exit components ITNO and QTY.

Associated with a storage structure is one or more storage assignment processes. A storage assignment process will apply the relative storage structure design to an instance of the data structure and assign actual physical hardware memory locations. The storage assignment process is responsible for maintaining all "data about data" which is necessary for assignment of positions and all positional information which is necessary for use by the implemented logical access processes. The anticipated number of length units, e.g., cells, required by a node to represent a data structure instance may be developed from the size of data item nodes, the cardinality relationships given by the problem definer, the amount of redundancy defined by the system designer, and the requirements of pointers and related additions. See McCuskey[3] for details.

A storage assignment process, like logical access processes, must be incorporated with the problem-defined processes to create program structure, whether at the operating system level or elsewhere. Any "data about stored data" which the storage assignment process requires is, from the point of view of the system designer, just like problem-defined data—data sets which must be given logical and physical structure.

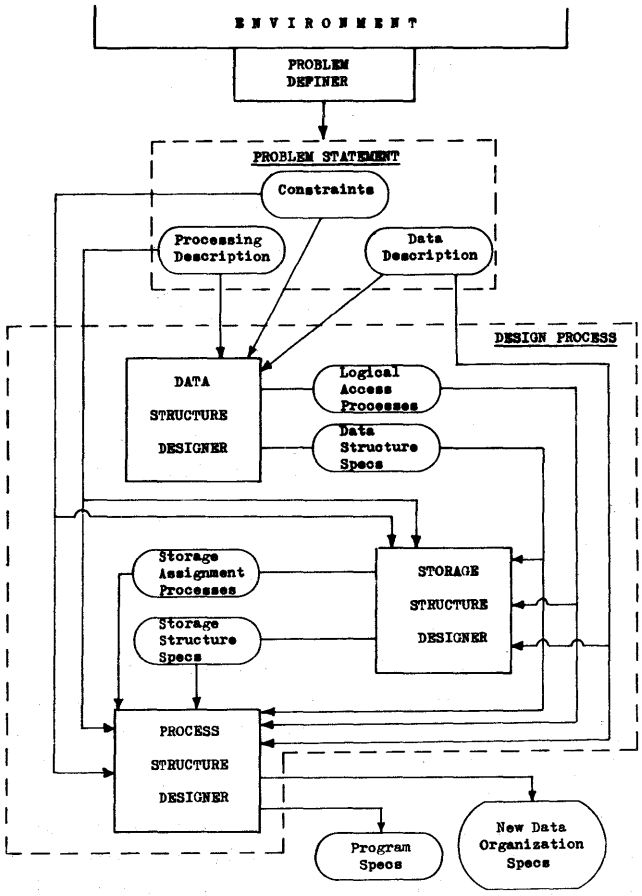


Figure 9—Data organization design process

DESIGN PROCESS

The goal of the concept descriptions above is to provide a framework within which to formulate an algorithm which, given a specification of problem-defined data, would specify how the actual data will be stored and accessed in the IPS. Figure 9 gives a very general flow chart of a design process for data organization.

In the design process the data structure designer accepts a specification of data sets and generates a specification of data structure (components, connections, and boundaries) and of logical access processes. While generating a specification of data structure, the designer acts itself as a problem definer; the problem is logical access to components of the data structure. The

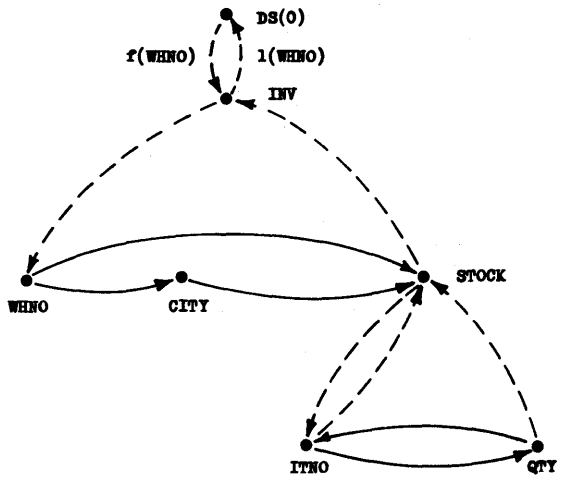


Figure 10—Result of process D

definition of logical access processes must be input to the process structure design in order to be incorporated in the program specification. The structural information must be specified in terms of data sets and then input to the design algorithm.

The storage structure designer accepts the specifications generated by the data structure designer and produces a specification of storage structure (relative storage representation of data structure components, connections, and boundaries) as well as the storage assignment processes which will map the storage structure into absolute storage. Like the data structure designer, the storage structure designer is a problem definer; the problem is storage assignment. The storage assignment processes and information required by those processes must be defined and run through the design algorithm.

The process structure designer organizes the original problem-defined processes, the logical access processes, and the storage assignment processes and generates program specifications. How the logical access processes are represented in programs depends on how the storage structure and storage assignment processes have been designed. How the storage assignment processes are represented in programs depends on the characteristics of the absolute storage of the IPS.

In the context of this general picture of the design process only the specification of data structure and storage structure is considered below. An initial attempt at a method of generating alternative designs is described. The purpose of this attempt was to gain an understanding of what decision points are involved. No decision-maker has been developed. How a decision should be made at each point depends on the relation between the designed structure, the processes operating on it, and the performance criterion. As yet this relation is not understood.

Consider the specification of data structure for set INV (Figure 3). Suppose first that the given redundancy is to be retained. Then a general, recursive data structure design procedure might be:

Process D

1. For each component of the given set, if the component is not a data item then apply Process D recursively.
2. Define connections among the components of the given set.
3. Define a boundary from among the given components.

The process assumes all instances of a component are structured alike. A component may be a data set component or, in a repeating group, a data structure instance. The result of an application of Process D to INV, yielding a structure similar to that in Figure 5d, is given in Figure 10. Note that redundancy of WHNO and CITY will be maintained here while in Figure 5d it is not retained.

Suppose now that Process D has not been applied to INV. Suppose one wishes only to reduce redundancy. Reduction of redundancy may be accomplished in the following way:

Process R

1. Partition the original set according to the data values of instances of one or more components. A partition element is a subset of the original set. In a partition element each criterion component maintains multiple instances of the same data value.
2. Replace the partition element by a new element in the following way:
   a. one instance of each criterion component replaces the multiple instances;
   b. the remainder of the original subset is grouped by itself as a separate element.

The replacement operation will be called here *truncation*. The remainder will be called the truncated set. Figure 11 develops from Figure 4 a partition and truncation of INV according to the values of WHNO. The deleted component instances are blacked out. As in Figure 4 rows represent (unordered) data set elements and columns represent (unordered) data set components.

In Process R step one establishes which redundancies may be reduced in step two. The partition in Figure 10 establishes the redundancy of WHNO by definition; redundancy of CITY is established because the problem definer specified only one CITY instance per WHNO instance. The truncation operation performs the actual redundancy reduction. Neither, one, or both of WHNO and CITY may have redundancy reduced. In Figure 11 both were chosen.

These operations may be extended. A sequence of partitions leads to several levels of redundancy reduction. The sequence of partitions establishes a set of candidates for redundancy reduction at each level. The candidates are the criterion components established at that level or above and other components which are in one-to-one correspondence to criterion components. Starting at the deepest level and working upward, the design process can decide which candidates to choose for redundancy reduction. For a given candidate to have its redundancy reduced to a minimum its redundancy must be reduced at each level from the deepest up to the level at which it originally entered the set of candidates. If its redundancy is not reduced at the deepest level then its full redundancy is maintained. Partial redundancy for a component is established by not selecting the component for reduction at some level in between. Once the component fails to be chosen it drops out of the set of candidates; its redundancy is fixed.

This expanded redundancy reduction scheme at each level factors out selected components to the next higher level and leads to a hierarchical form of organization. The scheme may be combined with Process D above to form Process DS:

Process DS

1. Define n-level partition.
2. For level n, $n-1, \ldots, 0$.
   a. Define a truncation at this level.
   b. In the truncated set.
      i. apply Process D with data set components and, possibly, truncated sets as components.
      ii. apply Process D with truncated set elements as components.

Operation 2.b.i specifies the structure of an element of a repeating group or data set. Operation 2.b.ii amounts to specifying the structure of that repeating group. Once a component or truncated set has been structured it is a data structure component.

Figure 5d shows the pattern resulting from one application of Process DS to INV. Figure 12 shows the results of applying this pattern to the instance of INV given in Figure 4.

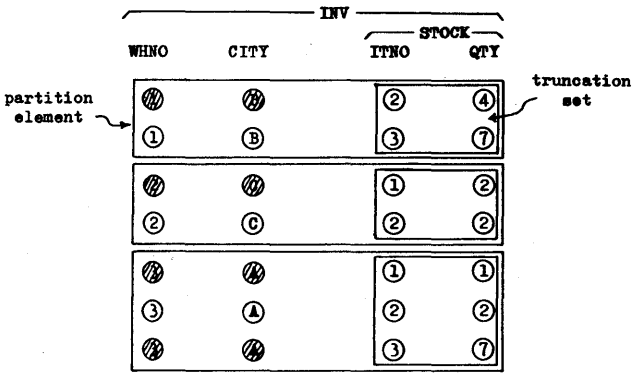


Figure 11—Partition, truncation, and redundancy reduction

Consider next the specification of storage structure. A data structure INV has been specified in Figure 5d. The cardinality relationships among WHNO and STOCK are known. Following the definitions given earlier, one can specify a storage structure design process as follows:

Process SS

1. Represent each data structure component.
   a. If the component is a data item then specify the storage representation of its value and represent the data item by a set of contiguous cells large enough to contain the storage representation of the value.
   b. If the component is a data structure, apply Process SS recursively.
2. Represent each connection by either a fixed separation or variable separation.
3. Represent data structure boundary by a node boundary.

Figure 13 shows the result of an application of this process to the instance of the data structure given in Figure 12. The decisions shown in Figure 8 have been repeated here. Note that in the storage representation of data structure DS(1) the entry and exit cells of the node are additional cells; the broken arcs between DS(1) to STOCK in Figure 5d have been treated as true connections. These additional cells may be viewed as a header and a trailer for the data structure DS(1). Variable separation represents connections among instances of STOCK in DS(1) and among instances of INV in DS(0). One could question the value of this

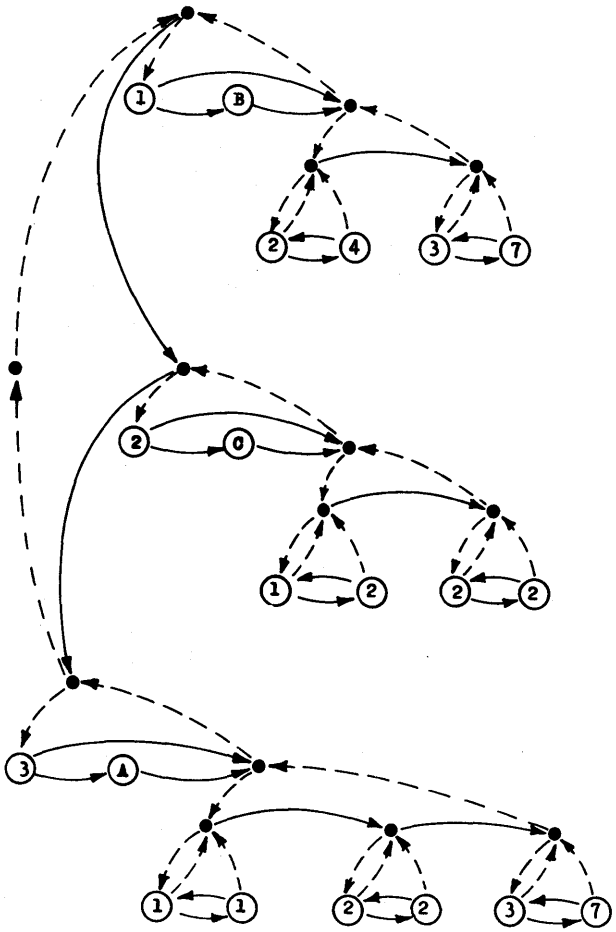


Figure 12—Instance of data structure

design if one knew what the data usage was to be. However, the important point is that the storage structure process has represented all data structure components, connections, and boundaries as given in the data structure specification.

Processes DS and SS have been implemented in ALGOL on the B5500.

CONCLUSION

Comments and implications

The intent of the research on data organization described above was to build a model of the data organization process and to develop a method for generating specifications of alternative data organizations. The set-theoretic approach was helpful in abstracting and precisely defining the component pieces of the data design problem. The model of data organization and the initial design procedures, as described above, are based on an ideal. The ideal is that a problem definer can provide a complete, concise and consistent problem statement which is completely sufficient for a system design process to generate specifications of an IPS to solve the information processing problem defined in the problem statement. One goal of Project ISDOS is to develop a problem statement language to facilitate such problem definition. Another goal is to define a design procedure to generate specifications. The following comments and implications must be viewed relative to this ideal.

First consider data sets. The set-theoretic approach was stimulated by Information Algebra described in CODASYL.[4] Many of the ideas developed there find similar concepts in the present formulation. For instance, property and value correspond to data name and data value. Property space corresponds somewhat

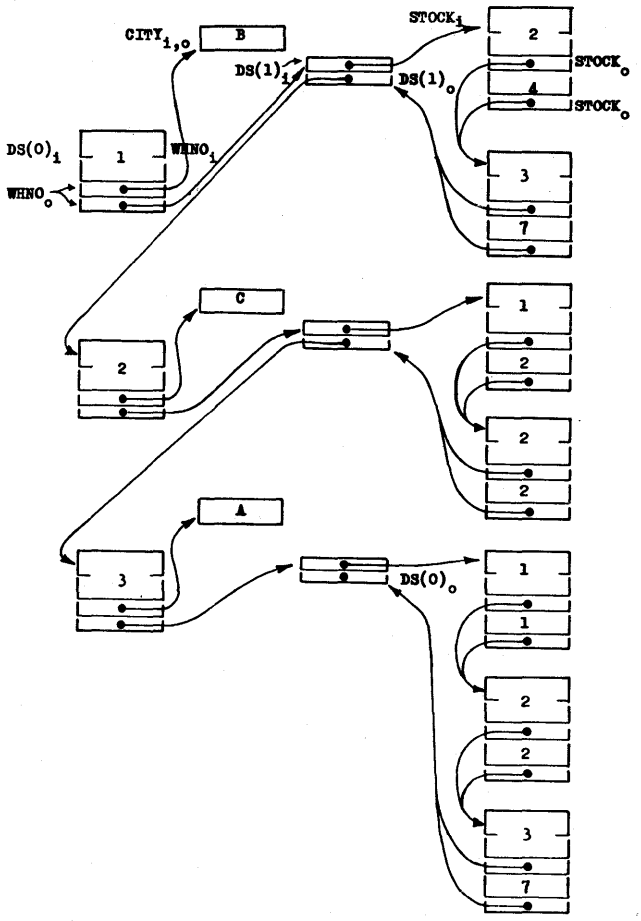


Figure 13—Instance of storage structure

to data set, except that property space is a set of *ordered* n-tuples. However, the goal of Information Algebra was a non-procedural language. The concepts of data in themselves do not lead to a solution of the problem of automatic design. The abstract formulation was also influenced by Young and Kent[5] who specify information sets and relationships which are the counterparts of data sets and cardinality relationships.

Throughout the current paper, data sets were assumed to be problem-defined, or in special cases system-defined. In order to obtain a good system design, the designer may wish to redefine and coalesce some problem-defined data sets before proceeding to logical and physical organization. Such redefinition does not alter the logical and physical structure design processes just described. A pre-processor—data set redesigner—is introduced between the problem statement and the data structure designer.

Next consider data structures. Several levels of meaning are commonly attached to the term "data structure". At the source language level, e.g., in ALGOL, the programmer considers an ARRAY to be a data structure and thinks in terms of, say, dimensions. However, the structure represented in storage may have array elements in row major order. The programmer may never see evidence of this actual logical structure (row major order) but the IPS operates on the directly represented structure and must be designed to handle it. The term data structure, used in this paper, refers to the latter structure, the logical structure which is actually and directly represented in the IPS storage. In order to consider all possible alternatives, including structures which perhaps have no common names, the design process must operate in terms of the structure actually represented in storage.

The design procedure for data structures involved partitioning, truncation and ordering. The given procedure developed a generally hierarchical logical organization. The implications of applying multiple partition sequences and truncation to the same data set must be studied. Multiple partitioning would be required to develop more complicated, IDS-type structures involving multiple paths through the same data structure components.

The idea of generating logical access processes for a given data structure has interesting implications. In current practice data sets are structured to take advantage of the known, good access processes. For instance, data is often organized to take advantage of binary search or indexed sequential access. Often the data and its usage may not be suited to the known access processes. Perhaps what is needed is a procedure to generate logical access processes to fit the given data structure. The appropriate access process would be chosen dynamically at run-time according to the current status of the data.

Consider next storage structures. The distinction between storage structure and "data structure" has been made before by others. See for instance D'Imperio[6] and Knuth.[7] However, their distinctions are apparently not specific enough to suggest the automatic design generator desired here.

The idea of generating storage assignment processes to fit a given storage structure has implications similar to those of logical access process generation. Instead of designing a storage structure to fit the results of a given storage assignment process, perhaps we should have a generator to provide assignment processes tailored to the given storage structure. Within the context of automatic design and implementation of IPS these alternatives are worth considering.

Finally, the utility of design procedures which generate alternatives is minimal until a decision-making process is developed to select among the alternative designs. The decision-making process relates the data organization to data usage, i.e., to the process and program structure design, and will require much more research.

REFERENCES

1 J F NUNAMAKER JR
On the design and optimization of information processing systems
PhD Dissertation Case Western Reserve University 1969

2 W A McLUSKEY
Toward the automatic design of data organization for large-scale information processing systems
PhD Dissertation Case Western Reserve University 1969

3 Ibid

4 CODASYL DEVELOPMENT COMMITTEE (LANGUAGE STRUCTURE GROUP)
An information algebra
Comm ACM Vol 5 No 4 pp 190-204 April 1962

5 J W YOUNG JR H K KENT
Abstract formulation of data processing problems
J Ind Engr Vol 9 No 6 pp 471-479 November-December 1958

6 M D'IMPERIO
Data structures and their representation in storage
Annual Review of Automatic Programming Vol 5 pp 1-75 Pergamon Press Oxford 1969

7 D E KNUTH
The art of computer programming
Vol I Addison-Wesley Reading Mass 1968

Analysis of retrieval performance for selected file organization techniques

by A. J. COLLMEYER and J. E. SHEMER

Xerox Data Systems
El Segundo, California

INTRODUCTION

With the rapid development of the Business Data Processing technology, numerous Data Management Systems have evolved. Many have been written for use on a variety of machines, in a variety of applications. In each case, the basic limitations on performance are a function of the file organization technique(s) supported within the system. Apart from such mundane considerations as cost, compatibility, etc., the selection of the "best" system is frequently reduced to the selection of the file organization technique best suited to the intended application. Just as a variety of applications exist, so a variety of file organization techniques are available. File organization techniques are frequently classified in terms of their operational characteristics. More basic, however, is a classification based on functional objectives. Collmeyer[1] has described four basic types of file organization techniques. The simplest, Type O, is exemplified by the "sequential" organization wherein records are filed in the order of arrival* and retrieval is accomplished via a sequential (top-to-bottom) search of the file.

On the other hand, a Type 1 technique embodies a store-retrieve philosophy which provides for the expeditious retrieval of a record given a key which uniquely identifies said record. Techniques of this type differ basically in their approach to indexing. By index, we mean that instrument which facilitates the location of a record given an identifier. There are three basic types of indices: Spatial, Tabular, and Calculated. With a Spatial Index, records in a file are stored in (physical) order according to the values of their respective identifiers. Given an appropriate identifier, retrieval is accomplished by an intelligent search (e.g., binary search) of the file itself. In contrast, the Calculated Index takes the form of an algebraic transformation, τ, which converts an identifier into a subfile address.** Here a subfile is defined as a set consisting of all records which, under the transformation τ, yield the same "address." To retrieve a record, its identifier is transformed; an address is thus obtained. The records stored at this address are examined (by identifier) in an effort to locate the record in question.

The third type of index is the Tabular Index. The tabular index approach involves the maintenance of a list of identifiers together with their respective record addresses. Retrieval is accomplished by searching the list for the given identifier, obtaining in the end the address of the desired record. The list itself may be viewed as a file, wherein each record consists of an identifier together with the address of the associated record. As a file, this list may be organized in accordance with any of the Type 1 techniques here described. That is, the index (list) may itself be indexed in one of the three methods (Spatial, Tabular, or Calculated). To be sure, the most common method is the spatial indexing method. However, the index could just as well be organized as a random file† with a Calculated Index; in which case one transforms an identifier to obtain the address of the appropriate index subfile. While this organization tends to minimize the time spent searching the index, it makes sequential processing very awkward. A third approach, which affords rapid retrieval without degrading sequential processing efficiency, involves the use of a (second) Tabular Index. This implementation is known as *multi-level indexing* and will be described fully in the next section.

In the paragraphs which follow, models of the foregoing techniques are developed. These models are

* It is noted that file organization techniques of this type are frequently augmented with a pre-store sort.

** File organization techniques of the Calculated Index variety are known as scatter storage techniques or "Hashing" techniques.

† A *random file* is one in which the physical sequence of records has no logical meaning.

	INPUT	OPERATION	OUTPUT
STEP 1	KEY (IDENTIFIER)	SEARCH INDEX FILE	ADDRESS
STEP 2	ADDRESS	ACCESS MAIN FILE	DESIRED RECORD

Figure 1—Retrieval process—Case I

analyzed to quantitatively define the retrieval process. While this analysis could be extended to define additional performance characteristics (e.g., sequential processing efficiency, insert/delete time), such is outside the scope of this paper.

STATEMENT OF THE PROBLEM

The techniques chosen for this investigation are members of a particular class of file organization techniques. They are, in the terminology of the preceding section, Type 1 techniques of the Tabular Index variety. That is to say, each of the techniques makes use of a (separate) list of identifier-address pairs to expedite the retrieval of a record given its identifier. Each technique thus involves an *index file* in addition to the *main file*. For the purpose of this study, it is assumed that the main file is organized as a random file.* The discussion focuses on three different approaches to the organization of the index file; that is, three different methods for indexing the index file.

Case I: *Spatial index*

The first technique features a Spatial Index. Records** within the index file are stored in (physical) order according to the values of their respective identifiers. Retrieval (of an index record) is accomplished by a search of the file. On finding the appropriate index record, the address of the desired data record is obtained. The retrieval process is a two-step process (Figure 1). The first step is a search of the index file for the address of the desired data record. With this address, the main file is accessed and the record retrieved. The time required to retrieve a record thus depends on file parameters, the characteristics of the storage media, and the search strategy applied.

* This is reasonable in view of the fact that we are concerned solely with retrieval. Other performance characteristics such as sequential processing efficiency are, of course, dependent on the organization of the main file.

** Here a record consists of an identifier-address pair.

Case II: *Calculated index*

The second technique employs a calculated index. The key (identifier) is transformed to obtain the appropriate subfile address. An index subfile is then accessed in an effort to locate the desired index record. In the event the record is not found in its "home" location, one of several overflow strategies must be invoked. Once located, the index record provides the address of the desired data record. Hence the retrieval process is a three-step process (Figure 2). As before, the retrieval time is a function of the file parameters, the characteristics of the storage media, and the overflow strategy employed.

Case III: *Tabular indices*

The third technique is known as multi-level indexing. By this technique K index files are maintained, each of which is constructed as a (logically) ordered set of sequential subfiles, where a subfile is defined as a chain of blocks (frequently a single block).

Definition: A *block* is the amount of data physically transferred between secondary storage and main memory as a result of a single I/O operation. The block size is usually given in bytes or characters.

For the purpose of this study, subfiles are fixed at *one block*. The lowest level index file (the Kth level) is therefore a logically connected (linked) set of blocks each of which contains up to b_K index records. In the aggregate, this file contains as many records as the main file. The next level (index) file is likewise a logically connected set of blocks each of which contains up to b_{K-1} index records. However, this file serves as an index to the Kth index file. Accordingly, it contains a record for each subfile (block) of the Kth index file. Similarly, the kth level index file (a logically connected set of blocks each of which contains up to b_k index

	INPUT	OPERATION	OUTPUT
STEP 1	KEY (IDENTIFIER)	TRANSFORM KEY	"HOME" ADDRESS
STEP 2	"HOME" ADDRESS	SEARCH "HOME" LOCATION AND OVERFLOW IF NECESSARY	ADDRESS
STEP 3	ADDRESS	ACCESS MAIN FILE	DESIRED RECORD

Figure 2—Retrieval Process—Case II

records) serves as an index to the $k+1$ index file. The retrieval process thus involves $K+1$ steps (Figure 3).

It is the purpose of this paper to quantitatively define the retrieval process for each of the techniques described above. Of particular interest is the mean time to retrieve a record, $\bar{T}_r$. In the derivation of $\bar{T}_r$ a uniprogramming environment is assumed; i.e., no delays are incurred as a consequence of shared equipments. In each case, $\bar{T}_r$ is expressed as a function of the file parameters, the characteristics of the storage device and the search or overflow strategy employed.

PROBABILITY MODELS

To quantitatively define the retrieval process in each of the cases described above, it is necessary to model the search strategies as well as the secondary storage devices. In modelling the search strategies, we are particularly interested in the number of blocks transported from secondary storage to main memory in the course of an index search. This number, represented by j, is a random variable; the distribution of j is of course a function of the file parameters as well as the search strategy. Once j has been defined, it remains only to model the secondary storage devices. For the purposes of this study, we have limited our consideration to two types of secondary storage. The first is exemplified by the drum, a fixed head device. The second is a moving arm device such as the disk. The time r required to transport (locate and transfer) a block from the media to main memory has been modelled for random addressing as well as sequential addressing (see Appendix C).

In modelling the search strategies, we assume that each of the m records has the same probability of being referenced. With this assumption, we proceed to derive the mean retrieval time for each of the several cases.

	INPUT	OPERATION	OUTPUT
STEP 1	KEY (IDENTIFIER)	SEARCH INDEX FILE (TOP LEVEL)	ADDRESS
STEP 2	ADDRESS	SEARCH DESIGNATED SUBFILE (BLOCK)	ADDRESS
⋮	⋮	⋮	⋮
STEP K	ADDRESS	SEARCH DESIGNATED SUBFILE (BLOCK)	ADDRESS
STEP K+1	ADDRESS	ACCESS MAIN FILE	DESIRED RECORD

Figure 3—Retrieval process—Case III

Case I: Spatial index

In this case, records in the index file are stored in (physical) order according to the values of their respective identifiers. We shall assume that these records are filed in contiguous blocks, where the mean number of records per block is denoted by $\bar{\imath}$. If the index file is searched sequentially from top to bottom, the mean number of blocks transported from the media to main memory is (from Appendix A)

$$E[j]=(n+1)/2 \tag{1}$$

where

$$n \triangleq m/\bar{\imath} \tag{2}$$

The mean time to retrieve a record is therefore*

$$\bar{T}_r=\underbrace{\bar{r}_{\text{rand}}+\{E[j]-1\}\bar{r}_{\text{seq}}}_{\text{Retrieval of index record}}+\underbrace{\bar{r}_{\text{rand}}}_{\text{Retrieval of data record}} \tag{3}$$

where

$\bar{r}_{\text{rand}} \triangleq$ Mean time to transport a block from the media to main memory assuming random addressing.

$\bar{r}_{\text{seq}} \triangleq$ Mean time to transport a block from the media to main memory assuming sequential addressing.

If a binary search of the index is performed, the mean number of blocks transported in the course of the search is (from Appendix A).

$$E[j]=q[(n+1)/n]-[(2^q-1)/n] \tag{4}$$

where q is obtained as the solution to

$$2^{q-1}\leq n\leq 2^q-1 \tag{5}$$

and

$$n \triangleq m/\bar{\imath} \tag{6}$$

Figure 4 describes the variation in $E[j]$ as a function of m for several values of $\bar{\imath}$. While the binary search involves fewer blocks than the sequential search, it does not follow that the binary search is faster. In this case the mean time to retrieve a record is given by

$$\bar{T}_r=\underbrace{E[j]\bar{r}_{\text{rand}}}_{\text{Retrieval of index record}}+\underbrace{\bar{r}_{\text{rand}}}_{\text{Retrieval of data record}} \tag{7}$$

* The expression above assumes that the processing time (CPU time) is negligible in comparison to the transport time. This assumption is quite reasonable; with present systems, a list of 100 keys can be searched in core in less than .5 msec, whereas a disk access may require 50 msec or more.

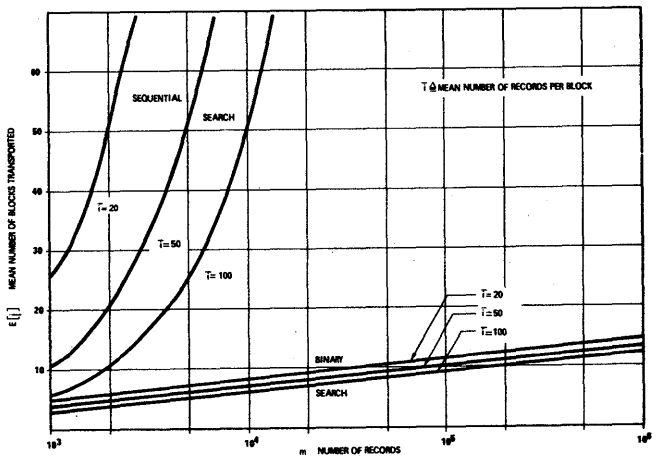


Figure 4—Mean number of blocks transported (Case I)

An actual comparison of the binary search and sequential search on the basis of retrieval time is provided in the next section.

Case II: Calculated index

With a calculated index, the key is transformed to obtain the appropriate (index) subfile address. The address obtained via the transformation is the "home" location of the desired index record; that is, the address of the first of a set of blocks containing the subfile. If the record is not found in its "home" location, the subfile must be transported a block at a time to main memory until the record is found or the subfile exhausted. Two schemes for mapping subfiles onto the media are described in Appendix B. The first is known as Linear Probing. With Linear Probing, the media is partitioned into n blocks each with a capacity of b records. A one-to-one correspondence between blocks and subfiles is established.* When the kth block is filled, a record belonging to the kth subfile is placed in the $k+1$ block, provided there is room. If the $k+1$ block is filled, the $k+2$ block is probed, and so it goes until a partially filled (or empty) block is found. To retrieve a record, the search begins with the "home" location. If the record is not found, the next block** is probed, and so on.

In an early paper on Linear Probing, Peterson[6] described the variation in the mean number of blocks probed to retrieve a record. His results, obtained after a large number of insertions and deletions, are recorded in Figure 5. With Linear Probing the mean retrieval time is given by Equation 3 above.

A second scheme, identified as Block Chaining, is defined in Appendix B. As with Linear Probing, n blocks each with a capacity of b records are allocated *a priori*, one for each subfile. However when the kth block is filled, records belonging to the kth subfile are placed in a separate overflow block chained to the (filled) kth block. Additional overflow blocks are added to the chain as needed. With Block Chaining, all the records belonging to a given subfile are contained in a chain of blocks, separated from those of another subfile. The obvious advantage of the Block Chaining approach is that a file may be expanded (beyond the original m records) dynamically, without restructuring. The mean number of blocks accessed to retrieve a record is derived in Appendix B and is plotted in Figure 5. The mean retrieval time is given by Equation 7.

Case III: Tabular indices

With multi-level indexing, K index files are maintained, each of which is constructed as a sequential file. In general the number of levels K is selected so that the top level is contained entirely within a single block. Under the assumption that the mean occupancy of a block is the same for each of the K files, K may be obtained as a solution to the following:

$$\bar{\imath}^{K-1} < m \leq \bar{\imath}^{K} \tag{8}$$

Figure 6 describes the variation in K as a function of m for several values of $\bar{\imath}$. The mean time to retrieve a

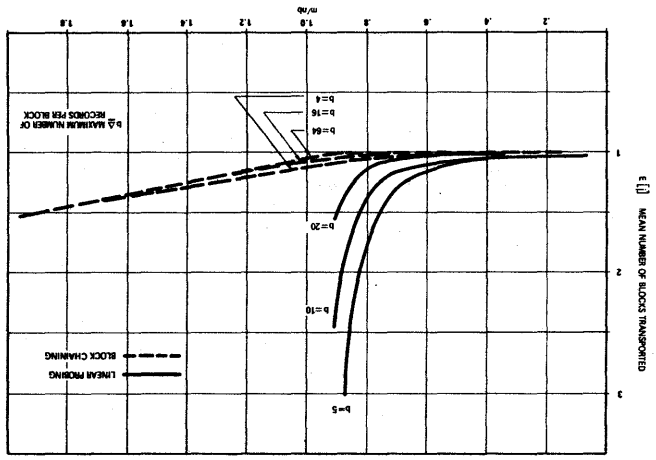


Figure 5—Mean number of blocks transported (Case II)

* To accommodate m records in n blocks, n must be greater than m/b.

** The "next block" is the next physical block as well as the next logical block.

record is simply

$$\bar{T}_r = \underbrace{\sum_{k=1}^{K} \bar{r}_{\text{rand}}}_{\text{Retrieval of index record}} + \underbrace{\bar{r}_{\text{rand}}}_{\text{Retrieval of data record}} \qquad (9)$$

If the top level is not contained within a single block, the mean retrieval time will be a function of the number of blocks comprising said level, and the search strategy employed. Under these circumstances, the results of Equations 3 or 7 may be applied, together with Equation 9, to obtain an expression for $\bar{T}_r$.

CASE STUDY

The results derived in the preceding section may be used to establish the relative merits of the several techniques. To illustrate this procedure, a hypothetical application is presented in the form of a brief case study. It is assumed that the file in question consists of m records. The index file(s), with the exception of the top level (Case III), are maintained on a drum while the main file is kept on disk. The block size is fixed (for all devices) at 1024 bytes. It is further assumed that the data records are 512 bytes in length; a retrieval thus involves a single disk access. Index records are 16 bytes in length (12 bytes for the key plus 4 bytes for the address). Therefore each block has a capacity of 64 index records ($b=64$). The drum and disk are characterized by the following parameters:

Drum		
Mean Latency	$\bar{l}_1$	=17 msec
Data Transfer	x_1	= .43 msec
Disk		
Mean Seek	$\bar{s}_2$	=75 msec
Mean Latency	$\bar{l}_2$	=12.5 msec
Data Transfer	x_2	= 4.16 msec

Case I: Spatial index

In this case, the index file is organized as a sequential file—a physically contiguous string of blocks containing the m index records, ordered by key. We assume for the sake of example that the mean occupancy of a block is 50 records. This corresponds to a utilization of roughly 80 percent ($U \approx .80$). Under these circumstances, the mean time required to retrieve a record is described in Figure 7. For files of 30,000 records or more, the advantage of the binary search is considerable.

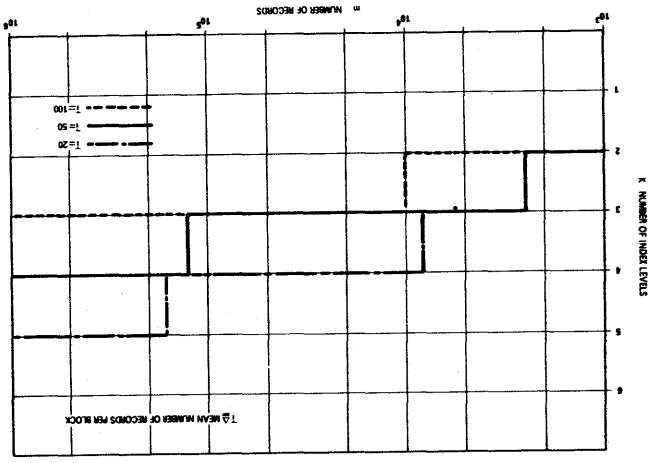


Figure 6—Number of index levels (Case III)

Case II: Calculated index

In this case, the index file is organized as a random file. Where linear probing is employed, the m records are distributed over n blocks. With block chaining, $n' = n + \Delta n$ blocks are allocated. Figure 8 describes the variation in $\bar{T}_r$ as a function of m/nb. The utilization factors, U (Linear Probing) and U' (Block Chaining) are likewise plotted in Figure 8. For a utilization factor of .80, the mean time to retrieve a record, assuming Linear Probing, is 109.1 msec. With Block Chaining (and a utilization of .80) the retrieval time is 115.2 msec. Note that in either case, the retrieval time is independent of the size of the file, dependent instead on the ratio m/nb where n is controllable via τ.

Case III: Tabular indices

In this case, K sequential index files are maintained. It is assumed here that K is selected so that the top level index is contained entirely within a single block. Assuming a utilization factor of .80, K is obtained as the solution to the following inequality:

$$50^{K-1} < m \leq 50^K \qquad (10)$$

We further assume that the top level index (a single block) is kept in core. The time spent searching the top level may therefore be neglected. Hence, the mean retrieval time is given by

$$\bar{T}_r = (K-1)(\bar{l}_1 + x_1) + (\bar{s}_2 + \bar{l}_2 + x_2) \qquad (11)$$

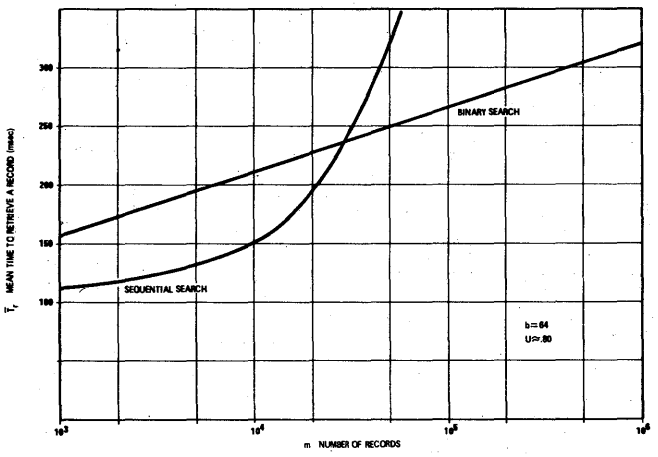


Figure 7—Mean retrieval time (Case I)

Figure 9 describes the variation in $\bar{T}_r$ as a function of m. For example, when $2{,}500 < m < 125{,}000 (K=3)$, the mean time to retrieve a record is 126.5 msec.

SUMMARY

This paper has presented a general overview of several basic file organization techniques. Models were developed to quantitatively examine the retrieval process given the characteristics of the storage device(s) and the details of the file organization. Attention was focused upon retrieval speed in a uni-programming environment. This was done to examine the retrieval process in a "best case" situation where no I/O queueing delays arise. Throughout this analysis it was assumed that once a block was transferred to main memory, any processing time spent searching the block was negligible in relation to the I/O time required to transfer the block from file storage.

As can be noted from the CASE STUDY, retrieval via the Calculated Index technique is generally faster than the other techniques; moreover, $\bar{T}_r$ is independent of the number of records m. However, there are several trade-offs which must not be overlooked. The Calculated Index organization tends to complicate expansion of the file since once the transformation τ is established, allocation upon the media is relatively concrete (especially with Linear Probing). Also such a strategy is not well suited for sequential processing because the transformation τ is inconsistent with the concept of a sorted key list. Block Chaining allows for dynamic expansion of a file, but Block Chaining is generally poorer than Linear Probing when there is considerable deletion activity. This is due to the fact that within a block chain records must contiguously occupy the media (starting with the 1st block of the subfile), thereby increasing the expected number of blocks which must be accessed for record deletion. Moreover, Block Chaining can be substantially less efficient in media utilization than any of the other strategies (especially for large b).

Although retrieval speed via Tabular Indices (multi-level indices) is comparable to that of the Calculated Index, files organized with multi-level indices are not well suited to insert/delete activity due to the fact that the contents of all K sequential index files could be affected by a single change in the main file. However files organized according to this technique lend themselves to restructuring and sequential processing.

As can be noted from the examples, retrieval speed for the Spatial Index case becomes progressively worse than that of the other techniques as the size of the main

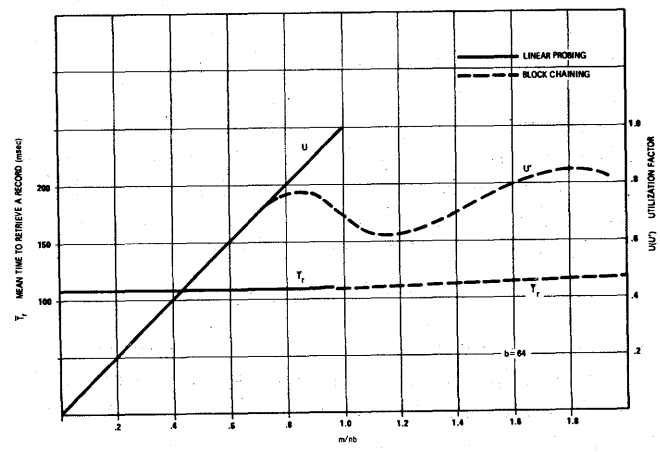


Figure 8—Mean retrieval time (Case II)

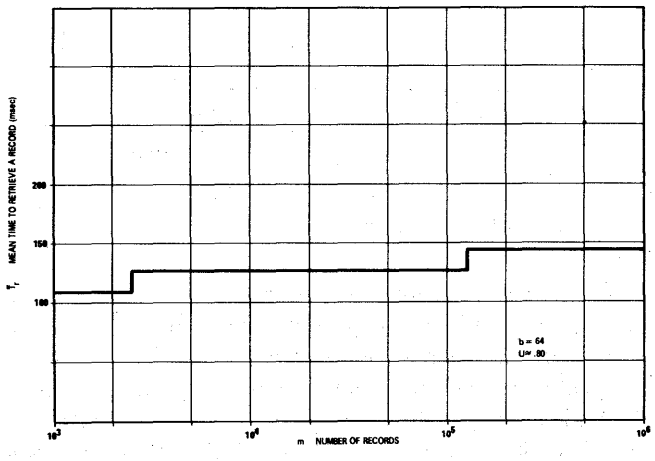


Figure 9—Mean retrieval time (Case III)

file increases. However it can be argued that this organization better supports sequential processing. Yet in multiprogramming environments, unless the file is dedicated, such an advantage is minimal since several independent programs may be concurrently contending for service from the same file I/O device(s), thereby randomly repositioning the access mechanism(s) with the occurrence of each I/O request.

Little, if any, attempt was made to answer questions such as: How should a file be allocated to the media? What level(s) of index should reside in core, on drum and on disk? These questions cannot be answered satisfactorily without considering the frequency with which a file is referenced, and, in multiprogramming environments, queueing delays[2–5] due to other users' requests for I/O. Questions dealing with the subjects of insert/delete time, efficiency of the multi-programming scheduling strategy, sequential processing time, etc., have likewise been ignored. However, performance characteristics such as these can be investigated by extending the framework developed in this paper. For example, to examine multiprogramming effects, the transport time r might be considered to consist not only of the block transfer and device positioning times but also queueing delays due to servicing of other program requests which, in order to maximize I/O throughput, are serviced according to a "shortest-access-time-first" algorithm.[4]

ACKNOWLEDGMENT

The authors wish to thank L. Lam for his programming assistance in the case study.

REFERENCES

1 A J COLLMEYER
File organization techniques
IEEE Computer Group News Volume 3 Number 2 pp 3-12 March/April 1970

2 J ABATE H DUBNER S WEINBERG
Queueing analysis of the IBM 2314 disk storage facility
JACM Volume 15 pp 577-589 October 1968

3 E G COFFMAN
Analysis of a drum input/output queue under scheduled operation in a paged computer system
JACM Volume 16 pp 73-90 January 1969

4 P J DENNING
Effects of scheduling on file memory operations
AFIPS Conference Proceedings Volume 30 pp 9-21 1967 SJCC

5 G E BRYAN J E SHEMER
The UTS time-sharing system: Performance analysis and instrumentation
Proceedings ACM Second Symposium on Operating Systems Principles pp 147-157 1969

6 R MORRIS
Scatter storage techniques
CACM Volume 11 pp 38-44 1968

7 W W PETERSON
Addressing for random-access storage
IBM Journal of Research and Development Volume 1 pp 130-146 1957

8 T C LOVE
The influence of data base characteristics and usage on direct access file organization
JACM Volume 15 pp 535-548 October 1968

9 B H BLOOM
Some techniques and trade-offs affecting large data base retrieval times
Proceedings ACM National Conference pp 83-95 1969

10 M E SENKO V Y LUM P J OWENS
A file organization evaluation model (FOREM)
Proceedings Congress of the International Federation for Information Processing (IFIP) pp C19-C23 1968

APPENDIX A: SEARCH STRATEGIES (SPATIAL INDEXING)

Consider a file α consisting of m records which are distributed over a set θ of n blocks according to the value of each record's identifier, where each block contains at least one record and sufficient space for as many as b records. Let $P[i \mid m, b, n,]$ denote the probability that an arbitrary block within the set θ contains i records of the file (i.e., $b-i$ records are not allocated within the block). The occupancy of any block may be characterized by the mean occupancy $\bar{\imath}$ where, by definition, $\bar{\imath}=m/n$ (i.e., the occupancy of each block is independent of its order within the set θ).

Since blocks within the set θ must contain at least one record and no more than b records, the integral number of blocks n which comprise the file varies from h to m where h is the smallest integer greater than or equal to m/b. Without loss of generality, the fractional utilization U of the storage media may be described by the relation

$$U=m/bn=\bar{\imath}/b \qquad \text{(A.1)}$$

where $h\leq n\leq m$.

Hence the allocation of the file is characterized by the parameters m, b, and n.

Sequential search

Assume that each item in a list of n items is equally probable of being referenced, and that upon any reference to the list, items are sequentially examined until the referenced item is located. Thus given an arbitrary reference to any one of the items, the probability that j items $(1\leq j\leq n)$ must be searched before the desired

item is found is expressed as

$$P_s[j \mid n] = \begin{cases} (n)^{-1} & \text{for } j=1 \\ [(n-j+1)^{-1}] \prod_{k=0}^{j-2} [1-(n-k)^{-1}] & \text{otherwise} \end{cases}$$

whereby

$$P_s[j \mid n] = (n)^{-1} \qquad 1 \leq j \leq n \qquad \text{(A.2)}$$

Thus the expected number* of items which must be searched (given the search is successful) is

$$E_s[j \mid n] = (n+1)/2 \qquad \text{(A.3)}$$

Binary search

Consider a list consisting of n items with each item associated with an identifier which is a unique positive number (e.g., an integer ≥ 1). Assume that items of the list are ordered according to their corresponding numeric identifiers. Now given an arbitrary reference to the list, assume the procedure to locate the referenced item is to successively divide the list into smaller sub-lists, each of which contain the referenced item. At each step of the search the list (or sub-list) is divided in half by examining the item in the middle of the list (or sub-list) and making a decision as to whether or not the item examined is the referenced item and, if not, a binary decision based upon the sequential ordering that determines in which part (the upper half or lower half of the list) the referenced item resides. Thus at each stage of the search if the search is unsuccessful, the size of the sampled list is reduced (roughly by a factor of $\frac{1}{2}$) until the referenced item is found.

With this method the referenced item is guaranteed to be located in a maximum of q search steps where $2^{q-1} \leq n \leq 2^q - 1$ for $n \geq 2$ and $q \geq 2$. If each item is equally probable of being referenced, the probability that j items must be searched (i.e., a search involving j steps before the referenced item is obtained) is given by

$$P_B[j \mid n] = \begin{cases} 2^{j-1}/n & \text{for } 1 \leq j < q \\ 1-(n)^{-1}(2^{q-1}-1) & \text{for } j=q \end{cases} \qquad \text{(A.4)}$$

Employing this result, the expected number of search steps is

$$E_B[j \mid n] = q[1+(n)^{-1}] + [(1-2^q)/n] \qquad \text{(A.5)}$$

* The $E[\cdot]$ notation is used to denote the expectation of the variable enclosed within the brackets.

APPENDIX B: SCATTER STORAGE AND RETRIEVAL STRATEGIES (CALCULATED INDEX)

Consider a file α which is constructed according to a scatter storage technique. Under this method, a transformation τ converts a record identifier into one of n subfile addresses. Corresponding to each subfile there is at least one block allocated on the media. For our purposes, assume that n sequential blocks are initially allocated (one block per subfile) on the media (e.g., contiguous sectors upon a disk storage device).

Now given a reference to an arbitrary record, application of the transformation τ upon the record's key* enables one to immediately calculate which block contains the record provided the block is not filled, whereby only one access is required to retrieve the record.

If a block becomes filled with b records during the construction of the file, there must be some provision to accommodate additional records which under τ map into the same subfile. There are basically two approaches to this overflow problem which for purposes of clarity are designated here as "method 1" and "method 2."

Method 1

Under "method 1" when the kth block is filled then an alternate block is chosen from the remaining $n-1$ blocks** according to a "collision strategy."[6]

For this study, consider as an example of "method 1" the commonly employed strategy known as Linear Probing.[6,7] Under this collision strategy, a record belonging to the filled kth block is placed in the (k modulo $(n)+1$) block provided there is room. If the $k+1$ block is filled then the $k+2$ block is probed, and so it progresses until an unfilled block is found.

Now since the transformation τ is assumed to be perfectly random, the number i of records per subfile may be regarded as a random variable distributed according to the binomial distribution

$$P_C[i \mid m, n] = \binom{m}{i} [(n)^{-1}]^i [1-(n)^{-1}]^{m-i} \qquad \text{(B.1)}$$

with mean m/n.

In order to examine the problem of accessing a file constructed using linear probing, we rely upon the work presented in Reference 7. In an exhaustive study

* Ideally the transformation τ should evenly distribute the m records over the set of n blocks.

** Obviously this can be accomplished providing $m \geq nb$

of Linear Probing, Peterson[7] empirically derived the mean number of block accesses $E_{C1}[j \mid m, b, n]$ required to retrieve an arbitrary record. These results are graphically summarized in Figure 5 which displays the mean number of accesses versus the utilization factor U.

Method 2

The second basic approach to the overflow problem is designated as "method 2." Here there are two classes of blocks—those allocated *a priori* and those allocated because of overflow. Overflow records are kept in separate blocks which are allocated as overflow occurs.

As one example of this method, consider that when a block is filled, another block is allocated to the file and chained to its filled predecessor. Thus a chain of one or more blocks comprise the kth subfile (remember, initially there is one block per subfile). Furthermore, assume that records are inserted into a subfile in the order of their creation, and if any record is deleted from a subfile then all records below the removed record are relocated such that the media is contiguously allocated within the block chain (starting with the 1st block of the subfile). Let us call this strategy Block Chaining.

Hence as opposed to "method 1," overflow is onto blocks other than the original set of n, and subfiles are physically separated (i.e., no block contains records from more than one subfile).

Let Δn denote the number of blocks allocated for overflow. Then the total number of blocks n' over which the file is distributed is given by $n' = \Delta n + n$. Now employing the result of equation B.1, the probability that there are η additional blocks appended to the block chain of any arbitrary subfile is given by

$$P_{C2}[\eta \mid m, b, n] = \sum_{i=\eta b+1}^{(\eta+1)b} P_C[i \mid m, n] \qquad \text{(B.2)}$$

where $0 \leq \eta \leq h-1$, with h denoting the smallest integer greater than or equal to m/b, as before. Hence the mean number of blocks is

$$E_{C2}[n' \mid m, b, n] = n\left(1 + \sum_{\eta=1}^{h-1} \eta P_{C2}[\eta \mid m, b, n]\right) \qquad \text{(B.3)}$$

and the storage utilization factor becomes

$$U' = m/(bE_{C2}[n' \mid m, b, n]) \qquad \text{(B.4)}$$

where $U' \leq U$.

Now given an arbitrary reference to any record within the file, the probability that j block accesses* are required to retrieve the referenced record is expressed

$$P_{C2}[i \mid m, b, n] = \begin{cases} \left(\sum_{i=(j-1)b+1}^{jb} \{[i-(j-1)b]/i\} \times P_C[i \mid m, n] + b \sum_{j=jb+1}^{m} (1/i) \times P_C[i \mid m, n] \right) \\ \quad \times \{(1-P_C[0 \mid m, n])^{-1}\} \\ \quad \text{for} \quad 1 \leq j \leq h-1 \\ \sum_{i=(h-1)b+1}^{m} \{[i-(h-1)b]/i\} \times P_C[i \mid m, n] \\ \quad \times \{(1-P_C[0 \mid m, n])^{-1}\} \\ \quad \text{for} \quad j = h \end{cases} \qquad \text{(B.5)}$$

Therefore with Block Chaining the expected number of blocks which must be accessed is

$$E_{C2}[j \mid m, b, n] = \sum_{j=1}^{h} j P_{C2}[j \mid m, b, n] \qquad \text{(B.6)}$$

APPENDIX C: I/O DEVICE MODELS

Without loss of generality, let us consider two basic file I/O devices designated as "device 1" and "device 2" where device 1 corresponds to a drum (fixed head device) and device 2 is a disk (moving arm device). Given these devices, a relevant performance index is the time r required to transport (retrieve and transfer) a block from the storage media to main memory. In the next sections, relatively simple mathematical models are developed to quantitatively express r for each device type.

Before proceeding to this analysis, it is convenient to establish some common notation. Let $p_y(t)$ represent the stationary probability density function for a random variable y with respect to the independent variable t (where t denotes time), and let $Y(s)$ be the Laplace-Stieltjes transform of the distribution for y (i.e., $Y(s) = \int_0^\infty e^{-st} p_y(t)\,dt$ and $Y(0) = 1$). This notation is employed in that which follows.

Device 1: Drum storage

Consider a drum is addressed to transfer a block of information. Since with a drum no head motion is

* It is assumed that blocks within the chain (in which the referenced record resides) are examined in the order of allocation.

required to position the read/write head(s) over the addressed track(s), the drum transport time r_1 is determined by computing the sum of drum latency l_1 and block transfer time x_1. Thus, assuming that l_1 and x_1 are independent

$$R_1(s) = L_1(s) X_1(s) \tag{C.1}$$

and the mean transport time is given by

$$\bar{r}_1 = -dR_1(s)/ds \,|_{s=0} = \bar{l}_1 + \bar{x}_1$$

Here, if the address is random, it is reasonable to assume that the latency delay is uniformly distributed over 0 to w_1, where w_1 denotes the drum rotation time. Hence with z_1 equal to the number of blocks that can be transferred in one drum rotation

$$\bar{r}_1 = w_1/2 + w_1/z_1 = [w_1(z_1+2)]/2z_1 \quad \text{If address is } \textit{random} \tag{C.2a}$$

If the drum is already positioned when it is addressed (as is the case when the access is sequential), there is no positioning time ($l_1 = 0$), whereby

$$\bar{r}_1 = w_1/z_1 \quad \text{If address is } \textit{sequential} \tag{C.2b}$$

Device 2: Disk storage (moving arm)

Now assume a disk is addressed to transfer a block of information. A disk transfer request is analogous to a drum request except there is an additional seek time s_2 required for arm motion to position the read/write head(s). Hence assuming disk seek time s_2, latency time l_2, and block transfer time x_2 are independent, the transport time r_2 for device 2 is characterized by

$$R_2(s) = S_2(s) L_2(s) X_2(s) \tag{C.3}$$

Therefore if the disk address is random, the mean transport time is expressed

$$\bar{r}_2 = (\bar{s}_2 + w_2/2 + w_2/z_2) \quad \text{If address is } \textit{random} \tag{C.4a}$$

where w_2 is the disk rotation period; z_2 is the number of blocks transferrable per rotation; and $\bar{s}_2$ is derived from the seek probability distribution

$$\bar{s}_2 = -dS_2(s)/ds \,|_{s=0}$$

If the disk address is non-random, but rather sequential (i.e., there is no positioning time whereby $s_2 = l_2 = 0$) then

$$\bar{r}_2 = w_2/z_2 \quad \text{If address is } \textit{sequential} \tag{C.4b}$$

APPENDIX D: RETRIEVAL

Now assume that a given I/O device (device 1 or device 2) is addressed j times to retrieve j blocks of information with all addresses being either all random or all sequential.* Let γ represent the total time required to transfer and retrieve the j blocks, then

$$\Gamma(s \mid j) = (R(s))^j \tag{D.1}$$

where $R(s)$ is the Laplace-Stieltjes transform of the block transport time r for the respective device.

Then removing the conditioning upon j

$$\Gamma(s) = \sum_{\text{ForAll } j} P[j](R(s))^j \tag{D.2}$$

Hence

$$\bar{\gamma} = -d\Gamma(s)/ds \,|_{s=0} = -\sum_{\text{ForAll } j} jP[j](R(s))^{j-1} \times dR(s)/ds \,|_{s=0} \tag{D.3}$$

But

$$R(0) = 1, \qquad -dR(s)/ds \,|_{s=0} = \bar{r},$$

and

$$\sum_{\text{ForAll } j} jP[j] = E[j] = \bar{j}$$

whereby

$$\bar{\gamma} = \bar{j}\bar{r} \tag{D.4}$$

* Hence all j accesses are independent and indentically distributed.

Analysis of a complex data management access method by simulation modeling

by V. Y. LUM, H. LING and M. E. SENKO

IBM Research Laboratory
San Jose, California

INTRODUCTION

The typical paper on file organization presents a qualitative discussion of a proposed search structure. In a very few instances, the writer may include one or two numbers to indicate system performance. In reading these papers, one is led to the impression that file design is an extremely simple problem and that intuitive guesses can easily provide optimal answers. However, the sophisticated file designer knows from experience that there are many relevant parameters and that they can drastically affect performance. This knowledge can, however, be frustrating because he cannot afford the machine time and resources to obtain a good design by performing an adequate parametric study on possible file organizations. (For example, setting up a large file in just one configuration can require several hours of machine time and several man days or weeks.)

In this paper, we present an example of an alternate method, the use of a comprehensive simulation model to obtain adequately accurate answers. Such a procedure can reduce the actual cost of a parametric study by factors of tens, hundreds, or thousands. The example is a parametric study of a complex indexed sequential access method involving thousands of machine runs.

A smaller version of this study would be suitable for the design of a specific file while a larger version would be required to provide the basis for a file design handbook for the access method. In any case, this study

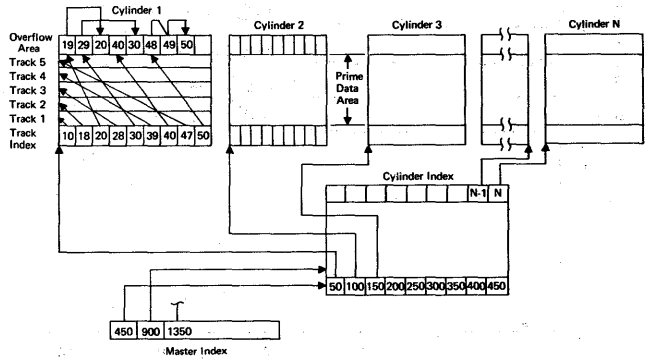
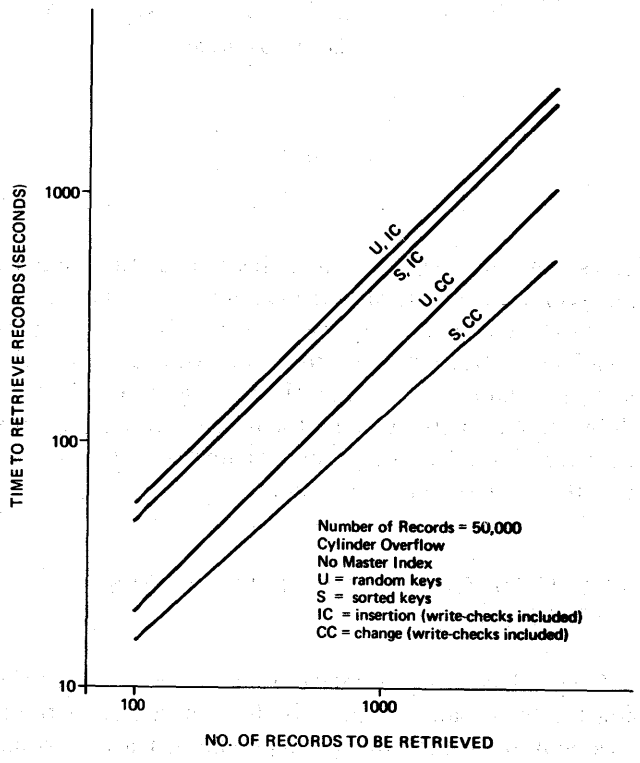
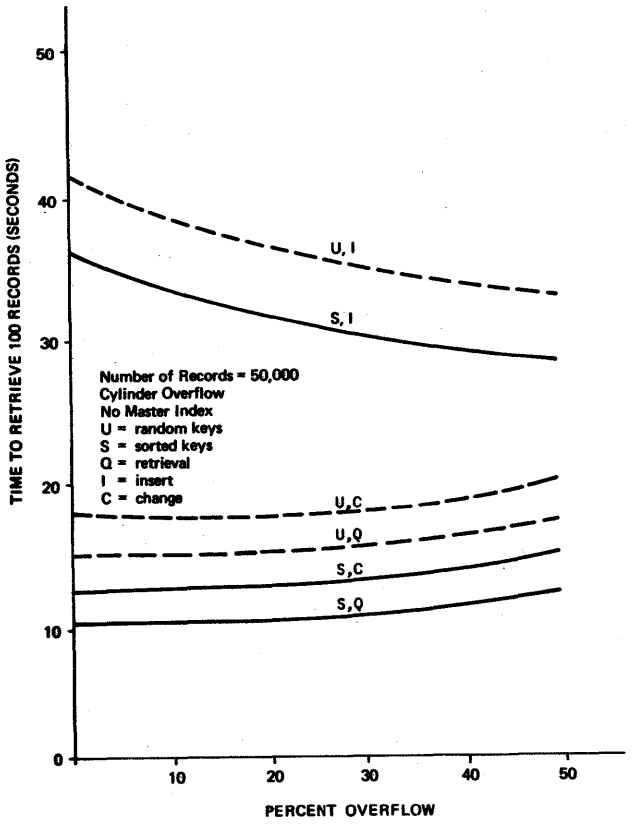


Figure 1



Graph 1



Graph 2

gives, for the first time, an indication of the complex behavior of an actual data management access program.

FOREM I, which was used for the study, contains 300–500 FORTRAN statements dealing with the analytic evaluation of access time and storage layout for different parameter values. Each run consumes on the average about 10 seconds of machine time and a few minutes of designer set-up time.

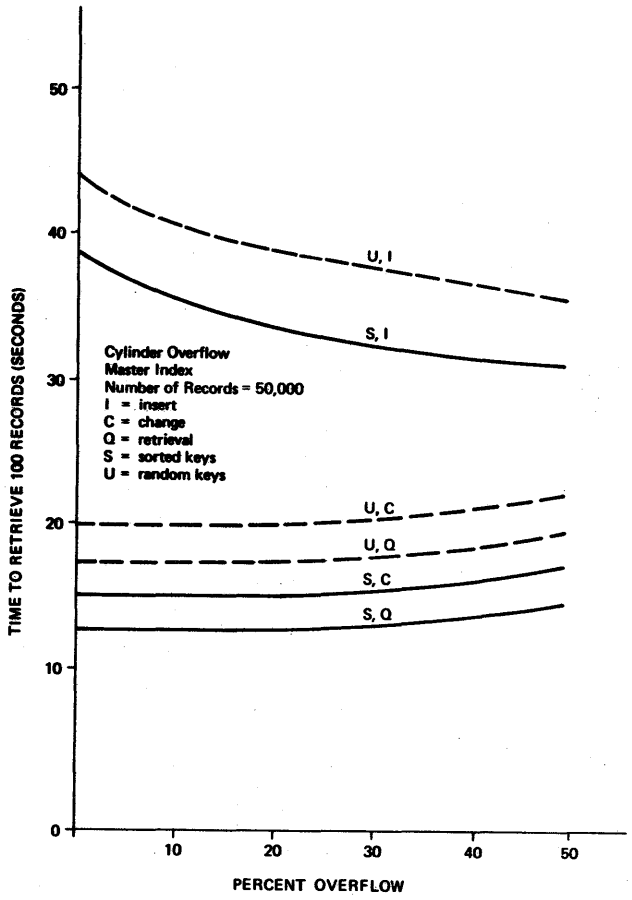
THE INDEXED SEQUENTIAL ACCESS METHOD

The indexed sequential access method is one of the few fundamental, qualitatively different access methods (sequential, direct, and perhaps chained access being other possibilities). It is based on data and a hierarchy of indexes that are sequenced on a particular identifier. The method has been programmed by manufacturers and users in a number of specific implementations. In Figure 1, we present the physical storage layout of one specific implementation. Its variable parameters include (a) number of index levels, (b) device placement of index levels, (c) device placement of data, (d) block size of data, (e) amount of overflow, (f) device placement of overflow, etc. Parameters which are fixed for a specific file design include (a) actual method of access-direct or buffered sequential, (b) number of buffers, (c) type and number of transactions, (d) file size, (e) record size, etc.

The paper is divided into three sections and an appendix. The sections are:

a. The characteristics of direct access through the indexes;
b. The characteristics of sequential search of the data;
c. A comparison of the two methods.

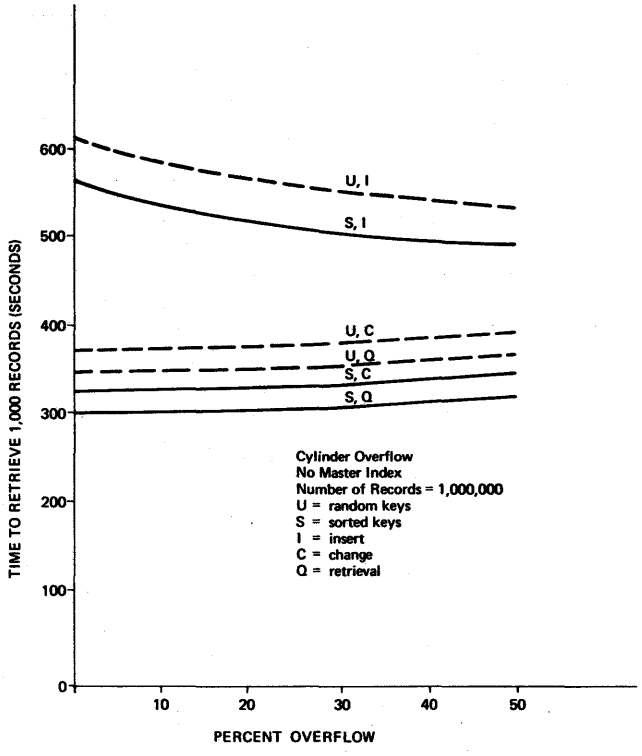
The appendix of the paper presents comparisons of model runs with actual computer runs to illustrate the accuracy attainable with the model's approach.



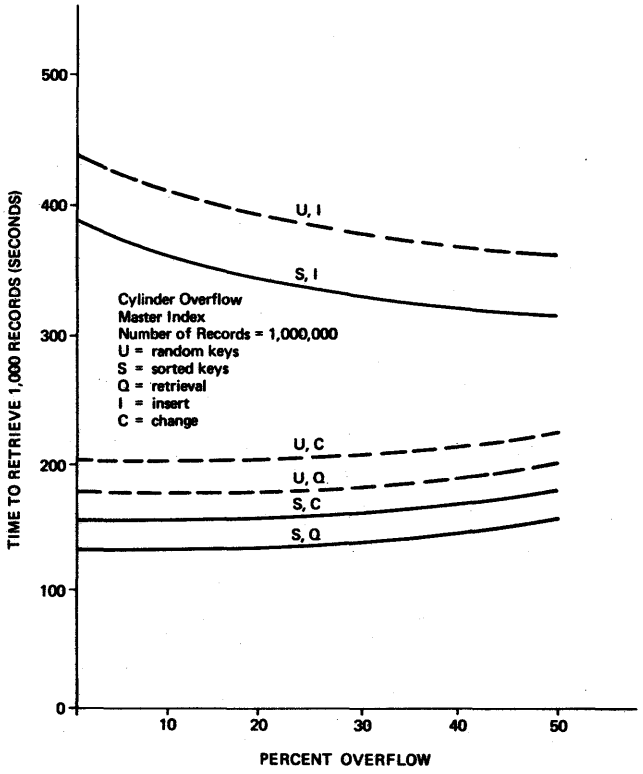
Graph 3

DIRECT INDEXED ACCESS

In direct indexed access, the data management routine is presented with the identifier of a particular record. The identifier is compared sequentially against the key entries in the highest level index. When the identifier is matched, equal or low, descent is made to a section of the next lower level index by means of an address pointer associated with the appropriate key. At the track level index, for every prime track* there are two index entries: one containing the highest key in the prime track and one containing the highest key of the overflow associated with the prime track. Search of the prime track involves sequential processing of blocked records on the track. Search of the overflow involves following chains through the records that have been pushed off the prime track by insertions. The critical parameters which we studied were:



Graph 4

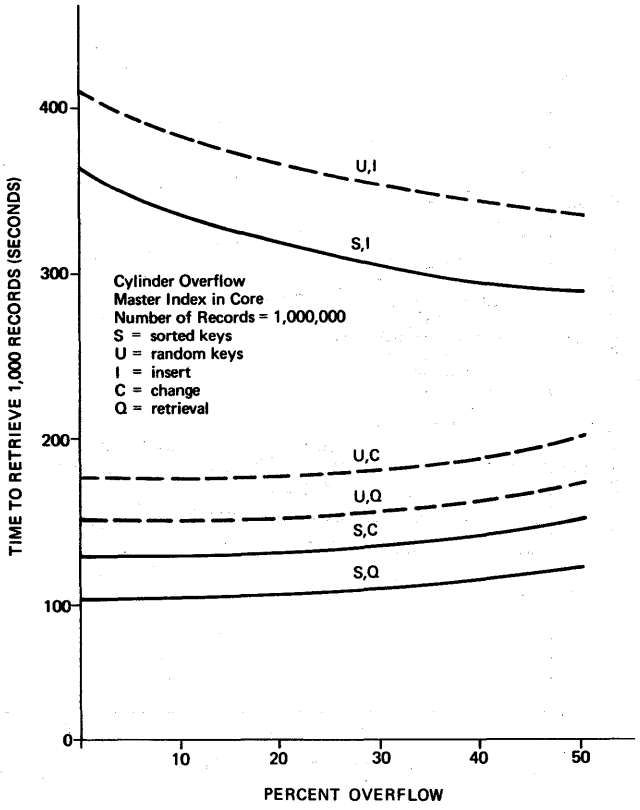


Graph 5

1. File size (two files: 50,000 and 1,000,000 records);
2. Number and placement of index levels (Master Index (MI = M), no master index (None), and master index in core (MC));
3. Overflow configurations:
   a. Overflow in the same cylinder as the prime track or cylinder overflow (IE);
   b. Overflow in the same pack (IB);
   c. Overflow on a separate pack (IS);
4. Percent of overflow (eleven values: 0–50 percent at 5 percent intervals);
5. Transaction types (query (Q), change (C or CC), and insert (I or IC)). (The second C indicates that a write check is made.)
6. Input key stream (random SU = U or sorted SU = S);
7. Number of records retrieved.

The records were 725 bytes long and were stored in unblocked form on an IBM 2314 disk storage device. The indexes were on separate packs from the data and

* A prime track is a track dedicated during the loading process to contain data records whose keys fall within a particular value range. When inserts are made and no space is available, records will be pushed off the track into an overflow area. These records are said to be overflow records.



Graph 6

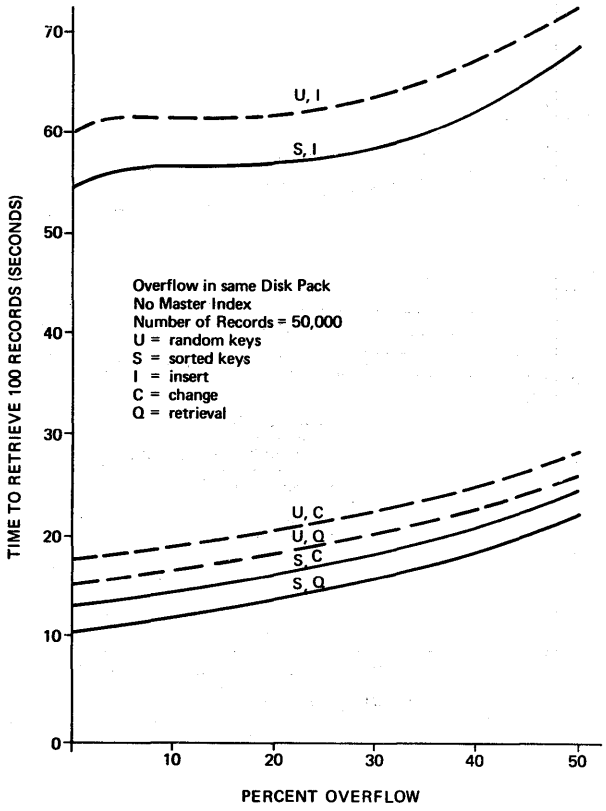
processing time for the qualified records was assumed to be negligible. Even though it was apparent that a large number of model runs were involved, it is also clear from the immediately previous statements that all possible parameters were not varied.

Number of records retrieved

Graph 1 indicates the general behavior of various transaction types as the number of records retrieved in a transaction is varied. In the unsorted key case, the average time per record remains constant, independent of number of records; the sorted key case diverges from this curve because the access arm requires smaller and smaller steps to transverse the data disk pack as more records are retrieved. In these runs, index blocks were not buffered so the divergence is not as great as it would be if the access arm on the index pack could march across the index files also. Insert requires more time than change because records must normally be moved to make room for the inserted record.

Index structure

Index structure tradeoffs can be considered by consulting Graphs 2, 3, 4, 5 and 6. Graphs 2 and 3 indicate that a master index* is not useful for a small file while Graphs 4 and 5 indicate that the opposite is true for large files. This in itself is a relatively obvious conclusion, however, the location of the decision point between the two file sizes is of more interest. This decision point depends on whether index entries are placed in full track blocks (for performance) or in smaller blocks (to conserve core storage). For full track blocking with reasonable key sizes, a master index becomes useful only after the cylinder index exceeds four tracks in length (for an IBM 2314, this is equivalent to seven disk packs). At the other extreme, where each



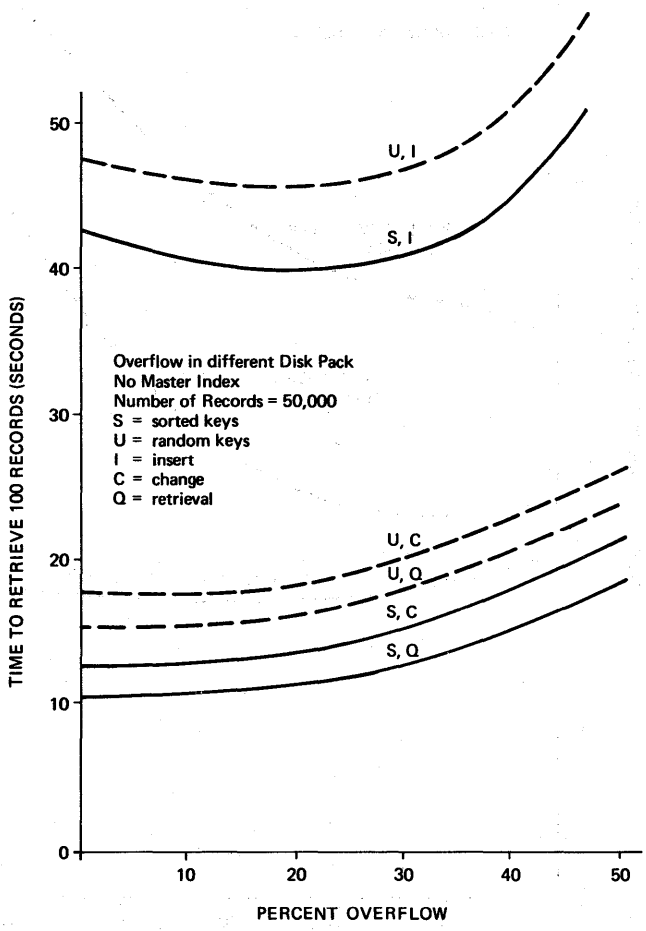
Graph 7

* Master index is an index to the cylinder indexes (Figure 1).

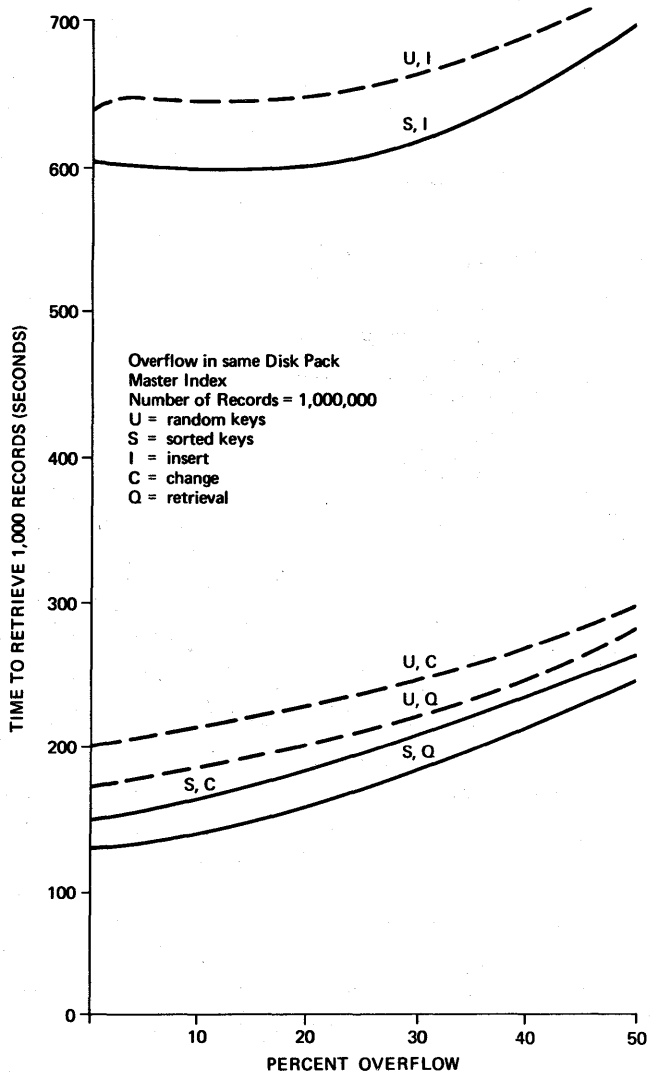
entry occupies a separate physical block, the decision point lies at two cylinder index tracks (corresponding to about one-half of a 2314 disk pack). For the large file, the differences between these choices can be quite significant:

a. master index, full track index blocking ~130 seconds;
b. no master index, full track index blocking ~300 seconds;
c. no master index, one index entry per block ~1600 seconds.

The permanent storage of the master index in core provides an additional 20 percent improvement over case (a) (Graph 6).



Graph 8

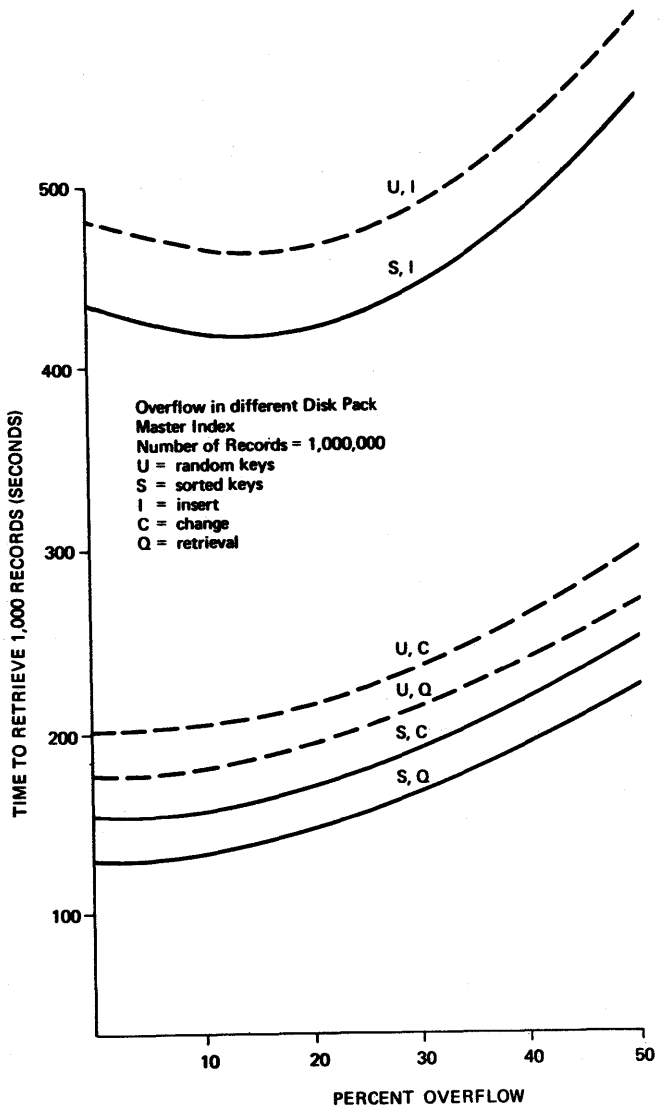


Graph 9

Overflow configuration

Graphs 7–10 supplement Graphs 2 and 5 (the most desirable index configuration for IE) to provide a picture of performance behavior by overflow configuration. In all overflow cases, the numbers of logical and physical records per track are significant parameters in predicting performance.

All operations are affected by the number of logical records per track; even small percentages of overflow result in long overflow chains when there are one-hundred or more records per track. On the other hand, the number of physical records per track primarily

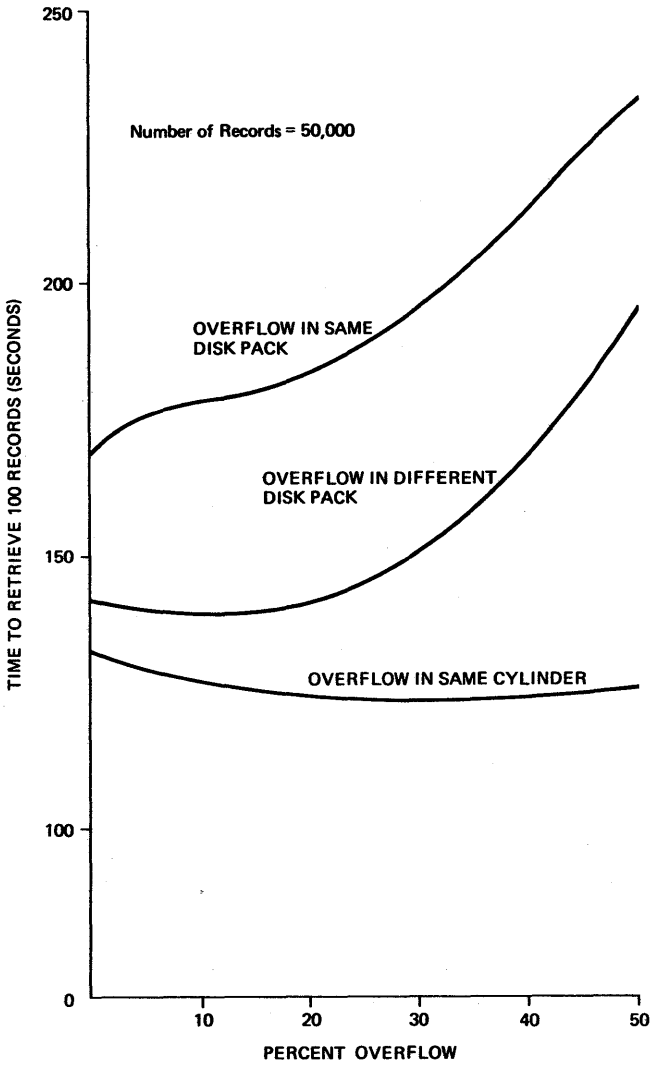


Graph 10

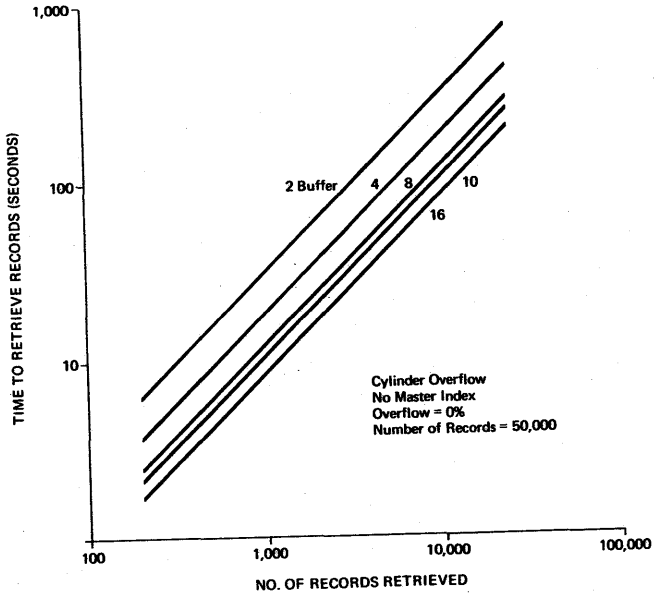
influences insertion behavior. When room must be made for inserts on the prime tracks, the following records must be rewritten block by block until the last record is pushed into overflow. The penalty for rewriting large numbers of physical blocks on the prime track is so drastic that performance generally improves as overflow initially increases, because insertion into overflow is less costly. The surprising fact is that insertion performance will normally improve until the number of blocks in the overflow chain is twice the number of blocks on the prime track.

As expected, cylinder overflow (IE) generally provides the best performance because no additional arm motion is required to access the overflow area. This performance advantage is somewhat compromised by the sensitivity of this configuration to insertions that are not uniformly distributed over all the cylinders. Since enough space must be reserved in every cylinder to hold the maximum number of inserts per any cylinder, there can be extreme space wastage in those cylinders which have little insertion activity. This problem can, of course, be eliminated by combining IE with one of the other overflow configurations, on same pack (IB) or on separate pack (IS), to handle unusually dense insert activity.

The differences between same pack and separate pack are less significant than their differences with respect to cylinder overflow. In general, performance will be worse than separate pack for small amounts of overflow, but



Graph 11



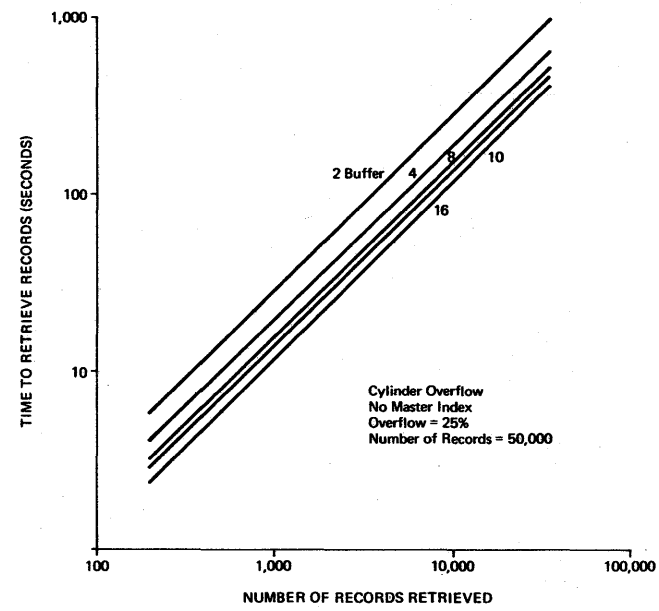
Graph 12

eventually will be better for very large amounts. This is because the initial arm movements to overflow for same pack overflow will be across half the prime area and a portion of the overflow. Arm movements for chain following inside the overflow area will be relatively small. In the case of separate pack overflow, the initial and subsequent arm motions will average one-half the number of cylinders in the overflow. For amounts of overflow exceeding one-half pack in size, these longer subsequent motions will dominate performance.

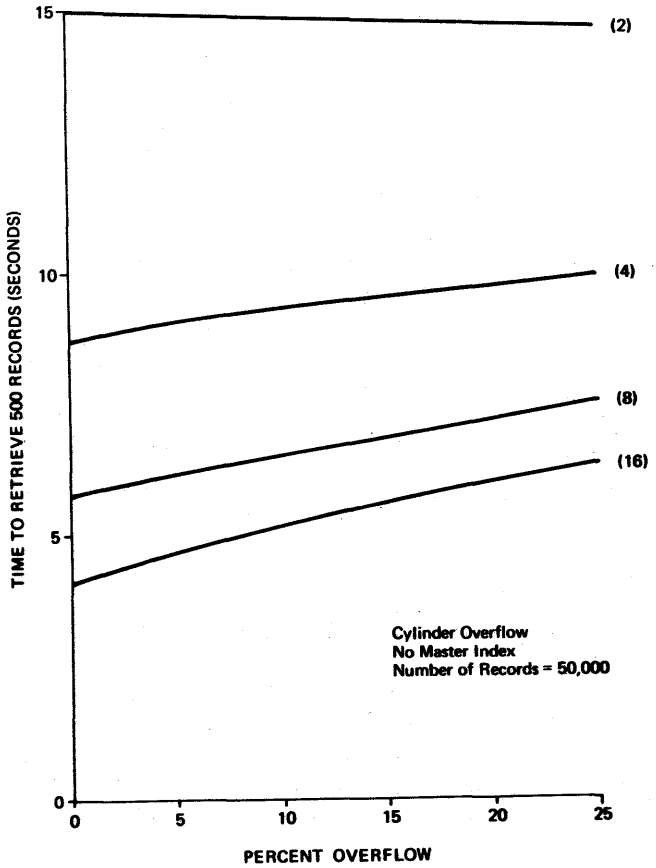
Designing a file with overflow

It is generally believed that overflow hampers performance. In fact, since insertion performance often improves with increased overflow, optimum total performance may be obtained when there is a certain amount of overflow in the file. The optimum can be determined by weighting each of the individual curves for retrieval, update, and insertion by the percentage of transactions of that type. When the curves are added together, the minimum on the total curve will lie at the optimum overflow percentage.

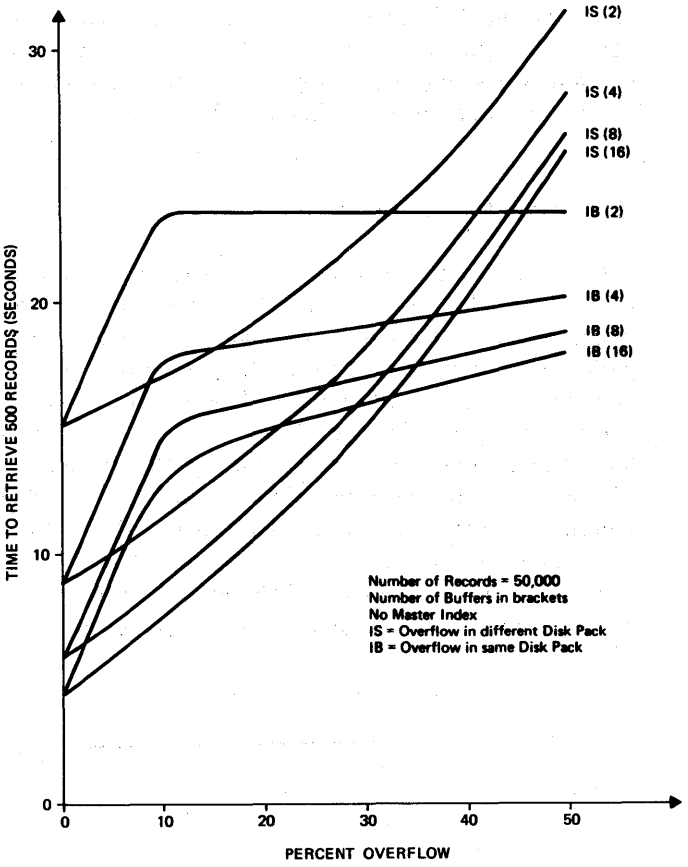
For example, in the case of the small file without a master index, we will assume that all transactions



Graph 13



Graph 14



Graph 15

involve 100 qualified records, and transactions are evenly distributed among updates, retrievals, and insertions with random as well as sorted keys. Graph 11 presents total performance curves for the three types of overflow allocation in the small file. Cylinder overflow (IE) performance is optimum with 25–45 percent overflow and separate pack overflow (IS) performs best at 10–15 percent overflow. The optimum for same pack overflow (IB) generally occurs at zero percent overflow.

General

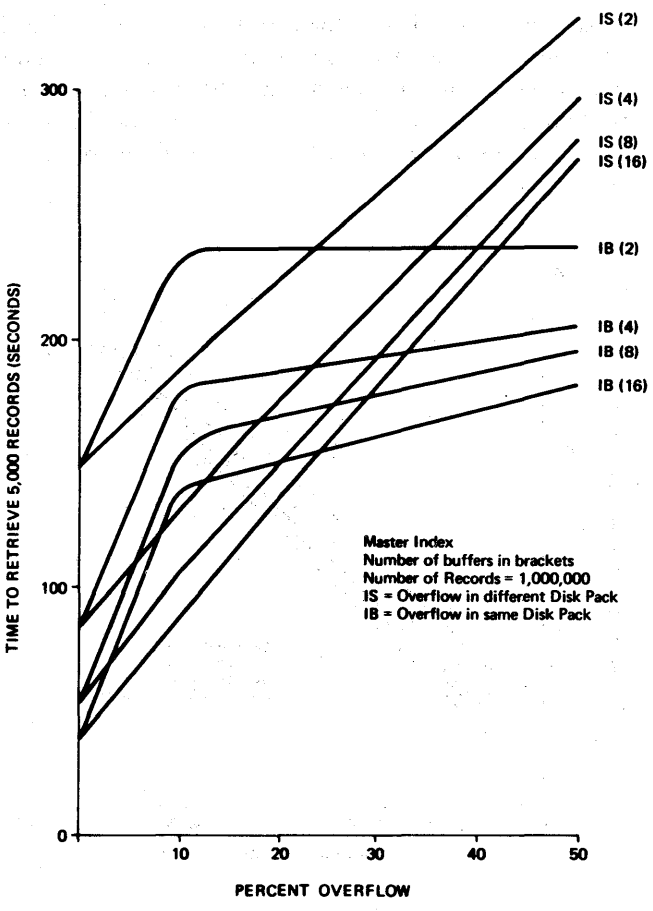
In all test cases, the indexes and data were on different disk packs; and record accesses driven by random key input strings took significantly longer than accesses driven by sorted key input strings. These differences would be marginal if the indexes and data were located in the same pack.

While update-in-place characteristics with or without write-check are very similar to retrieval characteristics since they involve only one or two added disk rotations, the use of write-check in record insertion creates entirely different characteristics. It can be extremely expensive if there are a large number of blocks on the prime track. Nonetheless, it is especially needed in insertion to protect the correctness of the rewritten data.

THE BUFFERED SEQUENTIAL ACCESS PROCESS

In this process, the system is presented with an identifier and finds, by means of an index search, the location of the record having that identifier or the next highest identifier. At this point, it begins a buffered sequential search of the data, pausing at the end of each prime track overflow area to access the track index.

For this study, we have assumed a particular implementation. That is, on the prime track, one-half the total number of buffers may be active in a chained read or write operation at any one time. If the total number of buffers is equal to twice the number of physical blocks on a track, then a complete track can be read or written in one revolution. Overflow tracks, on the other hand, are accessed one physical block at a time. When there is contention for reading and writing services, the principle of first-in-first-out is applied.



Graph 16

Number of records retrieved

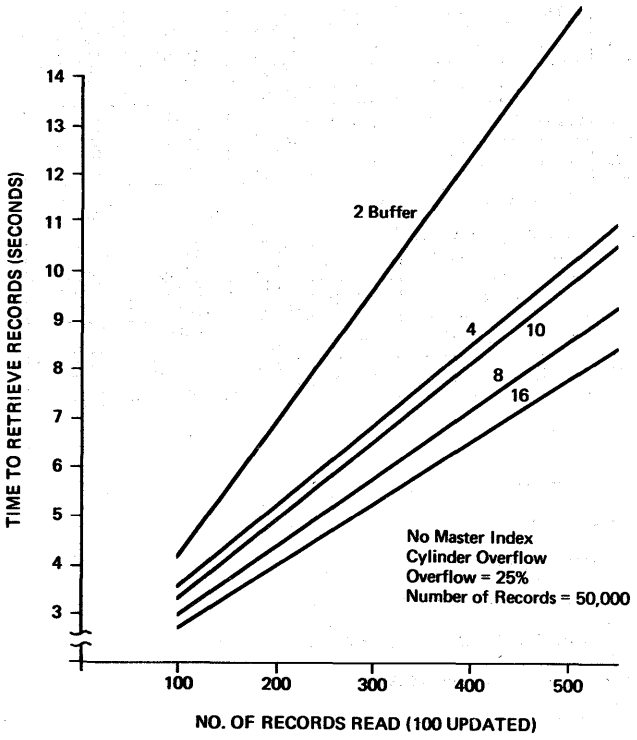
Graphs 12 and 13 indicate the general performance behavior of the access process for various numbers of records retrieved. For a given number of buffers and large numbers of records retrieved, it is an unexceptional linear function. These curves will, however, become more horizontal for fewer numbers of records, because the initial index search will be a more important factor in average access time per record. For similar reasons, the device placement of the indexes is only significant when small numbers of records are accessed.

While the effect of the number of buffers will be discussed later, it is interesting to note that large numbers of buffers are most useful for small amounts of overflow.

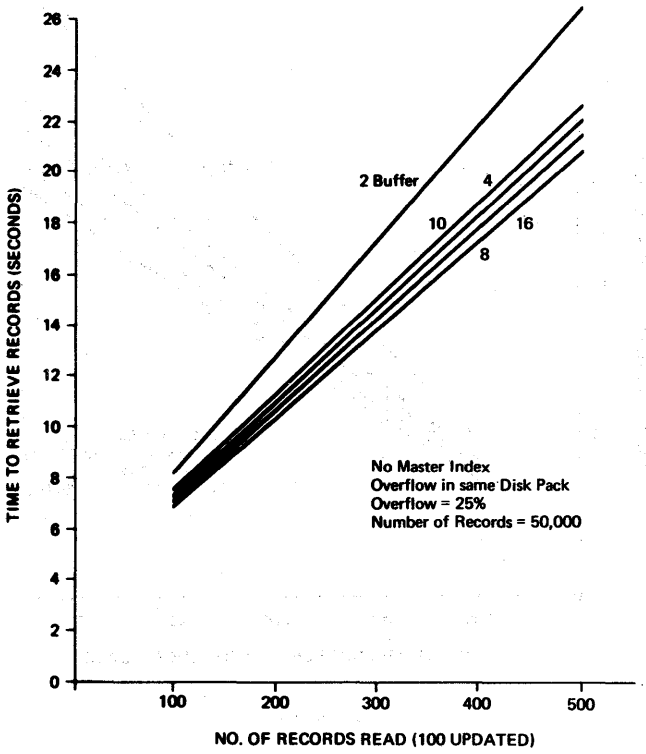
Overflow configuration and overflow percentage

Graphs 12 and particularly 13 and 14 indicate that sequential performance is significantly affected by the amount of overflow present in the file. Arm motion to and in the overflow area is primarily responsible for the rapid change in performance characteristics.

The slope of the cylinder overflow (IE) curves is determined by the differences in access time between



Graph 17



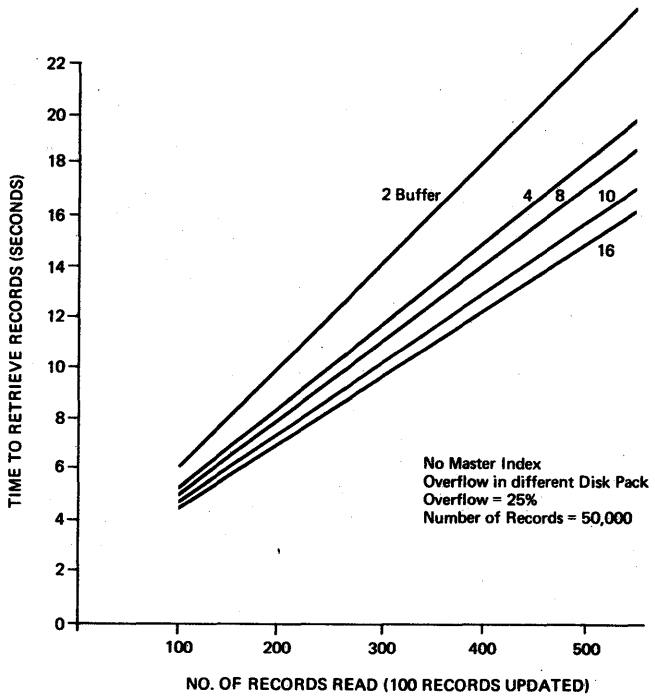
Graph 18

prime area records and overflow area records. This, in turn, is determined by the number of records that can be retrieved in one revolution from the prime area because accessing in the overflow area is always at one record per disk revolution. The primary factors in this determination are prime area record blocking and buffering. The slight downward slope of the cylinder overflow (IE) curve for two buffers is due to the fact that larger numbers of overflow records reduce the necessity for reading index tracks.

The knee in the pack overflow (IB) curves will occur at the overflow percentage where there is one overflow record per prime track. In these tests we have assumed that the overflow records are uniformly distributed over the prime tracks; if we had not, then the knee in the curve would be less sharp. As can be seen for the present experimental configuration, pack overflow begins to outperform separate overflow (IS) when each prime track has about three overflow records associated with it.

Buffers and update performance

In the case of retrieval discussed above, any increase in the number of buffers always causes the timing curves to shift downward, but parallel to their prior locations



Graph 19

(Graphs 12 and 13). When some fraction of the records are updated, and therefore rewritten, there need not be a regular increase in performance as the number of buffers is increased.

In Graphs 17, 18 and 19, as the number of buffers is increased from 2 to 8, the time to read x records and update $y \leq x$ of them decreases regularly. However, a further increase up to, but less than, 16 buffers reduces overall performance. The reason for this phenomenon lies in the interference of seeks for reading and writing of data. When the capacity of the buffers available is less than, or equal to, one-half of a track (in this case, 8 buffers or less), the access system can both write and read $n/2$ blocks in a single revolution (n is the number of buffers available). These two operations cannot be smoothly interspersed when ½ track < the capacity of the buffers <1 track.

In the above runs, the record processing time was not a significant factor. If processing time is significant, then instances will occur where the 2 buffer configuration will perform better than the 8 buffer one. A detailed analysis of these situations is quite involved and is best performed by simulation models.

GENERAL CONSIDERATIONS

Choice of access method

In certain special cases, particularly when the records relevant to a search are confined to a small area of the file, the designer may use either basic direct or buffered sequential search. We provide here an example situation.

If the overflow for the small file is organized on a cylinder overflow (IE) basis and the input keys are sorted, the basic direct access method will require 10 seconds to access 100 records. (See Tables I and II.) The queued sequential access method, using 10 buffers, can retrieve about 1,000 records in the same time. In this case, if better than one record in 10 is pertinent to the query and processing time is insignificantly small, then sequential access will provide better performance.

Generally speaking, if p is the number of records which must be read sequentially to find a qualified one, t_q, the average time to read a record in buffered sequential mode and t_b, the average time to read a record in basic random mode, then the queued mode is more efficient if $t_b > p \cdot t_q$. (This formula is most appropriate

Retrieval & Update Time (sec.)

Table I

over-flow	QISAM			BISAM		
	retrieve 500 records			retrieve 100 records		
	IE	IS	IB	IE	IS	IB
0	4.2-15	4.2-15	4.2-15	10(s) 15(u)	10(s) 15(u)	10(s) 15(u)
5	4.7-15	5.7-16	8.4-18	10(s) 15(u)	10(s) 15(u)	11(s) 16(u)
10	5.1-15	7.3-17	13-23	10(s) 15(u)	10(s) 15(u)	11(s) 16(u)
25	6.4-15	12-21	15-23	10(s) 16(u)	12(s) 17(u)	14(s) 19(u)
	retrieve 500 records update 100 records			update 100 records		
0	4.7-15	4.7-15	4.7-15	13(s) 18(u)	13(s) 18(u)	13(s) 18(u)
5	5.4-15	6.5-16	9.9-19	13(s) 18(u)	13(s) 18(u)	13(s) 18(u)
10	6-15	8.5-17	19-25	13(s) 18(u)	13(s) 18(u)	14(s) 19(u)
25	7.8-15	15-22	21-26	13(s) 18(u)	14(s) 19(u)	16(s) 21(u)

s = sorted keys

u = unsorted keys

No Master Index

number of records = 50,000

IS = Overflow in different Disk Pack

IB = Overflow in same Disk Pack

IE = Overflow in same Cylinder

Table I

when there are many records to be read because the initial reading of the index in buffered sequential mode can affect t_q substantially.)

To approximately determine t_q, let b be the number blocks per data track and T the track revolution time. Assuming the minimum number of buffers,

$$t_q \simeq (1.5 \times T)/b + T.$$

The factor of 0.5 represents the cost of possible revolutions. Thus, in the case of the sample files, the time to read a record is

$$t_q \simeq (1.5 \times 25)/8 + 25 \simeq 30 \text{ ms.}$$

If the file has no master index, t_b can be estimated by

$$t_b \simeq 2T + T_c + (N_c \cdot T)/2 \quad \text{(cylinder index and data on separate packs)}$$

$$\simeq 2T + 2T_c + (N_c \cdot T)/2 \quad \text{(cylinder index with data)}$$

where T_c is average cylinder search time of the file, and N_c is the number of cylinder index tracks.

$$t_b \simeq 4T + T_c + T_{cc} \quad \text{(cylinder index and data separate)}$$

$$\simeq 4T + 2T_c \quad \text{(cylinder index with data)}$$

where T_{cc} is average cylinder search time in the cylinder index. For the small file, we have $N_c = 1$. Thus, $T_b \simeq 2 \times 25 + 75 + 12.5 \simeq 138$. Reading 100 records in the basic direct mode requires approximately 14 seconds as confirmed by our measurement (Table I). Thus, if $p \simeq 5$ to 10, then the buffered mode and the basic direct mode provide similar performance.

Variation of hardware parameters

The results presented in this paper are for a particular device; it is, however, of interest to understand the effect of changes in hardware parameters, such as access arm speed, track size, rotational speed and processor speed.

Of these parameters, access arm speed is the most independent of the others in its effect on performance. In the basic access method for typical configurations, a 100 percent increase in arm speed will result in about a 20 percent improvement in total performance. While increased arm speed will significantly narrow the difference in performance between the direct indexed access processes for various overflow schemes, sequential performance will only be affected when large amounts of overflow exist in pack and separate overflow configurations.

Track size, rotational speed, and central processor speed do, however, interact in a complex fashion with regard to the loss of revolutions. Increases in CPU speed generally will result in no performance deterioration and they may improve performance by saving previously

Retrieval & Update Times (sec)

Table II

over-flow	QISAM retrieve 5000 records			BISAM retrieve 1000 records		
	IE	IS	IB	IE	IS	IB
0	40-146	40-146	40-146	130(s) 176(u)	130(s) 176(u)	130(s) 176(u)
5	44-146	64-190	85-189	130(s) 176(u)	132(s) 180(u)	136(s) 180(u)
10	49-148	92-190	138-237	130(s) 177(u)	138(s) 183(u)	143(s) 190(u)
25	61-149	160-242	155-237	133(s) 180(u)	156(s) 204(u)	169(s) 214(u)
retrieve 5000 records update 1000 records				update 1000 records		
0	45-146	45-146	45-146	154(s) 202(u)	154(s) 202(u)	154(s) 202(u)
5	51-146	95-190	101-193	154(s) 202(u)	157(s) 204(u)	161(s) 208(u)
10	59-148	152-241	199-256	154(s) 202(u)	161(s) 208(u)	168(s) 215(u)
25	76-149	191-264	222-267	158(s) 205(u)	181(s) 229(u)	194(s) 240(u)

number of records = 1,000,000

Master Index Exists

s = sorted keys

u = unsorted keys

IS = Overflow in different Disk Pack

IB = Overflow in same Disk Pack

IE = Overflow in same Cylinder

Table II

lost track revolutions. Track size and rotational speed will normally result in gradual improvements in performance, except in the cases where the CPU can no longer complete processing in time to access the next record. These cases will result in major discontinuous deteriorations in performance through lost revolutions.

Other parameter changes

The size of the records in a file influences performance considerably. For smaller record sizes, the timing curves will have a larger slope at all points and the intersections with the time axis will be lower. If the record size is very small, a slight increase in overflow percentage will degrade performance tremendously. A larger record size shows exactly the opposite effect. Here the performance curves will intersect the axis at a higher point and they will have less slope.

The number of records in a block or the blocking factor also affects performance. A large blocking factor will decrease storage space but it increases transmission time. Small blocking factors decrease transmission time but increase arm movement time. A thorough analysis is again needed to determine optimum blocking.

CONCLUSION

In this paper, we have presented a prototype parametric study of the type that is almost mandatory for knowledgeable design of a complex file organization. This study, which includes thousands of data points, would not have been possible without a fast, accurate simulation model such as FOREM I. The results are presented to give the reader an indication of the intricate interdependence of the many parameters that he must consider if he wishes to produce an excellent file design.

REFERENCES

1 *Formatted file organization techniques*
Final Report Air Force Contract AF 30(602)-4088 May 1967

2 M E SENKO V Y LUM P J OWENS
A file organization evaluation model (FOREM)
IFIP Congress 1968

APPENDIX

This section presents comparisons of the model runs with actual computer runs to illustrate the accuracy attainable with the model's equation evaluation approach. In calibrating FOREM I, more than 100 experiments were set up, representing the various modes of operation of the IBM Indexed Sequential Access Method. The following tables record the 37 measurements which approximate the model set-ups.

Mode of Retrieval	Overflow[2] Handling	Percent* Overflow	Model Result (secs.)	Measured Result (secs.)	Model Error (percent)
File creation	ind	0	186.	159.	17.0
Sequential retrieval	cyl	0	10.9***	8.51	28.1
Sequential retrieval	cyl	5.	16.6	16.1	3.11
Sequential retrieval	cyl	16.7	30.0	27.9	7.52
Sequential retrieval	ind	0	10.9***	8.64	26.1
Sequential retrieval	ind	5.	45.5**	36.9	23.3
Sequential retrieval	ind	16.7	82.1**	69.2	18.6
Sorted key[1] retrieval	cyl	0	422.	414.	1.93
Sorted key retrieval	cyl	5.	419.	414.	1.21
Sorted key retrieval	cyl	16.7	448.	451.	9.67
Sorted key retrieval	ind	0	422.	412.	2.43
Sorted key retrieval	ind	5.	457.	464.	1.51
Sorted key retrieval	ind	16.7	613.**	544.	12.7
Random retrieval	cyl	0	790.	732.	7.92
Random retrieval	cyl	5.	787.	744.	5.77
Random retrieval	cyl	16.7	816.	773.	5.56
Random retrieval	ind	0	781.	715.	9.2
Random retrieval	ind	5.	802.	752.	6.64
Random retrieval	ind	16.7	922.	846.	8.98
Random Update	cyl	0	915.	970.	6.01

1 - The keys of the records to be retrieved are sorted in ascending order and retrieval carried out in the order of this reference.

2 - cyl means cylinder overflow (overflow records in same cylinder as prime records), and

ind means independent overflow (overflow records in different cylinders as prime records).

Appendix insert 1

Mode of Retrieval	Overflow Handling	Percent* Overflow	Model Result (secs.)	Measured Result (secs.)	Model Error (percent)
Random update	cyl	5.	912.	951.	4.10
Random update	cyl	16.7	941.	999.	5.80
Random update	ind	0	906.	955.	5.13
Random update	ind	5.	927.	941.	1.49
Random update	ind	16.7	1047.	1038.	.87
Random insertion	cyl	0	1422.	1351.	5.25
Random insertion	cyl	5.	1418.	1381.	2.67
Random insertion	cyl	16.7	1446.	1371.	8.16
Random insertion	ind	0	2458.	1937.	26.9
Random insertion	ind	5.	2493.	2005.	24.3
Random insertion	ind	16.7	2717.	2243.	21.1
Sorted key insertion	cyl	0	1054.	1077.	2.13
Sorted key insertion	cyl	5.	1050.	1018.	3.14
Sorted key insertion	cyl	16.7	1078.	963.	11.9
Sorted key insertion	ind	0	1979.	1957.	1.12
Sorted key insertion	ind	5.	2027.	1984.	2.17
Sorted key insertion	ind	16.7	2288.	2207.	3.67

*In order to have the model set-up and the real data set-up be as close as possible, the 0 percent overflow actually has a very small number of overflow records.

**The discrepancy between model and measured results is mainly due to the set-up of overflow records in the created data set. The overflow records belonging to the same track, for example, are stored very close to each other in the actual set-up. In the model, each pair of records is considered to be separated by half as many cylinders as there are overflow cylinders.

***The error in this case is due to the assumption that missing of revolutions occurs when control is returned to CPU to set up the reading of the track index and next data track. In the particular data set chosen, no missing occurs probably because there is a considerable amount of empty space at the end of each data track.

Appendix insert 2

The measured results were obtained using an IBM 360/mod 50 processor and IBM 2314 disk storage devices. The file has 100,000 records and each record is 200 bytes long including an 8-byte key. The records are blocked full track in the prime area and unblocked in the overflow area. 5,000 records were processed in each case. The model assumes that record processing time is negligible.

As the tables below indicate, the model gives fairly accurate results. The average error for all the experiments is 8.3 percent. In some experiments, the actual set-ups are not quite the same as the model (those with asterisks) and large discrepancies exist between the two runs. Removing these cases results in an average deviation of 6.7 percent.

Measurements of other access methods not shown in the tables also have about the same accuracy.

Fast "infinite-key" privacy transformation for resource-sharing systems*

by J. M. CARROLL and P. M. McLELLAND**

The University of Western Ontario
London, Ontario, Canada

INTRODUCTION

In all systems affording real-time multiple access to shared computing resources, there exists the risk that information belonging to one user, may, contrary to his intent, become available to other users, and there is the additional risk that outside agencies may infiltrate the system and obtain information. Protection of information within central processors, auxiliary storage (disk, drum), and on-site bulk storage (tape), is a responsibility of the system; the responsibility for the protection of information in external communication links seems presently to devolve by default upon the user. The crux of the privacy issue is the design, evaluation, and implementation of hardware, software, and operating procedures contrived to discharge both of these responsibilities.

This paper describes a real-time software system for privacy transformation applicable to the problems of both system and user. It will be presented within the context of known threats to privacy, presently available counter-measures, and the current operational environment.

THREATS TO PRIVACY

The challenges to the privacy of information in a computer system may be accidental or deliberate; this discussion relates specifically to deliberate challenges, although the software developed may afford some protection against the undesired consequences of accidental compromise.

* Support of the Defence Research Board (Canada) and the Canada Council, Social Sciences and Humanities Division is gratefully acknowledged.

** Now with the Univac Division, Sperry Rand of Canada, Ltd., Ottawa, Ontario, Canada.

The objectives of deliberate infiltration include:

1. Gaining access to information in files.
2. Discovering the information interests of users.
3. Altering or destroying files.
4. Obtaining free use of system resources.

The nature of deliberate infiltration will be discussed within the framework presented by Peterson and Turn,[1] who established the following categories:

A. Passive Infiltration

1. Electro-magnetic pickup (from CPU or peripheral devices).
2. Wiretapping (on communications lines or transfer buses).
3. Concealled transmitters (CPU, peripheral devices, transfer buses, communications lines).

B. Active Infiltration

1. Browsing.
2. Masquerading.
3. Exploitation of trap doors.
4. "Between-lines" entry.
5. "Piggy back" infiltration.
6. Subversive entry by centre staff.
7. Core dumping.
8. Theft of removable media.

Browsing is defined as the use of legitimate access to the system to obtain unauthorized information.
Masquerading consists of posing as a legitimate user after obtaining proper identification by subversive means.
Trap doors are hardware or software deficiencies that assist the infiltrator to obtain information having once gained access to the system.
Between-lines entry consists of penetrating the system

when a legitimate user is on a communications channel, but his terminal is inactive.

Piggy-back infiltration consists of selectively intercepting user-processor communications and returning false messages to the user.

EXISTING COUNTERMEASURES

Methods to enhance privacy are roughly classified as follows:

1. Access control.
2. Privacy transformations.
3. Processing restrictions.
4. Monitoring procedures.
5. Integrity management.

Access control consists of authorization, identification, and authentication and may function on the system or file level. Authorization to enter the system or files is generally established by possession of an account number or project number. The user may be identified by his name, terminal, or use of a password. The user may be required to perform a privacy transformation on the password to authenticate his identity. Peters[2] recommends use of one-time passwords.

Passwords may also include authority codes to define levels of processing access to files (e.g., read only, write, read-write, change protection).

Privacy transformations include the class of operation which can be used to encode and decode information to conceal content. Associated with a transformation is a *key* which identifies and unlocks the transformation to the user and a *work factor*, which is a measure of the effort required of an infiltrator to discover the key by cryptanalysis.

Processing restrictions include such functions as provisions to zero core before assigning it to a second user, mounting removable files on drives with disabled circuitry that must be authenticated before accessing, automatic cancellation of programmes attempting to access unauthorized information, and software which limits access privileges by terminal.

Monitoring procedures are concerned with making permanent records of attempted or actual penetrations of the system or files. Monitoring procedures usually will not prevent infiltration; their protection is *ex post facto*. They disclose that a compromise has taken place, and may help identify the perpetrator.

Integrity management attempts to ensure the competence, loyalty, and integrity of centre personnel. In some cases, it may entail bonding of some staff.

Threat	Directed Against	Countermeasure				
		Access Control	Privacy Transform.	Process. Restrict.	Threat Monitor.	Integrity Manage.
Passive	CPU	NONE	NONE	NONE	NONE	FAIR
	DEVICES	"	GOOD	"	"	"
	LINES	"	"	"	"	NONE
Browsing	SYSTEM	GOOD	GOOD	GOOD	GOOD	NONE
Masquerade	"	FAIR	FAIR	FAIR	GOOD	FAIR
Between-Lines	"	NONE	GOOD	FAIR	FAIR	NONE
Piggy-Back	"	NONE	FAIR	FAIR	FAIR	NONE
Trap-Doors	CPU	NONE	NONE	NONE	FAIR	NONE
	DEVICES	FAIR	GOOD	FAIR	"	"
Systems Entry	CPU	NONE	NONE	NONE	NONE	GOOD
	DEVICES	"	FAIR	"	FAIR	"
Core Dump	CPU	NONE	NONE	NONE	GOOD	GOOD
Theft	DEVICES	NONE	GOOD	NONE	FAIR	GOOD

Figure 1—Threat-countermeasure matrix

EFFECTIVENESS OF COUNTERMEASURES

The paradigm given in Figure 1, grossly abridged from Peterson and Turn, characterizes the effectiveness of each countermeasure against each threat.

We independently investigated each cell of the threat-countermeasure matrix in the real-time resource-sharing environment afforded by the PDP-10/50 at Western (30 teletypes, 3 remote batch terminals). Our experience leads to the following observations:

Passive Infiltration: There is no adequate countermeasure except encipherment and even this is effective only if enciphered traffic flows on the bus or line attacked by the infiltrator. Competent, loyal personnel may deter planting wireless transmitters or electromagnetic pickups within the computer centre.

Browsing: All countermeasures are effective; simple access control is usually adequate.

Masquerading: If the password is compromised, most existing countermeasures are rendered ineffective. Use of authentication, one-time passwords, frequent change of password, and loyalty of systems personnel help to preserve the integrity of passwords. Separate systems and file access procedures make infiltration more difficult, inasmuch as two or more passwords must be compromised before the infiltrator gains his objective. Monitoring procedures can provide *ex post facto* analysis.

Between-Lines Entry: Only encipherment of files, or passwords applied at the message level rather than for entire sessions, provide adequate safeguards. Monitoring may provide *ex post facto* analysis.

Piggy-Back Techniques: Encipherment provides protection unless the password is compromised. Monitoring may provide *ex post facto* analysis.
Trap-doors: There is no protection for information obtainable from core, although monitoring can help in *ex post facto* analysis. Encipherment can protect information contained in auxiliary storage.
Systems entry: Integrity management is the only effective countermeasure. There is no other protection for information in core; even monitoring routines can be overridden. Encipherment protects information in virtual storage only to the extent that passwords are protected from compromise.
Core dump: There is no effective protection except integrity management, although monitoring procedures can help in *ex post facto* analysis.
Theft: Encipherment protects information stored in removable media.

Our initial study persuaded us that privacy transformation coupled with password authentication would afford the best protection of information. Integrity management procedures were not within the scope of this research.

RESOURCE-SHARING SYSTEMS

Number of Systems	Authorization	Identification	Authority
4	Account # Project #	Name	Password
3	Account #	-	Password
3	Account # Project #	Name	-
2	Account #	-	-
1	Account #	Name	Password
1	Project #	Name	-
1	-	-	Password
1	Project #	-	-

Figure 2—Access control in 16 Canadian resource-sharing systems

PRIVACY ENVIRONMENT: MANUFACTURERS

Our next task was to investigate the privacy environment of resource-sharing systems. Five manufacturers of equipment, doing business in Canada, participated in our study. Their contributions are summarized in the following points:

1. The problem of information security is of great concern to all manufacturers of resource-sharing equipment.
2. Most manufacturers are conducting research on privacy; only a small minority believes that the hardware and software currently supplied are adequate to ensure the privacy of customer information.
3. The password is the most common vehicle of system access control; dedicated direct lines are recommended in some special situations.
4. At least two manufacturers have implemented password authentication at the file level.
5. There appears to be no customer demand for implementation of hardware or software privacy transformations at this time.
6. Most manufacturers stress the need for integrity management.
7. Two large manufacturers emphasize the need for thorough log-keeping and monitoring procedures.
8. Communication links are seen as a major security weakness.

We next surveyed 25 organizations possessing hardware that appeared to be suitable for resource-sharing. Sixteen organizations participated in our study, representing about 75 percent by traffic volume of the Canadian time-sharing industry. From information furnished by them, we were able to obtain a "privacy profile" of the industry.

The average resource-sharing installation utilizes IBM equipment (Univac is in second place). The typical system has over 512 thousand bytes of core storage and 175 million bytes of auxiliary storage. The system operates in both the remote-batch and interactive modes. It has 26 terminals communicating with the central processors over public (switched) telephone lines.

In seven systems, authorization is established by name, account number, and project number. Five systems require only an account number. Nine systems require a password for authority to enter the system; the password is protected by either masking or print-inhibit.

Identification is established by some combination of name, account number, project number, or password; in no case is identification of the terminal considered. No use is made of one-time passwords, authentication, or privacy transformations. In no system is a password required at the file level; seven systems do not even require passwords. Access control provisions of 16 Canadian systems are summarized in Figure 2.

Only two systems monitor unsuccessful attempts to gain entry. In nine systems, both centre staff and other users have the ability to read user's files at will. In six systems, centre staff has unrestricted access to user files. Only three organizations have implemented integrity management by bonding any members of staff.

The state of privacy, in general, in the Canadian resource-sharing industry, can be described as chaotic and, with few exceptions, the attitude of systems operators towards privacy as one of apathy.

PRIVACY TRANSFORMATION: FUNCTIONAL SPECIFICATIONS

It was decided, therefore, to begin development of a software system for privacy transformation that would be synchronized by an authenticated password, anticipating that sooner or later some users will demand a higher degree of security in resource-sharing systems than is currently available. Such an authentication-privacy transformation procedure would afford the following advantages:

1. Provide protection for the password on communications channels.
2. Implement access control at the file level.
3. Obviate the need for storing passwords as part of file headings.
4. Afford positive user identification since only authorized users would be able to synchronize the keys of the privacy transformation.
5. Furnish "work factor" protection of files against browsing, "between-lines" entry, "piggy-back" infiltration, "trap doors" to auxiliary storage, entry of systems personnel to auxiliary storage, eavesdropping on transfer buses, and theft of removable media.

The technique of privacy transformation that seemed most promising was a form of the Vernan cipher, discovered in 1914 by Gilbert S. Vernan, an AT&T engineer. He suggested punching a tape of key characters and electromagnetically adding its pulses to those of plain text characters, coded in binary form, to obtain the cipher text. The "exclusive-OR" addition is used because it is reversible.

The attractive feature of the Vernan cipher for use in digital systems is the fact that the key string can readily be generated by random number techniques. For maximum security (high work factor) it is desirable that the cipher key be as long as the plain text to be encrypted. However, if the flow of information is heavy, the production of keys may place extreme loads on the arithmetic units of processors—the rate of message processing may then be too slow to be feasible. Two solutions have been proposed.

In the first, relatively short (e.g., 1,000 entries) keys are produced and permutations of them used until repetition is unavoidable. A second approach is to use an extremely efficient random number generator capable of producing strings that appear to be "infinite" in length, compared to the average length of message to be transformed.

PRIOR WORK (SHORT-KEY METHOD)

An algorithm for a short key method presented by Skatrud[3] utilizes two key memories and an address memory. These are generated off-line by conventional random number techniques. Synchronization of an incoming message and the key string is achieved by using the first information item received to address an address memory location. The contents of this memory location provides a pair of address pointers that are used to select key words from each of the key memories. The key words are both "exclusive-OR'ed" with the next data item, effectively providing double encryption of it. The address memory is then successively incremented each time another data item arrives. Each address location provides two more key address pointers and each key address furnishes two key words to be "exclusive-OR'ed" with the current data item. Key word pairs are provided on a one-for-one basis with input data items until the entire message has been processed. For decoding, the procedure is completely reversible.

PRESENT WORK ("INFINITE" KEY METHOD)

We decided to use the infinite key approach because it would:

1. Reduce storage requirements over those required by short key methods. This will tend to reduce cost where charges are assessed on the amount of core used; and, more importantly, will permit implementing the transformation on small computers (e.g., one having a 4096-word memory) located within the user's work space.
2. Obviate the need for off-line key production and virtual storage of superseded keys [or

reencipherment of existing files after a key memory change].
3. Provide extremely long key strings for improved work-factor protection of information.

Our privacy transformation combines the speed of the arithmetic congruential method of random-number generation with the previously established randomness of a mixed multiplicative congruential generator. The length and composition of the seed for the key string are determined by the password of the file to be coded or decoded. Each unit of input text is "exclusive-OR'ed" with the next random number of the key in a one-to-one manner until the input plain text has been fully processed. The system has been implemented on a PDP-10/50 computer; the "unit of input text", in this case, is a 36-bit word.

There are two programmes; the input/output (CRYPTO.IO) which handles password authentication; and the privacy transformation (CRYPTO.2).

In the first programme, the user calls the name of the file to be processed. The user's directory, CRYPTO. UFD, which is maintained as a protected file on the system disk area, is searched to verify that a password exists. (A user must contact the system security officer to have a file password entered.) If no password exists, the user receives an error message. Otherwise, the 6-digit password is retrieved from disk to core, and the file name is stored in core for future reference.

The user is now challenged to authenticate his identity. A random 5-digit octal number is transmitted to user's terminal, and he is expected to effect on this number a pre-determined transformation, dependent on the file password, and transmit the result to the processor. If the transformation is incorrect, an error message will be transmitted and the user line dropped. (A date-time group may be made part of this transformation to afford additional protection against "piggy-back" infiltration.)

If the transformation is correct, the file is initialized for output, a call is generated for the transformation routine, and the file name and password are supplied. Upon return of control from the transformation subroutine, the run is terminated. Figure 3 is a logic flowchart for this procedure.

IMPLEMENTATION ON A LARGE COMPUTER

The privacy transformation makes use of a two-stage random number generator, IRAND, which supplies key strings dynamically in 512-word blocks. Each word is exclusively OR'ed with an input string of five 7-bit ASCII characters to realize encryption or decryption, as the case may be. Synchronization of the key strings with text is achieved by using the 6-digit password as the starting seed S_0 for the seed-string generator, which is one stage of the two-stage random number generator.

A key string of N random numbers is generated using the additive congruential method

$$X_{i+1} \equiv X_i + X_{i-L} \pmod{m}$$

where $m = 2^b$ and b is 35, the bit length of the computer word; and where

$$X_1, \ldots, X_L$$

is the sequence

$$S_1, \ldots, S_L$$

which is called the seed string.

The seed string is generated by using a conventional mixed multiplicative congruential generator of the form

$$S_{i+1} \equiv aS_i + C \pmod{m}$$

where

$$a \equiv \pm 3 \pmod{8}$$

and is a value close to $a = 2^{b/2}$, to satisfy the Conveyou-Greenburger criterion.[5,6] The value 131,069 has been found to be satisfactory.

The value of C was selected to be $< a$ and relatively prime to m. The value 7 was found to be satisfactory.

The length of the seed string is variable but selected to fall within the range $16 \leq L \leq 79$, in accordance with the suggestions set forth by Green, Smith, and Klem.[7] Its length is determined by adding 16 to the six lower order bits of the password.

The period of usage of the seed string (N) is determined by adding 2^K to the K low-order bits of S_L, and is within the range

$$2^K \leq N < 2^{K+1}.$$

For high-security applications, $K = 18$; for lower-security applications $K = 12$ to decrease the load on the arithmetic unit of the processor.

After N random numbers have been generated in this fashion, the procedure is repeated using the current value of S_L instead of the password (S_0). Thus the complete contents of a file may be enciphered with a vanishingly small probability that any portion of the key string will repeat.

IMPLEMENTATION ON A SMALL COMPUTER

Implementation of this privacy transformation on a small computer such as the PDP-8I (12-bit word,

4096-word memory), requires certain modifications in the procedure. Two additive congruential generators are used so that all operations may be kept within the limitations of the machine's basic arithmetic capability (two's complement add).

The seed of the second generator is stored as a string of 64 4-digit octal numbers. (We have found that digits of the expansion of pi work well; any random string can be used.)

The second generator randomly initializes itself by cycling for a number of periods determined by the 12 low-order bits of the password. It then generates a seed $S_1, \ldots, S_L$, for the first generator. The length of the seed is determined by the six low-order bits of the password. The length of key string (e.g., period of the first generator) is determined by the value of the four octal digits of S_L. (Note: all operations are carried out in octal arithmetic modulo 4096.) At the end of this period, S_0 is set equal to S_L and the process is repeated.

EVALUATION

Three tests were conducted on our privacy transformation programme as implemented on the PDP-10/50. The first two tested in cipher key for random characteristics, comparing it with a mixed multiplicative congruential generator described in IBM,[8] which we knew from experience in our Simulation Laboratory to have acceptably random characteristics. The tests evaluated goodness-of-fit to a uniform distribution, and serial autocorrelation lagged 1 to 5 (the second test thus has five distinct parts).

The third test was for speed of producing encoded words.

The randomness tests were conducted according to procedures attributed to I. J. Good[9,10] as described in Naylor[11] and Lewis.[12] Each part of each test used 100 random blocks of 1,000 numbers each, sampled from the "infinite" cipher key string. The 100 resulting values of chi² (one value for each block), calculated

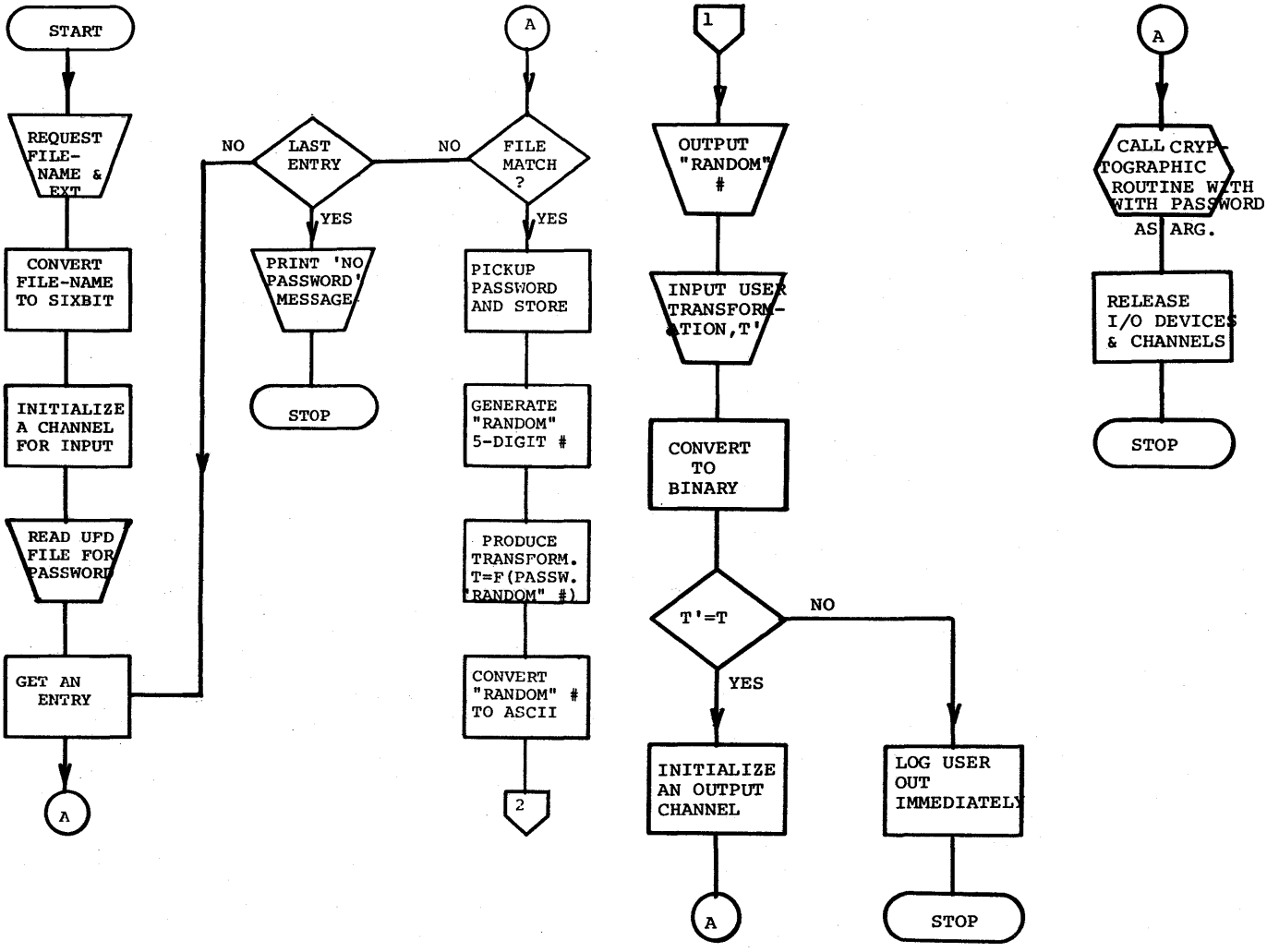


Figure 3—Privacy transformation routines

under the assumption of a flat distribution in the first case, and no serial autocorrelation in the second, were then tested for goodness-of-fit to a chi² distribution having 99 degrees of freedom. The criterion value, the 95 percent confidence level for chi² with 9 degrees of freedom, was 16.9. Values of chi² lower than this were deemed to signify that the outcome of a test was satisfactory.

The test results are tabulated in Figure 4. It can be seen that the high-security cipher (K-18) is superior to the low-security cipher (K-12) with respect to both goodness-of-fit to a uniform distribution, and absence of serial autocorrelation lagged 1 to 5. Both cipher key strings were found to be superior in these qualities of randomness to the conventional mixed multiplicative congruential generator. In addition, no evidence of cycling was observed in either cipher key string.

The test of processing speed was carried out in the time-sharing environment of the PDP-10/50 system under full load with "swap" times included. It was found that 135,168 five-byte words were produced in 18.28 seconds (37,000 bytes per second) using the high-security (slower) procedure. It is felt that this test represents worst case conditions and that double the observed speed can easily be realized.

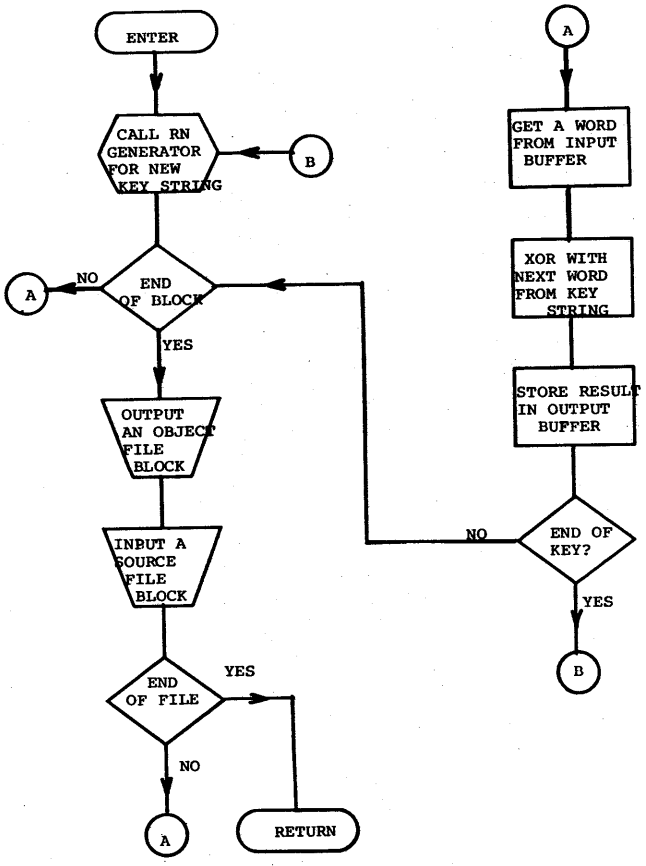


Figure 3—Privacy transformation routines

(VALUES OF chi²)

Method	Test					
	Goodness	Serial Autocorrelation				
	of Fit	Lag 1	2	3	4	5
High Security	5.4	8.2	12.8	15.4	8.8	7.6
Low Security	9.6	9.6	11.2	12.2	18.2	14.2
MMC Generator	12.8	12.8	16.6	16.4	26.4	11.2

(Acceptable chi² = 16.9)

Figure 4—Evaluation of the randomness of cipher key strings

SUMMARY

The two-stage random number generator used in this privacy transformation procedure appears to possess excellent characteristics of randomness and is capable of producing a cipher key string that is effectively infinite in length. These characteristics should ensure high "work factor" security against cryptanalytic attack. (Actually, it is a rather good random number generator for general simulation work.)

The speed of the encipherment routine is sufficient to keep up with normal data transfers between the processor and peripheral devices.

Use of an authenticated password to synchronize the cipher key string affords several advantages:

1. Differential access to files is achieved by assigning a unique password to each file and investing only authorized users with it.
2. Differential access at lower levels (e.g., records) requires only using more of these 6-digit passwords.
3. Storage of the passwords in a separate user file directory conserves space in user files and helps preserve the integrity of passwords.
4. The authenticating transformation effected on the password ensures that it never appears in clear on communications lines.
5. Possession of the password and authenticating transformation positively establishes the bona fide authority of the user.

Existence of high- and low-security modes for the privacy transformation allows the user to trade-off

"work factor" protection against load on the processor's arithmetic unit at will.

When the privacy transformation is modified to utilize two arithmetic congruential generators, the user is able to implement it on a computer having minimal word length, storage, and computational capability. He can, therefore, encipher his traffic on external switched lines at a hardware cost of roughly $10,000.

It is felt that the authentication—privacy transformation system described provides a flexible, efficient, and economical method for ensuring the privacy of information in resource-sharing computer environments.

[An additional benefit arising from our research was the discovery of two "trap doors" in the PDP-10/50 monitor, which were nailed shut by relatively simple programming fixes. This experience suggests to us that the security officer for a system handling sensitive information might find it worthwhile to assign at least one competent systems programmer as a "counter-infiltration" specialist with the assigned mission of discovering and repairing similar deficiencies.]

ACKNOWLEDGMENTS

This work was supported in part by the Canada Council, Social Sciences and Humanities Division, under grant number 69-0671 (Privacy and the Computer) and by the Defence Research Board (Canada) under grant number 9931-23 (Protection Methods for Real-Time Computer Systems).

The additive congruential generator for a short word length machine was programmed and tested by G. M. Dawdy of the Computer Science Department.

REFERENCES

1 H E PETERSEN R TURN
System implications of information privacy
AFIPS Conference Proceedings Spring Joint Computer Conference Vol 30 pp 291-300 1967

2 B PETERS
Security considerations in a multi-programmed computer system
AFIPS Conference Proceedings Spring Joint Computer Conference Vol 30 1967

3 R O SKATRUD
The application of cryptographic techniques to data processing
AFIPS Conference Proceedings Spring Joint Computer Conference Vol 35 pp 111-118 1969

4 T E HULL A R DOBELL
Mixed congruential random number generators for binary machines
J ACM Vol 11 No 1 pp 230-254 1964

5 R R COVEYOU
Serial correlation in the generation of pseudo-random numbers
J ACM Vol 7 pp 72-74 1960

6 M GREENBURGER
An a priori determination of serial correlation in computer generated random numbers
Math of Computation Vol 15 pp 383-389 1961

7 B F GREEN J SMITH L KLEM
Empirical tests of an additive random number generator
J ACM Vol 6 No 4 pp 527-537 1959

8 *Random number generation and testing*
IBM corp Form C20-8011 White Plains N Y 1959

9 I J GOOD
The serial test for sampling numbers and order tests of randomness
Proc Camb Phil Soc Vol 49 pp 276-284 1953

10 I J GOOD
On the serial test for random sequences
Annals of Math Stat Vol 28 pp 262-264 1957

11 T H NAYLOR J L BALINTFY D S BURDICK K CHU
Computer simulation techniques
John Wiley & Sons Inc New York pp 52-53 1966

12 P A W LEWIS A S GOODMAN J M MILLER
A pseudo-random number generator for the systems 360
IBM Syst J Vol 8 No 2 p 136 1969

On-line computer managed instruction— The first step*

by JACQUELINE S. VIERLING

Honeywell Educational Resource Center
Minneapolis, Minnesota

and

M. SHIVARAM

Quality Educational Development, Inc.
Washington, D.C.

INTRODUCTION

Computer Managed Instruction (CMI) is a term employed to designate a system which ". . .uses the computer to help the teacher administer and guide the instructional process."[1] The major features of CMI are diagnosis and testing, analysis, record keeping, and prescription. Diagnosis and testing is used to evaluate each student's performance at regular intervals. The results are then analyzed to update the student's record and to provide a prescription which suggests the learning process the student should pursue to achieve his objectives. A computer-generated prescription would assign the student to one of several teaching-learning units consisting of textbooks, films, slides, or any of a multitude of technological media.

A typical teacher today, who might be responsible for 50 or more students in at least two different courses, with 30 teaching-learning units per course, has no fewer than 3000 media assignments to make. The primary purpose of the computer in CMI is to help teachers manage these assignments in a way which increases the efficiency and the quality of instruction afforded to each student. In order to provide this guide or prescription for the student, the computer must have access to files containing each student's objectives, all records which characterize the student's abilities and performance, and the educational resources and materials which will be available to the student. Administrative uses of the computer (diagnosis, testing and record keeping) are currently available to many students throughout the country. The analysis of these records by a computer to provide an individualized learning prescription, however, is a relatively new concept. Algorithms must be developed which correlate a student's performance with a media prescription based on individual preferences and capabilities. In addition, the prescribed medium or media mix should be the one which will maximize the probability that a student will achieve the stated objectives.

The CMI system described here was used during a one semester multi-media physics course at the U.S. Naval Academy. The course was designed to evaluate seven media-mixes in order to elucidate an algorithm which correlates diagnosis, testing, and analysis with individualized prescriptions. The determination of such an algorithm is essential to the development of CMI. The first step in CMI development, therefore, is provided by the system described here which was used to perform diagnosis, testing, record keeping, and analysis. The second step is to determine the required correlation by a further analysis of the collected data. The third step would be the validation of the algorithm, and the fourth step would be the implementation of the complete CMI system. The system was used on a remote terminal connected via phone lines to a time-shared system. Time-sharing has many advantages: the remote terminal is easily accessible, the system services each user in a matter of seconds or minutes, and simple conversational languages are available. An added bonus is the fact that

* The work described in this article was carried out by Quality Educational Development, Inc., Washington, D.C. in conjunction with the New York Institute of Technology and the United States Naval Academy under the United States Office of Education Contract No. 8-0446.

TABLE I—CMI Programs

Program	Description	Input files	Output files	TTY Output
XN1	Scoring program for weekly pre-tests and post-tests.	H5 I5	X1-X12	Student scores, questions missed
XN2	Performs analysis of student scores by media group; determines percent of students missing each question.	I3 H4 X1-X12	None	Analysis by media and by question
TS2	Performs T-Score analysis of post-test scores	I3 X1-X12	TS4	T-Score distribution
XN9	Creates a file which lists each student's raw score and the corresponding T-Score.	TS4 X1-X12 J	M2	None
XN10	Updates the file used to calculate student's cumulative average T-Score; creates the file necessary to determine student's percentile standing in class.	K1 M2	K2 T3	None
XN8	Creates the output which gives for each student his raw score and T-Score for the current week, his cumulative average T-Score to date, and his percentile standing in the class.	H5 J K1 T3 M1	None	Continuous output for all students.

schools which cannot afford a data center can often afford the costs of a remote terminal and time-sharing.

Two design criteria were considered paramount to the structure of this CMI package: the system must be flexible enough to allow additions or changes with a minimum of effort, and the system should allow execution and maintenance by educators without the requirement for a staff of programmers. With these goals in mind, the programs were written in BASIC** by teachers who used time-sharing via a teletype terminal. The system was based upon student files which could be accessed by a series of application programs. Since the programs were essentially independent, the output could easily be changed by simply writing another program. The series of programs used is listed in Table I and the data files are described in Table II; they will be discussed in detail below. The data files were protected with a "password" to prevent unauthorized persons from reading or changing these files. This protection is necessary since students will often use the same time-sharing computer for problem-solving and tutorial sessions.

** Beginners Algorithmic Symbolic Instruction Code, developed by Dr. Kemeny and Dr. Kurtz at Dartmouth College.

COURSE STRUCTURE AND SYSTEM DESIGN

The physics course, which had been completely defined in terms of behavioral objectives[2] was given to twelve sections comprising about two hundred students. Independent of their previously assigned sections, the students were randomly assigned to ten "groups". One group (CAI-I) utilized the computer in a problem solving mode, a second group (CAI-II) was exposed to a series of fifty tutorial lessons, and a third group (CNTR) served as controls.[3] The other seven groups were assigned one of seven different media-mixes each week; during the semester, each of these groups tried each media-mix twice. The seven media mixes are lecture (L), lecture/study guide (L/SG), study guide (SG), illustrated book (IB), talking book (TB), audio visuals (AV), and student option (SO). Details of course design are well documented elsewhere[4,5] and only those items necessary for understanding of the CMI programs will be noted here.

The students were given a weekly pre-test to determine their knowledge of material to be presented and a post-test which measured their comprehension of the previous week's instructional sequence. These

TABLE II—DATA FILES

Name	Description	Created by	Accessed by
M4	Master file of student names.	User	—
J	Master file of student names with absentees indicated for the current week.	User	XN8 XN9
I3	Number of students in each section who took the exam.	User	TS2 XN2
H5	Title of work unit, week letter.	User	XN1, XN8
I6	Contains student input data: load one section at a time.	User	XN1
X1-X12	Student data files—contain scores, and results for each question, for every student.	XN1	TS2, XN9 XN2
TS4	Contains information on T-Scores: frequency, cumulative frequency, raw score, and T-Score.	TS2	XN9
M1, M2	Contains for each student—I.S. number, weeks raw score, corresponding T-Score.	XN9	XN10 XN8
K1, K2	Contains for each student—I.S. number, sum of T-Scores, number of exams; used to determine student's cumulative average T-Score.	XN10	XN8
T3	Contains highest T-Score for current week, T-Score and frequency distribution; used to determine student's percentile standing.	XN10	XN8

tests consisted of ten questions for which the students formulated their answers. In addition, they indicated their confidence that the answer was correct using a scale of 0–100 percent. Confidence testing was employed since it yields a more precise measure of a student's knowledge.[6] It is also designed to discourage guessing which leads to a more reliable item analysis.[7] The tests were scored using a logarithmic confidence algorithm,[5] which assigns credit for an answer based both on its correctness and the percent confidence indicated. The logarithmic confidence algorithm is shown as it appears in the scoring program (Figure 1). For example a student who answers a question incorrectly and indicates 100 percent confidence in the answer receives no credit (Table III: look for 100 per-, cent confidence under "Percent Confidence" column, move across line to column labeled "Answer Incorrect", read 0 percent of credit to be allotted). In contrast a student who answers a question incorrectly but indicates 0 percent confidence (read credit to be allotted under "Incorrect Answer" column on same line as 0 percent confidence) receives 40 percent of the allotted credit. For a ten question test, each ques-

TABLE III—Logarithmic Confidence Testing

	Assignment of Credit	
Percent Confidence	Answer Correct	Answer Incorrect
100	100	0
90	99	20
80	98	26
70	97	30
60	96	32
50	94	34
40	92	36
30	90	37
20	86	38
10	80	39
0	60	40

Percent Confidence The student expresses the confidence he has that his answer is correct.

Assignment of Credit Credit is assigned based on both the correctness of the question and the percent confidence. For example, if a question is correct and the confidence expressed was 90 percent, 99 percent of the allowed credit will be given. On a ten question exam, this is 99 percent of 10 points or 9.9 points.

```
1540 FOR I=1 TO N
1550 READ #2,A(I)
1560 IF A(I)=10 THEN 1700
1570 IF A(I)=111 THEN 1700
1580 IF A(I)=1 THEN 1610
1582 IF A(I)=0 THEN 1610
1584 LET C(I)=0
1586 GO TO 1642
1610 READ #2,C(I)
1620 IF C(I)>100 THEN 1720
1630 IF A(I)=1 THEN 1670
1640 IF C(I)>=99 THEN 1650
1642 LET E= .01*C(I)
1645 LET G=G+2*(2+(LOG(1-E)/2.303))
1650 LET O(I)=1
1660 GO TO 1695
1670 IF C(I)=100 THEN 1675
1671 IF C(I)<=1 THEN 1678
1672 LET E=.01*C(I)
1673 LET G=G+2*(5+(LOG(E)/2.303))
1674 GO TO 1680
1675 LET G=G+10
1676 GO TO 1680
1678 LET G=G+6
1680 LET G1=G1+100/N
1690 LET O(I)=2
1695 NEXT I
1700 RETURN
1705 PRINT"DATA PUNCHED INCORRECTLY"
1710 STOP
1720 PRINT"CONFIDENCE GREATER THAN 100"
1730 STOP
```

Figure 1—The logarithmic confidence algorithm is shown as it appears in Program XN1

tion is worth 10 points, and 40 percent of the credit or 4 points would be given for this question. Blank answers were scored using 0 percent confidence. This type of complicated and time-consuming scoring is easily implemented with the aid of the computer.

Thus each week, students were given a ten question pre-test and post-test. Their answers and percent confidence were recorded on a separate answer sheet. A typist was used to punch this information on paper tape for input to the computer. The first week, the student's formulated answers were input to the computer for grading. This proved tedious and time consuming in terms of the variety of acceptable answers and the time necessary to punch the information on paper tape. In subsequent weeks, the wrong answers were marked and the typists used a code to indicate whether the answer was blank, correct, or wrong, and the percent confidence. In addition, the students were given a copy of the answer sheets so that they could check the results. During the course of the semester (200 students and 14 tests), only two corrections attributable to input errors were entered. It is important to realize that this mode of input was made necessary because of course design constraints and the hardware facilities available. There are more desirable possibilities and these will be noted in a later section.

SYSTEM OPERATION

The student's responses after being punched on paper tape, were input via the teletype to the computer and stored in the form of separate data file for each section. The scoring program (XN1 in Table I) read the input data in each of the section files and produced output to the teletype. This output (Figure 2) listed the confidence score, the conventional score, and questions missed for each student. The output was posted, and students would then schedule extra instruction with their section professors to discuss the behavioral objectives related to the questions they missed. At the same time, the student responses were scored, program XN1 created another set of

```
                    ON-SITE COMPUTER MANAGED DIRECTIVES

          VOLUME 1          TRANSIENTS  (INDUCTANCE)

SECTION  1005               POST-TEST RESULTS

                    SCORE:-
NAME                CONF    CONV        QUESTIONS MISSED
ADAMS,J.            62.3    60          3  5  6  7
BIENHOFF,P.         89.7    90          5
DAVIDSON,F.         70.9    60          2  8  9  10
DILGREN,G.          93.7    90          10
JOHNSON,M.          92.2    90          7
KENNEDY,M.          63.4    50          1  2  3  4  5
MEYER,D.            80.5    70          6  7  8
MINNIS,R.           36.8    10          1  2  3  5  6  7  8  9  10
SHACKLETT,A.        41.7    10          2  3  4  5  6  7  8  9  10
SMITH,J.A.          57.3    40          2  3  6  7  8  10
SMITH,K.R.          99.2    100
SPEIGHTS,M.         47.4    20          2  3  4  5  7  8  9  10
STOCKS,A.           36.3    30          3  10
WECHSELBERGER       59.2    50          3  4  6  7  9
WHITE,J.            59.5    70          1  4  8
WHITE,K.L.          52.2    50          1  2  3  6  7

THE MEAN SCORE FOR THIS SECTION IS   1103.42  /   16  =     68.9637

                    SECTION  1005
THE DISTRIBUTION OF SCORES FOR POST-TEST 1 IS-

    0-60        61-70        71-80        81-90        91-100
     6            3            1            3            3
```

Figure 2—The output for section 1005 is shown to illustrate the output produced by Program XN1 when the weekly post-tests are scored

```
80 REM  THIS IS THE X5 FILE
100  720056,  3,  43.4,  1,2,1,1,2,2,1,1,1,1
101  720133,  5,  57.8,  2,2,1,1,1,2,2,2,1,1
102  721799,  1,  47.3,  2,2,1,1,1,1,1,1,1,1
103  722835,  5,  74.5,  2,2,1,2,2,2,1,2,1,2
104  723934,  5,  28 ,  1,1,1,1,1,1,1,1,1,1
105  724165,  5,  39.2,  1,1,1,1,1,1,1,2,1,2
106  724872,  5,  66.3,  1,2,1,2,2,2,1,1,2,2
107  725390,  5,  64.8,  2,1,1,2,2,2,1,2,1,2
108  725425,  5,  52.9,  2,1,1,1,2,2,1,1,1,1
109  725796,  1,  91.9,  2,2,1,2,2,2,2,2,2,2
110  726699,  5,  54.8,  1,2,2,1,1,1,2,1,1,2
111  726811,  1,  63.8,  1,1,2,2,2,1,1,2,1,2
112  727539,  5,  42. ,  1,1,1,1,2,1,1,1,2,1
113  727945,  1,  71.4,  2,1,2,2,2,2,1,2,1,1
114  728561,  1,  59. ,  2,1,1,2,2,2,1,1,1,2
115  728659,  5,  53.6,  2,1,1,2,2,2,1,1,1,1
116  728925,  5,  49.4,  2,1,1,2,2,1,1,1,1,1
117  738927,  5,  39.8,  1,1,1,1,1,1,1,1,2,1
```

Figure 3—One of the section data files created by Program XN1 is shown. These files contain each student's I.D. number, confidence score, and his response to each question on the test

data files (Table II, data files X1–X12) for each section (Figure 3) which contained the student's identification number, group number, confidence score, and his response to each question in coded form (Figure 4). The identification or I.D. number is used to uniquely identify each student's responses. This number is stored along with the data and checked whenever programs manipulate these numbers to ensure that the responses of one student are *not* attributed to another student. These data files could then serve as input for any variety of analysis programs.

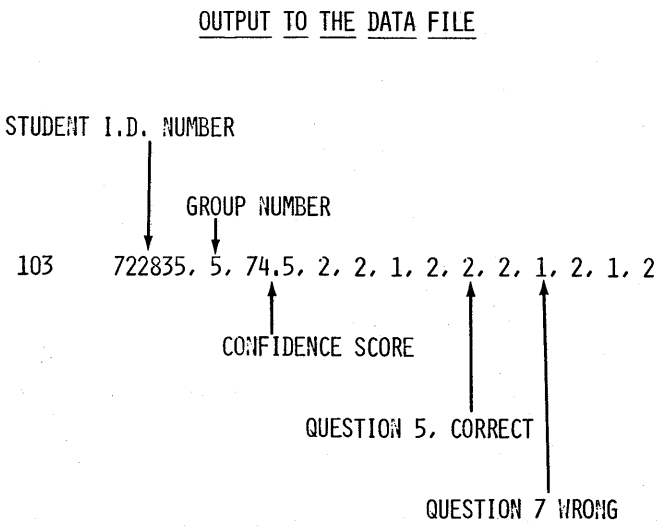


Figure 4—A line from one of the data files (X1-X12) is annotated to indicate the items stored

QUALITY EDUCATIONAL DEVELOPMENT

COMPUTER ANALYSIS OF STUDENT PERFORMANCE

POST-TEST K CURRENT AND RESISTANCE

ANALYSIS OF SCORES BY MEDIA

GROUP	0-60	61-70	71-80	81-90	91-100	MEAN
SG	0	0	5	2	13	91.1
IB	0	1	4	6	12	89.7
AV	2	0	3	6	8	87.
SO	0	2	5	5	11	87.4
TB	1	2	3	5	12	87.3
L	1	1	4	8	7	87.4
L/SG	0	2	4	4	11	88.8
CNTR	1	1	2	7	17	90.2
CAI-I	2	1	3	2	6	82.2
CAI-II	1	0	1	7	6	88.8

Figure 5—The results of the analysis of scores by media group is shown for Post-test K. This output is produced by Program XN2

Two of these programs will be described here. The first program (Table I, Program XN2) accessed the data files to produce an analysis of performance by media group (Figure 5), and an analysis of performance by question (Figure 6). It should be recalled that the tests were scored by section since the section professors were responsible for student grades but the media groups were randomly assigned independent of the sections. It was thus necessary to perform analysis both by section and by group. The data in the section files (files X1–X12), therefore, had to be analyzed or sorted to determine the effectiveness of the different media-mixes. The "analysis by group" sorted the student confidence scores by media-group. On a weekly basis, the difference in performance by group was generally small. Conclusions on the effectiveness of the media-mixes must await an analysis of variance of group performance during the semester. In addition, the response of each student to each question was sorted by computer to determine the percent of the students in each media group who missed each test question. This "analysis by question" or item analysis often showed distinct differences in performance by the various media groups. As an example, consider the results of the twenty-question test given at mid-semester (Figure 7). Question one which tested an

QUALITY EDUCATIONAL DEVELOPMENT

COMPUTER ANALYSIS OF STUDENT PERFORMANCE

POST-TEST K CURRENT AND RESISTANCE

ANALYSIS: PERCENT OF STUDENTS WHO MISSED A QUESTION

QUES.	1	2	3	4	5	6	7	8	9	10
SG	15.	0	0	0	10.	25	15.	0	20.	15.
IB	4.35	0	21.7	4.35	17.4	21.7	8.7	0	21.7	13.
AV	10.5	10.5	15.8	10.5	10.5	21.1	10.5	0	36.8	10.5
SO	0	0	4.35	17.4	26.1	39.1	0	0	43.5	17.4
TB	4.35	4.35	4.35	4.35	17.4	47.8	8.7	4.35	30.4	17.4
L	0	0	4.76	14.3	19.	42.9	4.76	0	33.3	14.3
L/SG	4.76	0	14.3	9.52	0	57.1	0	0	33.3	9.52
CNTR	25	3.57	14.3	3.57	14.3	7.14	7.14	3.57	0	35.7
CAI-I	21.4	28.6	21.4	28.6	7.14	21.4	21.4	0	0	35.7
CAI-II	13.3	0	13.3	6.67	20.	26.7	13.3	6.67	0	26.7
TOTAL	11.2	4.49	12.9	10.7	16.9	36.	9.55	1.69	26.4	22.5

Figure 6—Program XN2 also analyzes student responses to determine the percent of students missing each question as a function of media group. A sample output is shown

objective on the manipulation of vectors was correctly answered by more than 95 percent of the students taking the exam. In contrast, question 19 was answered incorrectly by 28 percent of the students. If the breakdown by group is scanned for this question, the individual group percentages vary from 14 to 48 with a low of 3.7 percent for the CAI group. It should be noted that all of the course objectives were not treated by all of the media.[8] The behavioral objective tested by question 19 (electric field/superposition) was treated by the lecture, study guide, and CAI material of week G, and by a review study guide made available to all groups during the review week. If we can assume that all media were of equal content, it would appear that CAI best enables the achievement of this objective in general. Further analysis could well refine the conclusion in terms of individual student characteristics.

The second program which will be discussed (Table I, Program XN8) provides the cumulative output of the weekly bookkeeping. This program illustrates the scope of data manipulation possible using only simple BASIC statements and a file oriented time-shared system. Several other programs (see Tables I and II) must be executed to T-Score the exams and to produce the data files read by this program. The order of execution is XN9, XN10, and XN8. The output for each section (Figure 8) lists each student's name, his current weeks raw confidence score, the equivalent T-Score, the cumulative average of all T-Scores to date, and his percentile standing in the class. This program reads data from four different data files, three of which were written by computer programs. The M4 file (Figure 9) contained the master list of students in the course. This file contained the list of student names, identification numbers and group numbers and it was grouped by section. After the post-test and prior to executing Program XN8, the M4 file was copied into the J file and absentees were indicated in the J file. In this manner, the master list (M4 file) remained unaltered. This file is created at the beginning of the semester and is altered only when a student

QUALITY EDUCATIONAL DEVELOPMENT

COMPUTER ANALYSIS OF STUDENT PERFORMANCE

POST-TEST H REVIEW EXAM

ANALYSIS: PERCENT OF STUDENTS WHO MISSED A QUESTION

QUES.	A	B	C	D	E	F	G	CAI	TOTAL
1	0	0	4.76	0	4.76	0	0	0	1.13
2	20.	21.7	19.	27.3	28.6	28.6	45.5	29.6	27.7
3	40.	39.1	47.6	63.6	57.1	28.6	54.5	63.	49.7
4	0	4.35	0	4.55	0	0	0	0	1.13
5	40.	21.7	33.3	63.6	28.6	28.6	27.3	44.4	36.2
6	70.	43.5	76.2	63.6	61.9	57.1	59.1	77.8	63.8
7	5.	13.	4.76	4.55	0	19.	9.09	3.7	7.34
8	10.	13.	14.3	13.6	4.76	19.	4.55	11.1	11.3
9	10.	17.4	28.6	27.3	19.	19.	36.4	25.9	23.2
10	5.	8.7	4.76	4.55	4.76	4.76	18.2	11.1	7.91
11	15.	8.7	19.	0	9.52	9.52	13.6	11.1	10.7
12	75	69.6	71.4	72.7	66.7	66.7	77.3	81.5	72.9
13	20	21.7	33.3	4.55	47.6	19.	36.4	18.5	24.9
14	25	26.1	19.	40.9	28.6	23.8	36.4	40.7	30.5
15	75	82.6	95.2	95.5	85.7	81.	81.8	63.	81.9
16	20	8.7	23.8	13.6	19.	19.	22.7	22.2	18.6
17	10.	8.7	19.	22.7	4.76	9.52	18.2	7.41	12.4
18	50	30.4	71.4	68.2	52.4	57.1	54.5	74.1	57.6
19	25	43.5	14.3	36.4	47.6	19.	36.4	3.7	27.7
20	75	69.6	95.2	59.1	85.7	90.5	86.4	74.1	79.1

Figure 7—The analysis of the percent of students missing each question is given for the mid-semester review exam to illustrate the feedback this output provides

ON-SITE COMPUTER MANAGED DIRECTIVES

CURRENT GRADES AND CUMULATIVE AVERAGE T-SCORES

SECTION 803

	POST-TEST M	MAGNETIC FIELD I		
NAME	RAW SCORE (M)	T-SCORE (M)	CAS (A-M)	PSM (A-M)
BRILLA,R.	68.5	53	55	86
CASKEY,H.	63.8	49	43	8
DRAWNECK,R.	83.6	63	51	63
HINSON,L.	67.8	52	56	91
HOPPER,W.	71.7	55	48	34
HORNE,B.	88.1	67	50	53
JOHNSON,G.	75.6	58	56	91
KENNEDY,T.	89	68	52	69
KRATOCHVIL	79.8	60	49	44
MARTIN,A.	64.1	49	51	63
MC DEVITT,R.	63.9	49	45	14
OSBORN,D.	63	48	50	53
SCHULER,T.	ABSENT	ABSENT	48	34
SHEARER,G.	67.7	52	48	34
SMITH,D.	82.6	61	56	91
SZOKA,M.	91.2	72	50	53
TOMLIN,E.	79	59	59	96
VAUGHN,D.	61.9	48	50	53
WOOD,C.	46	38	41	5
WILKENSON,J.	40.5	34	39	2

Figure 8—The output from Program XN8 for one section for week (M) is shown

drops the course. All programs which access data files check the I.D. numbers to ensure that there is no possibility of using the wrong data for a student's record.

The other files accessed by XN8 were the K1, T3, and M1 files (Table II); the K1 and M1 files were grouped by section. The K1 file (Figure 10) contains each student's I.D. number, the sum of his T-Scores, and the number of exams involved. This file can be used to calculate the cumulative average T-Score since that is the sum of the T-Scores divided by the number of exams. The K1 file is updated weekly by computer (Table 1, Program XN10) in the following manner: the program reads the cumulative information from the K1 file, reads the current T-Score from another file (file M1), checks I.D. numbers to ensure a match, then adds the current T-Score to the sum, increments the number of exams by one, and outputs this information to the K2 file. Thus after executing XN10, the K1 file contains the previous weeks cumulative data and the K2 file contains the updated cumulative data. The K1 file is then deleted (a paper tape of the file can be saved) and the K2 file is renamed to K1 before executing XN8. This minimizes the amount of data stored in the computer.

The XN8 program also reads data from the M1 file (Figure 11) which contains student I.D. numbers, the raw confidence score, and the equivalent T-Score for the current week. The fourth file read by program XN8 is the T3 file (Table II) which contains the data necessary to determine a student's percentile standing in the class.

A listing of program XN8 (Figure 12) is given to show the simple BASIC statements which are involved. For example lines 400–480 check that the I.D. numbers for all data refer to the same student. Should a mismatch occur, the program is written to print (lines 420–430) the name of the student (A$) from the master file, his section (T), and the files involved. The program reads the student data one student at a time, and prints the results to avoid excessive use of subscripted variables since this would affect the permitted

```
100 801,1,1
102 "BENHAM,W.",720490,5
104 "BLAKEY,B.",720651,5
106 "BRUCKER,B.",721029,1
108 "CROOK,K.",721694,1
110 "DIX,S.",722079,1
112 "ENGLUND,R.",722380,1
114 "GALLI,W.",722702,1
116 "GILLOOLY,J.",722863,5
118 "HEDRICK,M.",723584,5
120 "KEITHLY,T.",724410,5
122 "KING,M.",724571,1
124 "MOFFATT,W.",725978,5
126 "NEUPAVER,A.",726356,1
128 "NIELSEN,J.",726412,1
130 "PRINCE,T.",727014,1
132 "ROUND,W.",724399,5
134 "SCHUBERT,J.",727700,1
136 "SHOEMAKER,J.",727896,5
138 "SNYDER,W.",728155,5
140 "SPRINGMAN,R.",728232,5
142 "VANMAELE,J.",728932,5
144 "VANORSDEL,R.",728939,1
150 0,0,0
```

Figure 9—The Master File of student names (M4 file) is listed for one section to show how the names, I.D. numbers and group numbers are stored

```
100   1 ,
101   720490 , 855 , 14 ,
102   720651 , 616 , 12 ,
103   721029 , 641 , 13 ,
104   721694 , 661 , 13 ,
105   722079 , 482 , 11 ,
106   722380 , 612 , 14 ,
107   722702 , 487 , 10 ,
108   722863 , 704 , 14 ,
109   723584 , 541 , 13 ,
110   724410 , 617 , 12 ,
111   724571 , 676 , 14 ,
112   725978 , 663 , 14 ,
113   726356 , 804 , 13 ,
114   726412 , 742 , 14 ,
115   727014 , 716 , 13 ,
116   724399 , 618 , 14 ,
117   727700 , 635 , 14 ,
118   727896 , 664 , 13 ,
119   728155 , 604 , 12 ,
120   728232 , 602 , 13 ,
121   728932 , 614 , 13 ,
122   728939 , 673 , 13 ,
```

Figure 10—The data stored in the K1 file are listed for one section. This file contains the cumulative sum of each student's T-Scores and the number of exams involved

length of the BASIC program. The initial information on the heading for each section is read and printed (lines 210–360), while the data for each student in the section is read and printed in a loop (lines 390–600). Program XN8 will always run error-free since three input files (K1, M1, T3) are created by computer and the fourth file (M4) is checked during the execution of Program XN9.

CONCLUSIONS

The CMI programs described were successful from two points of view: the dead time between the student's responses and the analysis of results was kept to a minimum and the flexibility of the system was maintained. Both of these advantages accrue from the fact that the system was operated on-line in a time-sharing mode. There is no doubt that the use of a batch-processing system increases the dead time between collection of data and output since turn-around time for even efficient data centers can range from 3–24 hours. In contrast, time-sharing provides almost immediate responses with a limiting factor being speed of input and output. The modular design of this system, with an emphasis on easy access to data and results, allowed a flexibility which is often lacking in large CMI programs where the sheer size and complexity of the program often precludes changes. In turn, this flexibility ensures user satisfaction since the system is responsive to individual requirements. Furthermore, the use of an easy conversational language (BASIC) allows direct access to the system by teachers, educators, and students.

As noted above, a major area of consideration for on-line CMI is which remote terminal to use. In this experiment, a relatively large amount of time was required for input and output because the rate of transmission of the teletype is 10 characters/second. This time can be cut considerably simply by using one of the faster remote terminals available which transmit at rates of 30 characters/second and are compatible with most time-shared systems. Furthermore, if the course design utilizes multiple-choice exam questions, students can mark their answers on a card and a mark-sense card reader can be used. This eliminates

```
100   1 ,
101   720490 , 70.2 , 54 ,
102   720651 , 73.3 , 56 ,
103   721029 , 56.8 , 45 ,
104   721694 , 84.2 , 63 ,
105   722079 , 65.2 , 50 ,
106   722380 , 26.8 , 22 ,
107   722702 , 0 , 0 ,
108   722863 , 53.6 , 43 ,
109   723584 , 52.1 , 41 ,
110   724410 , 84.6 , 65 ,
111   724571 , 55.2 , 44 ,
112   725978 , 43.3 , 36 ,
113   726356 , 85.4 , 65 ,
114   726412 , 75.3 , 57 ,
115   727014 , 69.5 , 54 ,
116   724399 , 0 , 0 ,
117   727700 , 67 , 51 ,
118   727896 , 57.1 , 45 ,
119   728155 , 76.3 , 58 ,
120   728232 , 69.2 , 53 ,
121   728932 , 31.4 , 28 ,
122   728939 , 49.8 , 40 ,
```

Figure 11—The M1 file, which contains a student's raw confidence score and the corresponding T-Score for the current exam, is listed for one section

```
100 REM * * PROGRAM XN8 * *
200 DIM C(100)
210 FILES T3,H5,J,M1,K1
220 READ #1,X1,N,C(N)
230 IF,N=X1 THEN 245
240 GO TO 220
245 READ #2,V$,L$,K$
250 READ #3,T,X,P
255 READ #4,Y
256 READ #5,Z
260 PRINT"              ON-SITE COMPUTER MANAGED DIRECTIVES"
270 PRINT
280 PRINT
290 PRINT"        CURRENT GRADES AND CUMULATIVE AVERAGE T-SCORES"
310 PRINT
320 PRINT TAB(26);"SECTION ";T
322 PRINT
324 PRINT
326 PRINT"      POST-TEST ";V$,L$;K$
330 PRINT
340 PRINT
350 PRINT"  NAME","RAW SCORE","T-SCORE","CAS","PSM"
360 PRINT TAB(16);"(N)";TAB(32);"(N)";TAB(44);"(A-N)";TAB(59);"(A-N)"
370 PRINT
380 PRINT
390 READ #3,A$,D,U
395 IF D=0 THEN 610
400 READ #4,U1,G,T1
410 IF U1 =D THEN 440
420 PRINT"I.D'S FOR ";A$;" OF SECTION ";T;" DO NOT MATCH "
430 PRINT" IN THE J AND M1 FILES"
435 GO TO 660
440 READ #5,U2,S,N
450 IF U2=U1 THEN 490
460 PRINT"I.D'S FOR ";A$;" OF SECTION ";T;" DO NOT MATCH"
470 PRINT" IN THE J AND K1 FILES"
480 GO TO 660
490 IF G>0 THEN 540
500 LET M=INT((S/N)+.5)
510 LET W=(C(M)/C(X1))*100
520 PRINT A$,"           ABSENT",M,INT(W+.5)
530 GO TO 590
540 LET S=S+T1
550 LET N=N+1
560 LET M1=INT((S/N)+.5)
570 LET W=(C(M1)/C(X1))*100
580 PRINT A$,G,T1,M1,INT(W+.5)
590 PRINT
600 GO TO 390
610 FOR I= 1 TO 10
620 PRINT
630 NEXT I
640 IF X<13 THEN 250
650 PRINT" THIS COMPLETES THE OUTPUT FOR THIS PROGRAM"
660 STOP
700 END
```

Figure 12—A copy of Program XN8 is listed to show the simple BASIC statements which can be used to manipulate student data files

the time and personnel required to punch student responses on paper tape. Nothing in the system design precludes each user from choosing the terminal best suited to his needs.

In summary, the implementation of CMI is essential if teachers are going to effectively manage the learning process to provide individualized instruction. On-line CMI satisfies the needs of educators for a rapid, flexible, easily accessible system. On-line CMI can currently perform the tasks of diagnosis, testing, record keeping, and analysis. Such a system is also capable of elucidating, validating, and implementing algorithms which provide individualized learning prescriptions.

ACKNOWLEDGMENTS

The authors wish to acknowledge the many helpful discussions and suggestions contributed by Dr. A. F. Vierling, Manager of the Honeywell Educational Instruction Network (EDINET), and Dr. A. T. Serlemitsos, Staff Physicist at Quality Educational Development, Incorporated.

REFERENCES

1 H J BRUDNER
Computer-managed instruction
Science Volume 162 pp 970-976 1968

2 *Revised listing of objectives*
Technical Reports Numbers 4.3a and 4.3b United States Office of Education Project Number 8-0446 November 3 1969

3 A F VIERLING
CAI development in multimedia physics
Technical Report Number 4.30 United States Office of Education Project Number 8-0446 November 3 1969

4 W A DETERLINE R K BRANSON
Evaluation and validation design
Technical Report Number 4.7 United States Office of Education Project Number 8-0446 November 3 1969

5 W A DETERLINE R K BRANSON
Design for selection of strategies and media
Technical Report Number 4.9 United States Office of Education Project Number 8-0446 November 3 1969

6 E H SHUFORD JR
Confidence testing: A new tool for management
Presented at the 11th Annual Conference of the Military Testing Association Governors Island New York 1969

7 W C GARDNER JR
The use of confidence testing in the academic instructor course
Presented at the 11th Annual Conference of the Military Testing Association Governors Island New York 1969

8 A F VIERLING A T SERLEMITSOS
CAI in a multimedia physics course
Presented at the Conference on Computers in Undergraduate Science Education Chicago Illinois 1970

Development of analog/hybrid terminals for teaching system dynamics

by DONALD C. MARTIN

North Carolina State University
Raleigh, North Carolina

INTRODUCTION

A recent study completed by the School of Engineering at North Carolina State University brought to light a very serious weakness in our program to employ computers in the engineering curricula, i.e., the inherent limitation on student/computer interaction with our batch, multiprogrammed digital system. The primary digital system available to students and faculty is the IBM SYSTEM 360/Model 75 located at the Triangle Universities Computation Center in the Research Triangle area. This facility is shared with Duke University and the University of North Carolina at Chapel Hill. In addition, the Engineering School operates a small educational hybrid facility consisting of an IBM 1130 interfaced to an EAI TR48 analog computer. We use some conversational mode terminals on the digital system but it has been our experience that they are of limited value in the classroom and, of course, only accommodate on the order of two students per hour.

It is our feeling that terminals based on an analog or hybrid computer would materially improve student/computer interaction, especially aiding the comprehension of those dynamic systems described by ordinary differential equations. This paper has resulted from our attempts to outline and define the requirements for an analog computer terminal system which would effectively improve our teaching in the broad area of system dynamics. The need for some reasonably priced dynamic classroom device becomes apparent when we consider the ineffectiveness of traditional lecture methods in such courses as the Introduction to Mechanics at North Carolina State University. This is an engineering common core course in mechanical system dynamics which has an enrollment of about 400 students per semester. This is a far cry from early Greek and Roman times when a few students gathered around a teacher, who made little or no attempt to teach facts but instead attempted to stimulate the students' imagination and enthusiasm. As pointed out by Professor Alan Rogers at a recent Joint SCI/ACEUG Meeting at NCSU, this intimate student-professor relationship simply cannot be achieved in today's large classes unless the instructor learns to make effective use of the modern tools of communication, i.e., movies, television, and computers. In effect, we turn students off by our failure to recognize the potential of these tools, especially the classroom use of computers.

The material which follows points out the need for interactive terminals, describes the capabilities of prototype models already constructed, and then outlines the classroom system which is presently being installed for use in the fall of 1970.

THE NEED FOR INTERACTIVE TERMINALS

Classroom demonstrations

The computer has long held great promise both as a means for improving the content of scientific and technical courses and as an aid for improving methods of teaching. While some of this promise has been realized in isolated cases, very little has been accomplished in either basic science courses or engineering science courses in universities where large numbers of students are involved. It is certainly true that significant improvement in course content can be achieved by using the computer to solve more realistic and meaningful problems. For example, the effects of changing parameters in the problem formulation can be studied. With centralized digital or analog computing facilities, this can be accomplished only in a very limited way, e.g., problems can be programmed by the instructor and used as a demonstration for the class. Some such demonstrations are desirable but it is impossible to get the student intimately involved, and at best they serve only as a supplement to a text book. At North Carolina State

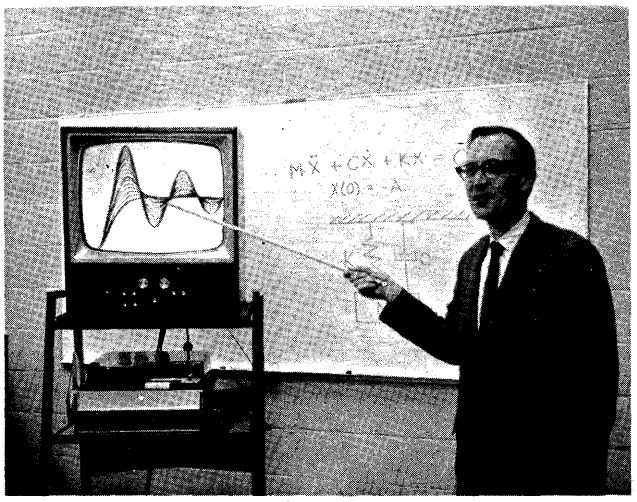


Figure 1

University we have developed a demonstration monitor for studies of dynamic systems which seems to be quite effective. We recently modified our analog computers so that the primary output display device is a video monitor which can be used on-line with the computer. Demonstrations have been conducted for other disciplines, for example, a Sturm Liunville quantum mechanics demonstration for a professor in the Chemistry Department. This demonstration very graphically illustrates the concept of eigenfunctions and eigenvalues for boundary value problems of this type. A picture of the classroom television display is shown in Figure 1. The instructor's control panel makes several parameters, function switches and output display channels available for control of the pre-patched demonstration problem. Switches are also available to control the operation of the analog computer. The direction we are proceeding in the area of demonstration problems is to supply the instructor with a pre-patched problem, indicate how to set potentiometers, and how to start and end the computation. Since the display is retained on a storage screen oscilloscope with video output, he can plot multiple solutions which will be stored for up to an hour, or he can push a button to erase the screen at any time. Some demonstrations have been put on the small video tape recorder, shown in Figure 1, but we find the loss of interactive capability drastically decreases the effectiveness of the demonstration.

Student programming

In addition to classroom demonstrations, the student can be assigned problems and required to do the programming. While we believe this to be an excellent approach for advanced engineering courses, such as design, systems analysis, etc., it has proven less than satisfactory for the first and second year science and engineering courses. Even when the students have had a basic programming course, valuable classroom time must be spent in techniques for programming, numerical methods, and discussions of debugging. The students tend to become involved in the mechanics of programming at the sacrifice of a serious study of the problem and its interpretation. With inexperienced students, even when turnaround time on the computer is excellent, the elapsed time between problem assignment and a satisfactory solution is usually much too long.

We have tried this student programming approach for the past four years in a sophomore chemical engineering course. While it has been of some value, we are now convinced that it is not a satisfactory method of improving teaching. The teaching of basic computer programming does, however, have a great deal of merit in that it forces students to logically organize their thoughts and problem solving techniques. Also it helps to provide an understanding of the way computers can be applied to meaningful engineering problems. Thus, we intend to continue the teaching of basic digital computer programming in one of the higher level languages at the freshman or sophomore level, and then make use of stored library programs for appropriate class assignments. In addition, we will continue to assign one or more term problems in which the student writes his own program, especially in courses roughly categorized as engineering design courses.

Digital computer terminals

It is appropriate at this point to emphasize why we feel time-shared or conversational mode terminals are not the answer to our current problem. It has been our experience that the conversational terminal is an outstanding device for teaching programming, basic capabilities of computers, and solving student problems when the volume of data is limited. However, if the relatively slow typing capability of students is considered, we have found that a class of 20 or 30 students can obtain much faster turnaround time on the batch system. To be sure, the student at the terminal has his instantaneous response, but the sixth student in the queue is still waiting two hours to run his program and use his few tenths of a second CPU time. One can certainly argue that this is an unfair judgment since the solution is to simply buy more terminals for about

$2500 each. Unfortunately, these terminals are like cameras and their use involves a continuing expense often greater than the initial cost. Connect time and communication costs for a sufficient number of terminals have discouraged such terminal use on the campus at the present time. The experience of the North Carolina Computer Orientation Project, which essentially provided a free terminal for one year at have-not schools, has been similar in that it proved very difficult for these schools to utilize the terminal in any science course other than a course in programming.

Present limitations on classroom use of computers

It is reasonable to ask why, in a university that has had good computing facilities for some time, computer use in the classroom is so limited. We feel there are several reasons why the majority of instructors do not use this means of communication to improve instruction techniques. The first of these reasons must be classed as faculty apathy. There is no other explanation for the fact that less than 25 percent of our engineering faculty use the digital computer and less than 5 percent avail themselves of our analog facility. Admittedly, it is extremely difficult for a physicist, chemist or engineer who is not proficient in computing to program demonstration problems for his classes. Because such demonstrations, while a step in the right direction, do not really make use of the interactive capability of the computer to excite the students' imagination, there is often little motivation for the professor to learn the mechanics of programming. Fortunate indeed is the occasional instructor who has a graduate student assistant competent to set up and document such demonstrations.

The second reason, closely coupled to the first, is that computer output equipment for student use in the classroom is either not available or just too expensive for large classes. Digital graphics terminals, for instance, sell for between 15 and 70 thousand dollars, depending on terminal capability, and an analog computer of any sophistication at all will cost 5 to 10 thousand dollars with the associated readout equipment. In our basic introductory mechanics course, with ten sections of forty students, a minimum of twenty such terminals would be required if we assume two students per terminal as a realistic ratio. Even if such analog or digital computer terminals were available, we would then be faced with the problem of teaching the students (and faculty) a considerable amount of detailed programming at the expense of other material in the curriculum. We feel that analog/hybrid computer terminals designed to accomplish a specific, well-defined task, will provide an economical interactive student display terminal for many engineering courses. Such a terminal is described in this paper.

CLASSROOM TERMINAL SYSTEM

We have recently received support from the National Science Foundation to study the effect of student interaction with the computer in courses which emphasize the dynamic modeling of physical systems. It is a well known fact that interaction with a computer improves productivity in a design and programming sense. The question to which we are seeking the answer is: Will computer interaction also improve the educational process effectively without leaving the student with the impression that we are using a "magic black box"? To accomplish this goal, we are installing sixteen analog/hybrid terminals in a classroom to serve thirty-two students. The classroom in which these terminals will be placed is about 150 feet from the School's analog and digital computer facility.

At this point, we should place some limits on terminal capability and function. If we accept the premise that the student need not learn actual patching to use the analog computer terminal and eliminate the traditional concept of submitting a hard copy of his computer output as an assignment, the desirable features of a terminal might be as follows:

1. *Parameters:* The student must have the capability of varying at least five different parameters in a specific problem. Three significant digits should be sufficient precision for these parameters and their value should be either continuously displayed or displayed on command.
2. *Functions:* The student should have access to function switches to perform such operations as changing from positive to negative feedback to demonstrate stability, adding forcing functions to illustrate superposition, adding higher harmonics to construct a Fourier approximation of a function, introducing transportation delay to a control system, etc. Three to five of these function switches should be sufficient.
3. *Problems:* The student should be able to select several different problems, say four, at any of the individual terminals. Depending on the size of the analog computer, the student could use the terminal to continue a study of a problem used in a prior class, compare linear and nonlinear models of a process, etc.

4. *Response Time:* The response time for each of twenty terminals should be about one to three seconds, i.e., the maximum wait time to obtain one complete plot of his output would be something like one second plus operate time for the computer. Computer operate time has been selected as 20 milliseconds for our equipment although a new computer could operate at higher speeds if desired.
5. *Display:* The display device for each terminal must be a storage screen oscilloscope or refreshed display for x-y plots. Zero and scale factors must be provided so that positive and negative values and phase plane plots can be plotted. Scaling control must be presented in a manner which is easy for the student to use and understand.
6. *Output selector:* The student should be able to select from four to five output channels for display on the oscilloscope.
7. *Instructor display:* The instructor should have a terminal with a large screen display which the entire class can observe. His control console should have all the features of the student terminals and should also have the capability for displaying any one of the student problem solutions when desired. He should also have master mode controls to activate all display terminals. It would be advantageous for the instructor to have a small screen display to monitor student progress without presenting the solutions to the class on the large screen.

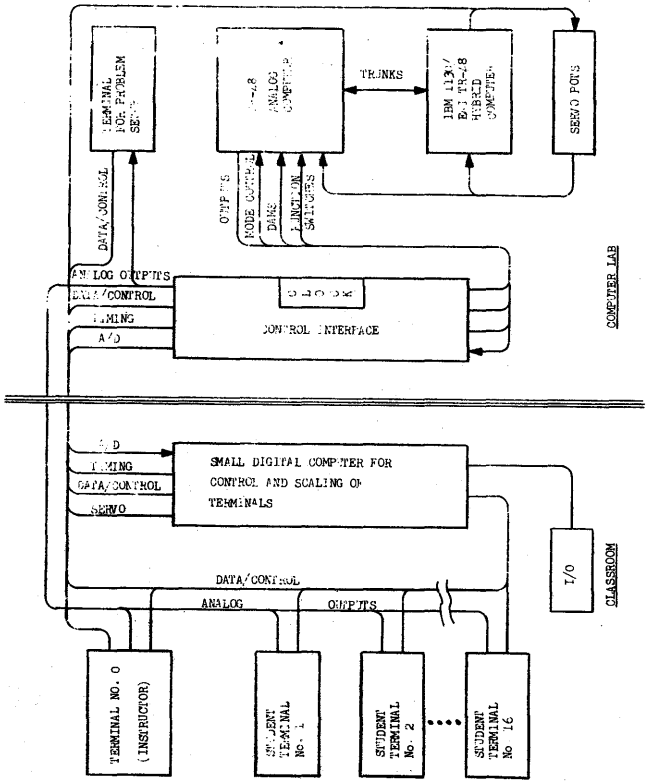


Figure 2—Flow sheet for proposed analog/hybrid terminal system

Given a terminal system with these features, we have then defined the primary objective as being a study of the use of this classroom in some specific courses. We are initiating this evaluation with one course in Engineering Mechanics, Introduction to Mechanics (EM 200) and two courses in Chemical Engineering, Process Analysis and Control (CHE 425), and Introduction to System Analysis (CHE 225). Thus, we start our evaluation with one sophomore engineering core course with three to four hundred students per semester and two courses with thirty to forty students per semester at the sophomore and senior level. In addition to these two courses, we are attempting to schedule as many demonstrations of the system as possible for other departments in the hope of stimulating their imaginative use of the terminals.

A flow sheet for the classroom terminal system is given in Figure 2. The system includes a small digital mini computer which is to be used for control, storage, and scaling of terminal information. The system consists of the following components:

a. Sixteen student terminals
b. One instructor terminal
c. Digital mini computer with I/O device for programming
d. Control interface to analog computer

The inclusion of a small digital computer in this terminal system opens up some very interesting future possibilities such as using the terminals for digital simulation as well as analog. As will be seen in the next section, the digital computer provides scaling so that parameters can be entered in original problem units. It also acts as temporary storage for each terminal as it awaits service and controls the terminal system. The system is designed to operate in the following manner. The instructor informs the computer operator that he wishes to use problem CHE 2 during a certain class or laboratory period. The operator places the proper problem board on the TR-48 and then sets the servo potentiometers which are not controlled by the student terminals and static checks the problem with the hybrid computer. He also sets up the program CHE 2 on the mini computer just prior to class. From this point on the system is under the control of the instructor and students.

FUNCTIONAL DESCRIPTION OF TERMINAL

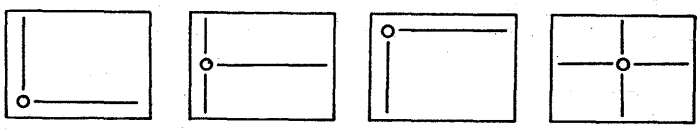
General

The basic terminal configuration is shown in Figures 3 and 4. All display functions are located on the upper display panel and control or data input is provided on the inclined control panel.

Control

The control functions available to the student include power on-off, store, non-store, erase, and trace intensity. These controls are located on either side of the oscilloscope display as shown in Figure 4. Indicator lights are also provided in this area for terminal identification, error and terminal ready status. The erase function is used quite often by the student and thus is also available on the keyboard.

In addition to the basic operating controls, the student can request either a single solution or repeated solutions on his display unit. For the single solution mode, he displays one solution as soon as the sequencer reaches his terminal address. In the repeat solution mode, his oscilloscope is unblanked and displays a solution each time the sequencer reaches his address.



Display Scaling

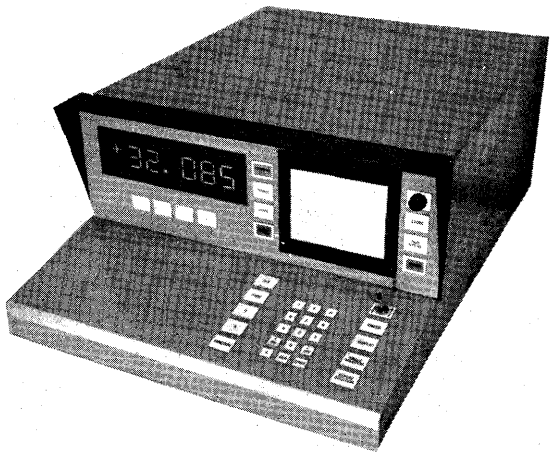
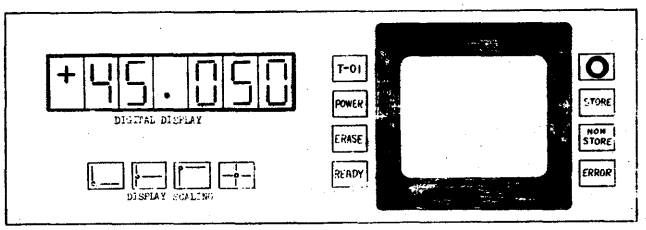


Figure 3



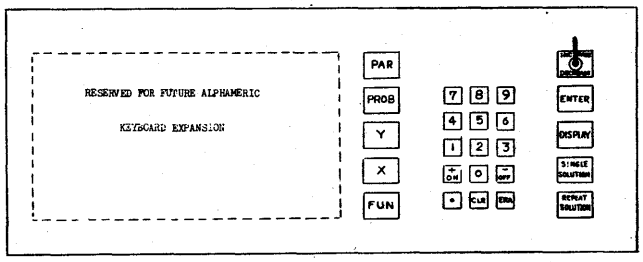


Figure 4—Terminal display and control panels

The worst case response time for either mode would be on the order of one second, even if all other terminals had solution requests pending.

Output display and scaling

The primary output display device is a Tektronix type 601 storage screen oscilloscope. This oscilloscope is mounted directly behind the vertical front display panel as shown in Figures 3 and 4. These oscilloscopes are modified to provide for scaling as shown below. The operating level circuit is modified to provide a switched store and non-store operation for the user. Output scaling is automatically selected when the student depresses any one of the four push buttons on the left hand side of the display panel. The picture which indicates the normalized scaling is printed on the face of the lighted push button switch.

The left hand switch scales the output signal to display positive X and Y values. The second switch displays positive X and allows for both positive and negative values of the Y variable. Phase plane plots can be displayed by selecting the right hand switch in this set. We have been using this type of output scaling for oscilloscopes in our laboratory for over a year with very satisfactory feedback from student users.

The student must have the option of selecting from several X and Y analog output lines. This option is provided at the terminal by depressing either [X] or [Y] then the appropriate number on the keyboard, and then the [ENTER] key. For example, if

the student is instructed to plot X_1 versus Y_4, he would actuate the following keys on the terminal keyboard as shown below:

[X] [1] [ENTER] [Y] [4] [ENTER]

The digital software sets up the linkage between the requested output line and the control unit which switches variable outputs for specific problems to the two analog output lines leading to the classroom. All analog outputs are on the two X and Y lines, but each terminal Z line is energized only at the appropriate time, i.e., in answer to a solution request for that terminal.

Parameter entry

There are many ways in which parameters can be set from an analog or hybrid computer terminal. In the first terminals we constructed, parameters were simply multiplexed potentiometers connected to the analog computer with long shielded cable. Thumbwheel switches can be effectively used to set digital to analog coefficient units or servo potentiometers at the analog computer. Since this hybrid terminal system includes a small digital computer, parameters will be entered with a fifteen digit keyboard as indicated in Figure 4.

The parameter function switch, [PAR], keyboard, and [ENTER] keys are used in the following sequence.

Suppose the student wishes to set parameter number four at a value of 132. He would depress switches in the following sequence:

[PAR] [4] [1] [3] [2] [ENTER]

or [PAR] [4] [1] [3] [2] [.] [ENTER]

If the parameter number three were less than unity, say 0.05, he would enter

[PAR] [3] [0] [.] [0] [5] [ENTER]

or [PAR] [3] [.] [0] [5] [ENTER]

This system allows the student to enter parameter values in the actual problem units, e.g., if the input temperature for a heat exchanger simulation is 150°F, he sets this value rather than some normalized fraction.

If an error is made before [ENTER] is depressed, the register can be cleared with the [CLR] key. The use of actual rather than normalized parameters requires additional registers in the terminal but is essential for beginning students. We must remember that they are studying the dynamic system, not analog computer scaling. It is also in keeping with the concept of using the terminal as input and output for the hybrid computer. If the parameters represent frequency, temperature, pressure, etc., they should be entered as they appear in the problem statement if at all possible.

Since parameter values are scaled by a program in the digital computer, scientific notation can also be used to permit both E and F format data entry from the hybrid terminal. The digital software interprets the input data to separate the mantissa and exponent portions of the number entered in E format. For example, the student might enter

[PAR] [2] [+] [1] [.] [5] [−] [4]

[ENTER]

and the digital computer would convert this number to +0.00015.

Reading parameter values

One significant advantage of the thumbwheel switch parameter entry as opposed to the keyboard is the ability to remember a specific parameter value at any time. If the student forgets the value, he needs to be able to display it at the terminal on request. This capability is provided by a six character plus sign display module located in the upper left corner of the terminal as shown in Figures 3 and 4. This display unit automatically shows the student that his parameter value was correctly entered at the keyboard and accepted by the digital computer. The format of the output display is controlled by the digital computer software. In addition to displaying the correct value of the parameter when entered at the keyboard, the

value of any parameter previously set can be indicated using the DISPLAY key. Suppose the student wishes to retrieve but not change the value of parameter three. He would press PAR, then 3 on the keyboard, and then the DISPLAY key. The digital computer software then causes the proper value to be displayed and light keyboard button number 3 to identify the requested parameter number.

A separate digital display module could be used to indicate the requested parameter number, but lighting the keyboard number has some advantage when displaying the status of function switches as noted later.

Analog/digital readout

One of the advantages which immediately becomes apparent when the terminal includes digital readout and a small digital computer is the capability of returning numbers which are the result of an iterative calculation. A first order boundary value problem where the unknown initial condition or time constant is sought would be one illustration. Another example which we have been using in a basic process instrumentation course is to demonstrate the operation of a voltage to frequency converter or time interval analog to digital converter.

Any digital or analog number can thus be returned to a terminal by selecting one of the output lines for display. The student presses Y, followed by the appropriate number on the keyboard, and then the DISPLAY function key. This display feature is also extremely valuable in presenting calculated results from the hybrid computer through the trunk lines as indicated in the overall system diagram, Figure 1.

One variable parameter

The availability of a control which can vary one parameter through some range of values is an important feature of the terminal. Thumbwheel switches and keyboard entry of parameters are fine but tend to be somewhat slow when the student is interested in observing the effect of a range of parameter variations on a simulation or fitting a model to experimental data points. A potentiometer on the terminal, as we have employed in the past, avoids this problem but involves transmission of analog signals over long lines. As a compromise, either a two or four position switch is used to increment the value of any selected parameter at a rate determined by the digital software. Thus, the student increases or decreases a particular parameter value at either a fast or slow rate with this switch. The sequence of operations would be as follows: suppose the student wishes to vary parameter four through a range of values to observe its effect on the solution. He might choose a starting value of zero for this parameter which, for example, might represent the damping coefficient in a linear, second order system. He presses PAR, followed by the numbers 4 and 0 on the keyboard and then selects the ENTER key. He then selects the REPEAT SOLUTION mode to observe the solution each time the sequencer reaches his terminal number. If the student has selected the STORE mode, he can then plot a family of curves as he increases the damping factor from zero to unity by pushing the parameter slewing switch in the increase direction. A four position switch allows a choice of either fast or slow incremental changes in the parameter value. Another obvious application for this function is in curve fitting of experimental data with one or more parameters.

Function switches

Control of the electronic function switches is provided at each terminal. To set function switch one in the ON position, the student presses FUN, followed by the number 1 and +ON on the keyboard and then the ENTER key. If he wishes to know the present state of any function switch, he presses FUN, the switch number, and then the

DISPLAY key. The terminal response is to light the function switch number and either the + ON or − OFF keys on the keyborad to indicate the present state of that particular switch. These function switches can be used in any way desired by the instructor, e.g., adding successive terms of a power or Fourier series to demonstrate the validity of these approximations, adding various controller modes in a process control simulation, etc.

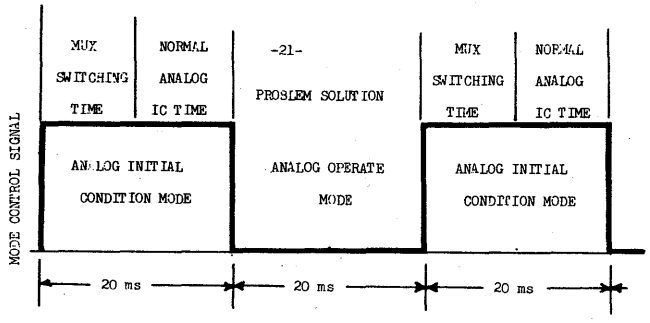
The instructor's terminal

The instructor's terminal is designated as terminal number zero. This terminal uses as its primary output device a Tektronix Type 4501 storage oscilloscope instead of the Type 601. Since this scanning oscilloscope has video output, the instructor can display his solution on the closed circuit television monitor for the class at any time. In addition, the instructor has the capability of unblanking any or all student terminals to let them have a "copy" of his solution to compare with their own. He can also unblank his terminal and pick selected student solutions for display to the rest of the class.

SOFTWARE

Basic operating system

The basic software to serve the analog terminals is written in assembly language for the PDP-8 control computer. This is a 4K machine with hardware multiply and divide, although this feature is not essential for terminal operation. The basic cycle time for the system is controlled by the analog clock which alternately places the analog computer in the initial condition and operate modes every twenty milliseconds. The first ten milliseconds of each initial condition period is to ensure adequate time for problem and function selection by the relay multiplexer. The second ten milliseconds is the normal initial condition time to charge the integrating capacitors as shown below.



A terminal user can activate an action key at any time, e.g., ENTER , DISPLAY , SINGLE SOLUTION , or REPEAT SOLUTION .

This request for action, along with the necessary data and address is stored in a 32 bit shift register in each terminal. As each terminal is interrogated in sequence by the PDP-8 the action bit is tested. If the user wants service, his data is transferred to a specific core area. For instance, suppose he wishes to set parameter number 1 at a value of 0.32. He activates the following keys: PAR 1 0 · 3 2 ENTER . The ENTER key is the action code in this instance. When the sequencer reaches his terminal, this data is transferred to storage in the proper memory locations in the PDP-8. A similar action is taken to set function switches, and select problems or output channels.

The basic operating system software controls all of these action operations. When a solution is requested, the parameters, functions, and outputs, along with an unblanking signal, is sent back to the terminal during the next analog computer operate cycle. The basic system software also converts the floating point parameter values supplied by the student to integer values used by the digital coefficient units or digital to analog converters. This feature of floating point data entry requires that the instructor provide the scaling information for each problem as described next in the application software section.

Application software

The application software is written in a special interactive language developed for the PDP-8. This language makes use of a cassette tape recorder in our system, but could be used from the teletype if necessary. The information required by the terminal operating system to convert floating point to integer parameters is their maximum and minimum values. When the instructor is setting up the terminal problem, the computer software solicits responses similar to the following:

IDENTIFY YOUR PROBLEM NUMBER

The instructor would then type,

CHE 4

The computer responds with

PROVIDE MAXIMUM AND MINIMUM
VALUES FOR THE PARAMETERS

If the instructor wishes to give the student control over parameters one and two, he types

PAR 1, MAX 50, MIN 25
PAR 2, MAX 0.5, MIN 0
END PAR LIST

A similar conversational procedure is used to identify function switches, problems, and analog computer output channels. In our system, this scaling and switching data is stored on the magnetic tape cassette. A paper tape unit could also be used if desired. When the instructor wishes to use the terminal system at a later date, he places his cassette in the tape deck and the proper problem board on the analog computer. From this point on, the problem or problems can be controlled from the individual terminals.

COSTS

The cost of a system such as described in this paper is naturally dependent on the number of terminals involved. Since our system was developed jointly with Electronic Associates, it is difficult to evaluate the actual development and design costs. The individual terminals, including a type 601 Tektronix storage oscilloscope should be on the order of $3500 to $4000 each. Mini computers such as used in this system would range from $6000 to $10,000 and cassette tape systems are available for about $3000. The major question mark in the estimation of system cost is the hybrid control interface to couple the analog and digital computers. If a special interface could be developed for about $10,000, the cost of a ten terminal system would be on the order of $60,000. This system could be coupled to any analog computer and, of course, provides basic hybrid capability as well as terminal operation. If a hybrid computer were already available, the terminals could be added for about $3500 to $4000 each plus wiring costs.

CONCLUSION

The key to student and instructor use of these terminals is the development of appropriate handout materials. Several of these handouts have been written in programmed instruction form and have resulted in very favorable feedback from students who used early models of the terminals. Although the complete classroom system will be used for the first time in the fall of 1970, we have been very gratified with student acceptance of the few terminals now in use. Laboratory reports now consist of answering specific questions concerning the dynamic system under study rather than computer diagrams and output. Also, the student can really proceed at his own pace, and return at any time to repeat a laboratory exercise simply by giving the computer operator the problem number. We are excited about the potential of this classroom terminal system and believe that we will see significant improvement in the students' understanding of dynamic systems as the system is used in additional curricula.

Computer tutors that know what they teach

by L. SIKLÓSSY

University of California
Irvine, California*

INTRODUCTION

Computer tutors hold the promise of providing truly individualized instruction. Lekan[1] lists 910 Computer Assisted Instruction (CAI) programs, and this large number demonstrates the wide interest in the field of computer tutors.

The computer is eminently suited for the bookkeeping tasks (averaging, record keeping, etc.) that are usually associated with teaching. In such non-tutorial tasks, the computer is greatly superior to a human teacher. On the other hand, in strictly tutorial tasks, the computer is usually handicapped. In particular, CAI programs seldom know the subject matter they teach, which can be seen by their inability to answer students' questions.

We shall consider the structure of tutorial CAI programs and discuss some of their shortcomings. Some of the latter are overcome by generative teaching systems. Finally, we shall outline how we can construct computer tutors that know their subject matter sufficiently to be able at least to answer questions in that subject matter.

"IGNORANT" COMPUTER TUTORS

Most CAI programs have a structure very close to that of mechanical scrambled textbooks. These textbooks and their immediate CAI descendants, which we shall call selective computer teaching machines,* consist of a number of frames. Figure 1 shows the structure of a frame of a selective computer teaching machine.

The computer tutor may either start the frame with some statements (box labelled K2), or directly ask the student some question (K3). The student's answer (K4) is compared to a finite store of anticipated responses (K5). The answer itself may have been forced into a limited domain (multiple-choice question), or it must match exactly or partially (through key-words) some stored answers.

The result of the diagnostic is submitted to a strategy program (K6). The strategy program may use additional data, past performance for instance, to determine the next frame of the course.

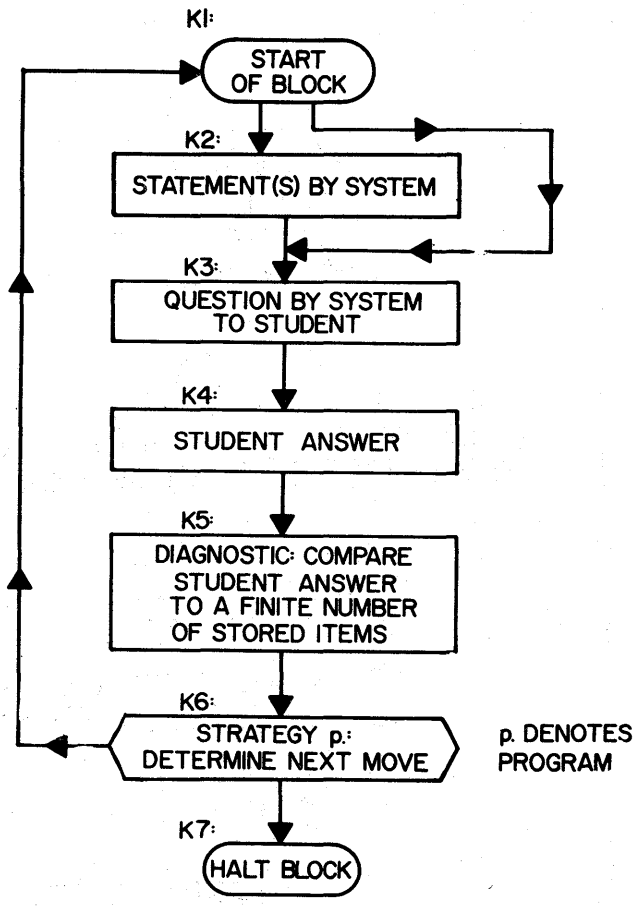


Figure 1—Frame of a selective computer teaching machine

* Present address: University of Texas, Austin, Texas.
* To use the terminology of Uttal.[5]

Disregarding the bookkeeping tasks that the computer tutor can perform, we shall concentrate on the structure of the tutor. The two major criticisms that have been levied at selective computer teaching machines are:

a. Their rigidity. Questions and expected answers to these questions have been prestored.
b. Their lack of knowledge. They cannot answer a student's questions related to the subject matter that is being taught.

GENERATIVE TEACHING MACHINES

In an effort to overcome the rigidities of selective computer teaching machines, some researchers have developed generative teaching machines.

Figure 2 describes a frame of a generative teaching machine. In this case, the computer tutor, instead of retrieving some question or problem, generates such a question or problem, a *sample* of some universe. The generation is accomplished by a program called the generator program (L2).

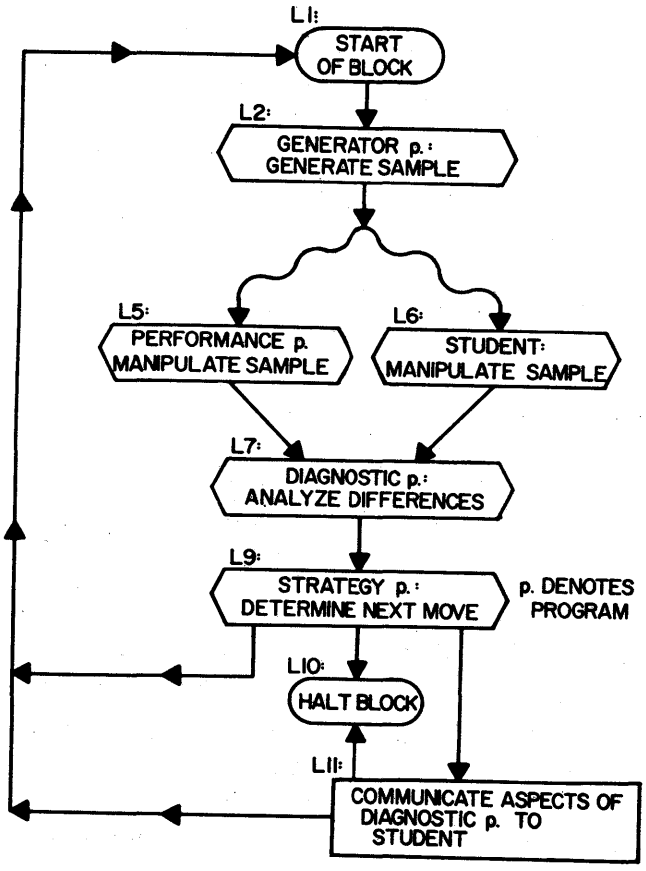


Figure 2—Frame of a generative teaching machine

The sample is presented to the student who tries to manipulate the sample: i.e., answer the question or solve the problem. Concurrently* another program, the performance program, manipulates the sample (L5). The performance program knows how to manipulate samples in the universe of the subject matter that is being taught.

Before continuing with our description, we shall give some examples of generative teaching machines. Uhr[2] has described a very elementary system to teach addition. A program by Wexler[3] teaches the four elementary arithmetic operations. Peplinski[4] has a system to teach the solution of quadratic equations in one variable. Uttal *et al.*[5] describe a system to teach analytic geometry.

When Wexler's system teaches addition, the generator program generates a sample, namely two random numbers that satisfy certain constraints. An example would be: four digits long, no digit greater than 5. (Note that the number of possible samples may be very large.) The performance program simply adds the two numbers.

A diagnostic program (L7) analyzes the differences between the answers of the student and the performance program. In Wexler's system, the numbers given by the student and the system may or may not be equal. If unequal, the diagnostic program may determine which digits of the answers are equal, which number is larger, etc.

The findings of the diagnostic program, together with other information (such as past student performance), are given to a strategy program (L9). This program may decide to communicate some aspects of the diagnosis to the student—for example: "Your answer is too small." (L11); it may halt (L10); or transfer to a new or the same frame (L1). Transferring to the same frame is not an identical repetition of the previous frame, since usually the generator program will generate a different sample.

In a generative computer tutor, questions are no longer prestored but are generated by a program. Since the questions are not usually predictable with exactitude, a performance program is needed to answer them. The performance program is at the heart of a computer tutor that knows what it teaches.

PROGRAMS THAT KNOW WHAT THEY TEACH

The performance program of a generative computer tutor can solve the problems in some universe; in other words, we may say that the program knows its subject

* This is the meaning of the wavy lines in the flowchart.

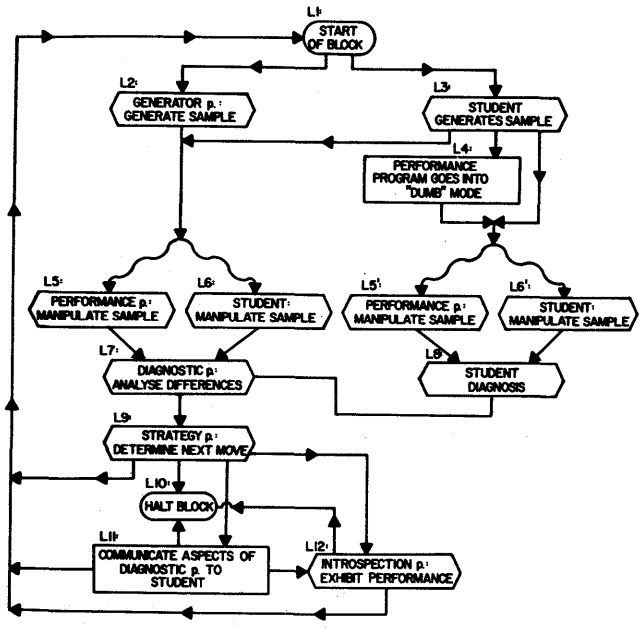


Figure 3—Frame of a computer tutor that knows what it teaches

matter. We can use the performance program in two additional ways beyond its use in generative tutors. The performance program may answer questions generated by the student. It can also explain how it answers some questions and thereby teach its own methods to the student.

Figure 3 describes a knowledgeable computer tutor. The path of boxes L1, L2, L5, L6, L7, L9, L10 and L11 has been discussed above in the framework of a generative tutor. The function of box L12 is to explain to the student the problem-solving behavior of the performance program in those cases when the behavior of the performance program can be fruitfully imitated by a human being.

In box L3, the student is allowed to generate samples. The previous path can then be followed: both student and computer tutor can manipulate the samples with the tutor helping the student. The tutor can also manipulate the sample directly, thereby, in effect, answering a student's question (path L3, L5′, L6′). We can even let the tutor make mistakes (L4, L5′), which gives the student an opportunity to catch the teacher in error (L8).

It is important to let the student generate samples so that he can find out what happens in particular cases about which he feels unsure. It is impossible to preprogram a set of samples that would be satisfactory to all students. In addition, some experimental evidence (Hunt;[6] Crothers and Suppes[7]) indicates that learning occurs faster when students generate their own examples.

A COMPUTER TUTOR THAT KNOWS SOME SET THEORY

We shall illustrate the framework of a knowledgeable computer tutor by a program that teaches set union. The subject matter was selected for the ease with which appropriate performance and diagnostic programs could be written.

The programming language is the LISP 1.6 version of LISP running on the PDP–10 computer. The program is not completely polished and the outputs have been slightly edited. A more definitive version of the program will be described elsewhere, but we hope to give here enough feeling for the power of such a computer tutor to be appreciated.

Sets are represented by unordered lists. The sets $(A(B\ C))$ and $((C\ B)A)$ are equal. Each contains two elements, one of which is itself a set. ( ) is the empty set.

A dialogue between the computer tutor and a student follows. We first consider the computer generative mode:

Computer: What is the union of (HI 32) and ((DOG))?
Student: (HI).
C.: Your answer is not right. Do you see why?
S.: No.
C.: You missed some elements. The following also belong to the union: (DOG) 32. The union is ((DOG) HI 32) . . .

The output does not make it evident that the sets were generated using a random set generator. The set generator generates sets of a given number of elements and of a given maximum depth. (A set with no subsets as elements has depth 0.) The atomic elements of a set are randomly selected from an array which includes the atoms HI, 32 and DOG. The length and depth of the set are themselves random numbers in ranges that are input to the set generator routine.

The program computes the union of the sets (HI 32) and ((DOG)). The union is the set ((DOG) HI 32). This result is compared to the student's response. The computer tutor analyzes the differences between the desired and obtained answers and, in this case, exhibits the terms missing from the student's answer.

The path we have followed in Figure 3 is L1, L2, L5,

L6, L7, L9, and L11. We shall follow a different path below and let the student generate sets.

C.: Type two sets and I will tell you their union.
S.: (AS AS) (B D).
C.: (AS AS) is not a set. Do you understand why?
S.: No.
C.: AS are duplicate elements in your first set. Type two more sets.
S.: (A V D) ( ).
C.: The union of (A V D) and ( ) is (A V D) . . .

The inputs proposed by the student are checked for well-formedness. The program finds that (AS AS) is not an acceptable representation for a set since the element AS is repeated. An appropriate diagnostic message is given. The path in Figure 3 was L1, L3 and L5′, with a return to L1. Next the student wants to determine how union operates with the empty set. The path followed was again L1, L3, and L5′.

We allow the tutor to make mistakes and give the student a chance to catch them.

C.: Would you like to quiz me? You type in two or more sets and I will find the union. Then you type whether I am right or wrong . . .
S.: (R S T) ( ).
C.: The union is (R S T).
S.: Wrong.
C.: No, you're wrong. I was right. Do you see why?
S.: No.
C.: Type what you think the union is.
S.: (R S T).
C.: But that's what I typed. Type some sets.
S.: (A B) (A).
C.: The union is (A).
S.: Wrong.
C.: Oh my goodness! I did make a mistake . . .

We expanded the previous path to include box L4. The tutor now may make mistakes, and the student has an opportunity to play teacher. If, when asked "Type what you think the union is," the student had typed "(T R S)," the tutor's response would have been: "The sets (R S T) and (T R S) are equal." Here we have followed the path L1, L3, L4, L5′, L6′, L8, L7, L9 and L11.

The tutor gives a final test to the student. In it, the following appears:

C.: What is the union of the following sets: (BIRD ( ) S7) (S7 (DOG) 32 S7)?
S.: Impossible.
C.: You caught me. The second list was not a set.

There is nothing like giving impossible problems!

TABLE I—Diagnostic Program for Set Union and Some Possible Comments

Cases Determined by Diagnostic Program	Possible Partial Comments to Student
1. $A = S_1 \cup S_2$ (set equality)	Your answer is correct.
2. $A \neq S_1 \cup S_2$	Your answer is incorrect.
2-1. $(S_1 \cup S_2) - A \neq ( )$	You left out some element(s).
2-1-1. $((S_1 \cup S_2) - A) \cap S_1 \neq ( )$	You left out some element(s) of the first set.
2-1-2. $((S_1 \cup S_2) - A) \cap S_2 \neq ( )$	You left out some element(s) of the second set.
2-2. $A - (S_1 \cup S_2) \neq ( )$	Some element(s) in your answer are neither in S_1 nor in S_2.

We have not yet coded the introspection program that would explain to the student how the performance program calculates set unions. Table I lists diagnoses that can be used in teaching set union. The two sets are S1 and S2: the student's answer is A. We assume that all sets have been checked for well-formedness. $\cup$, $\cap$ and $-$ denote set union, intersection and difference.

The tutor can diagnose not only that (for instance in Table I, case 2–2) some elements in the answer should not have been there, but also tell the student which elements these are. Table II lists diagnoses that can be used in teaching set intersection. The two tables show how algorithmic computations allow the computer tutor to pinpoint the student's errors.

TABLE II—Diagnostic Program for Set Intersection and Some Possible Comments.

Cases Determined by Diagnostic Program	Possible Partial Comments to Student
1. $A = S_1 \cap S_2$ (set equality)	Your answer is correct.
2. $A \neq S_1 \cap S_2$	Your answer is incorrect.
2-1. $(S_1 \cap S_2) - A \neq ( )$	You left out some element(s) which belong to both S_1 and S_2.
2-2. $A - (S_1 \cap S_2) \neq ( )$	Some elements in your answer do not belong to both S_1 and S_2.
2-2-1. $(A - S_1) \cap S_2 \neq ( )$	Some element(s) in your answer belong to S_2 but not to S_1.
2-2-2. $(A - S_2) \cap S_1 \neq ( )$	Some element(s) in your answer belong to S_1 but not to S_2.
2-2-3. $A - (S_1 \cup S_2) \neq ( )$	Some elements in your answer are neither in S_1 nor in S_2.

The symbol-manipulating capabilities required of the computer tutor would be difficult to program using one of the computer languages that were designed specifically to write CAI programs.

RECIPE FOR A KNOWLEDGEABLE COMPUTER TUTOR

The framework of Figure 3 shows explicitly how a knowledgeable computer tutor can be built. First, we need a performance program which can do what we want the student to learn. We have programs that appear to know areas of arithmetic, algebra, geometry, group theory, calculus, mathematical logic, programming languages, board games, induction problems, intelligence tests, etc. The computer models which have been developed in physics, chemistry, sociology, economics, etc., are other examples of performance programs. To complete the computer tutor, attach to the performance program appropriate generator, diagnostic, strategy and introspection programs.

We used our recipe for a knowledgeable computer tutor to develop a tutor to teach elementary set theory and gave examples of the capabilities of this tutor. The manpower requirements for the development of a computer tutor are considerable and we have not applied the recipe to other areas. Our demonstration, therefore, remains limited but we hope that it was sufficiently convincing to encourage other researchers to develop more knowledgeable and powerful computer teaching systems.

The major difficulty that we experienced was in the area of the topic of understanding of the diagnostic program. In particular, linguistic student responses could not be handled in general. Presently, we only accept very limited student answers expressed in natural language. The development of computer programs which better understand language* would lead to a much more natural interaction between student and tutor.

CONCLUSION

Most CAI programs cannot answer student questions for the simple reason that these programs do not know the subject matter they teach. We have shown how programs that can perform certain tasks could be augmented into computer tutors that can at least solve problems or answer questions in the subject matter under consideration. We gave as an example a program to teach set theoretical union and showed the diagnostic capabilities of the tutor. These capabilities are based on programs and are not the result of clever prestored examples.

The student-tutor interaction will become less constrained after enough progress has been made in computer understanding of natural language.

* See Simmons[8] for an effort in that direction.

ACKNOWLEDGMENTS

J. Peterson and S. Slykhous contributed significantly to this research effort.

REFERENCES

1 H A LEKAN
Index to computer assisted instruction
Sterling Institute Boston Mass 1970

2 L UHR
The automatic generation of teaching machine programs
Unpublished report 1965

3 J D WEXLER
A self-directing teaching program that generates simple arithmetic problems
Computer Sciences Technical Report #19 University of Wisconsin Madison Wisconsin 1968

4 C A PEPLINSKI
A generating system for CAI teaching of simple algebra problems
Computer Sciences Technical Report #24 University of Wisconsin Madison Wisconsin 1968

5 W R UTTAL T PASICH M ROGERS R HIERONYMUS
Generative computer assisted instruction
Communication #243 Mental Health Research Institute University of Michigan Ann Arbor 1969

6 E B HUNT
Selection and reception conditions in grammar and concept learning
J Verbal Learn Verbal Behav Vol 4 pp 211-215 1965

7 E CROTHERS P SUPPES
Experiments in second-language learning
Academic Press New York New York Chapter 6 1967

8 R F SIMMONS
Linguistic analysis of constructed student responses in CAI
Report TNN-86 Computation Center The University of Texas at Austin 1968

Planning for an undergraduate level computer-based science education system that will be responsive to society's needs in the 1970's

by JOHN J. ALLAN, J. J. LAGOWSKI and MARK T. MULLER

The University of Texas at Austin
Austin, Texas

INTRODUCTION

The purpose of this paper is to discuss the planning of an undergraduate level computer-based educational system for the sciences and engineering that will be responsive to society's needs during the 1970's. Considerable curriculum development research is taking place in many institutions for the purpose of increasing the effectiveness of student learning. Despite the efforts under way, only limited amounts of course matter using computer-based techniques are available within the sciences and engineering.

Planning for a frontal attack to achieve increased teaching effectiveness was undertaken by the faculty of The University of Texas at Austin. This paper presents the essence of these faculty efforts.

An incisive analysis of the state of the art with regard to the impact of technology on the educational process is contained in the report "To Improve Learning" generated by the Commission on Instructional Technology and published by the U.S. Government Printing Office, March, 1970.[1] The focus is on the potential use of technology to improve learning from pre-school to graduate school. The goals stated in the above report are (1) to foster, plan, and coordinate vast improvements in the quality of education through the application of new techniques which are feasible in educational technology, and (2) to monitor and coordinate educational resources.

AN OVERVIEW OF COMPUTER-BASED TEACHING SYSTEMS

Until recently, interest in using machine-augmented instruction has been centered primarily on research in the learning processes and/or on the design of hardware and software.

Digital computer systems have now been developed to the point where it is feasible to employ them with relatively large groups of students. As a result, defining the problems involved in the implementation of computer-based teaching techniques to *supplement* classical instructional methods for large classes has become a most important consideration. Whether the classes are large or small, colleges and universities are faced with presenting increasingly sophisticated concepts to continually-expanding numbers of students. Available teaching facilities, both human and technical, are increasing at a less rapid rate than the student population. Typically, the logistics of teaching science and engineering courses becomes a matter of meeting these growing demands by expanding the size of enrollments in lectures and laboratory sections.

It is now apparent that we can no longer afford the luxury of employing teachers in non-teaching functions —whether on the permanent staff or as teaching assistants. Many chores such as grading and record keeping as well as certain remedial or tutorial functions do not really require the active participation of a teacher, yet it is the person hired as a teacher who performs these tasks. Much of this has been said before in various contexts; however, it should be possible to solve some of these problems using computer techniques. In many subjects, there is a body of information that must be *learned* by the student but which requires very little *teaching* by the instructor. Successful methods must be found to shift the onus for learning this type of material onto the student thereby premitting the instructor more time for teaching. Thus, computer techniques should be treated as resources to be drawn upon by the instructor as he deems necessary, much the same as he does with books.

Basically, the application of computer techniques is *supplemental to*, rather than a *supplantation of*, the

human teacher. The average instructor who has had little or no experience with computers may be awed by the way the subject matter of a good instructional module can motivate a student. If the program designer has been imaginative and has a mastery of *both* his subject and the vagaries of programming, the instructional module will reflect this. On the other hand, a pedantic approach to a subject and/or to programming of the subject will also unfortunately be faithfully reflected in the module. Thus, just as it is impossible to improve a textbook by changing the press used to print it, a computer-based instructional system will not generate quality in a curriculum module. Indeed, compared with other media, computer methods can amplify pedagogically poor techniques out of proportion.

The application of computer techniques to assist in teaching or learning can be categorized as follows:

1. *Computer Based Instruction* (CBI)—this connotes interactive special purpose programs that either serve as the sole instructional means for a course or at least present a module of material.
2. *Simulation*
   a. of experiments
      1. that allow "distributions" to be attributed to model parameters
      2. that are hazardous
      3. that are too time consuming
      4. whose equipment is too expensive
      5. whose principles are appropriate to the student's level of competence, but whose techniques are too complex
   b. for comparison of model results with measurements from the corresponding real equipment
3. *Data Manipulation* (can be interpreted as customarily conceived computation) for
   a. time compression/expansion—especially in the laboratory, as in data acquisition and reduction with immediately subsequent "trend" or "concept" display
   b. advanced analysis in which the computation is normally too complex and time consuming
   c. graphic presentation of concepts—possibly deflections under dynamic loading, molecular structures, amplitude ratio and phase lead or lag, . . .
4. *Computer-Based Examination and Administrative Chores* to accompany self-paced instruction.

SYSTEM DESIGN PHILOSOPHY

Design concepts that foster a synergistic relationship between a student and a computer-based educational system are based upon the following tenets:

1. The role of the computer in education is solely that of a tool which can be used by the average instructor in a manner that (a) is easy to learn, (b) is easy to control, and (c) supplements instructional capability to a degree of efficiency unattainable through traditional instructional methods.
2. Computer-based education, although relatively new, has progressed past the stage of a research tool. Pilot and experimental programs have been developed to the point where formal instruction has been conducted in courses such as physics[2] and chemistry.[3,4] Despite this, the full potential of these new techniques has yet to be realized. Future systems, that are yet to be designed, must evolve which will provide sufficient capacity, speed and flexibility. These systems must be able to accommodate new teaching methods, techniques, innovations and materials.

   Programming languages, terminal devices and communications should be so conceived as to not inhibit pedagogical techniques which have been successfully developed over the years. The system should incorporate new requirements which have been dictated through progressive changes in education and adjust without trauma or academic turbulence.
3. Initial computer-based instructional systems should be capable of moderate growth. Their role will be that of a catalyst to expedite training of the faculty as well as a vehicle for early development of useful curriculum matter. Usage by faculty will grow in parallel with evolving plans for future systems based upon extensive test and evaluation of current course matter.

The design of individual instructional modules to supplement laboratory instruction in the sciences and engineering will follow the general elements of the systems approach which has gained acceptance in curricular development.[5] This approach to curriculum design can generally be described as a method for analyzing the values, goals, or policies of human endeavor. The method as applied to computer-assisted instruction has been described in detail by Bunderson.[6]

Although there have been several different descriptions of the systems approach to the design of curricular materials, two major elements are always present: course analysis and profiling techniques. **Course analysis** consists of

1. ***a concise statement*** of the behavioral objectives

of the course expressed in terms of the subject matter that is to be learned.
2. ***the standard*** that each student must reach in the course.
3. ***the constraints*** under which the student is expected to perform (which may involve an evaluation of his entering skills).

The results of the course analysis lead to an implementation of the suggested design by incorporating a curriculum profile ("**profile techniques**"), which contains

1. ***function analysis,*** i.e., the use of analytical factors for measuring the parameters of a task function
2. ***task analysis,***[7] i.e., an analysis that identifies in depth various means or alternative courses of action available for achieving specific results stated in the function analysis
3. ***course synthesis,*** the iterative process for developing specific learning goals within specific modules of instructional material.

A general flow diagram which shows the relationship between the elements in the systems approach to curriculum design appears in Figure 1.

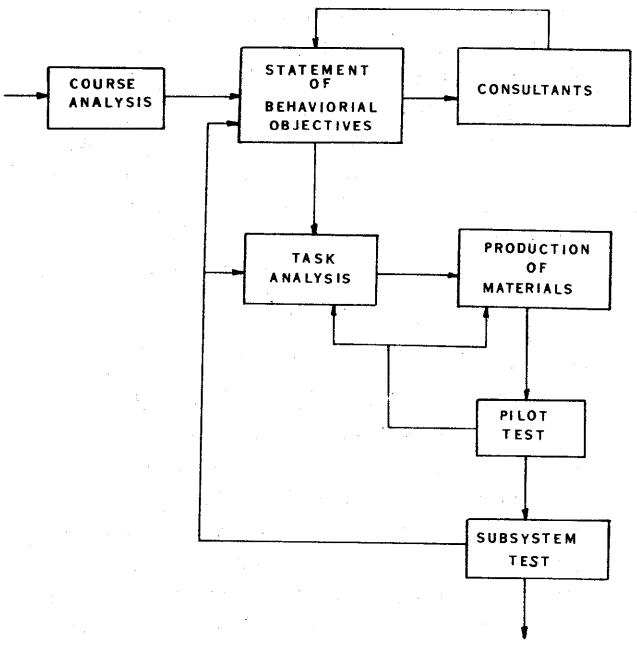


Figure 1—Systems approach to curriculum design

Goals and behavioral objectives

Some of the longer range goals defined should be to demonstrate quantitatively the increased effectiveness gained by computer-based techniques and further, to develop skill in the faculty for "naturally" incorporating information processing into course design. Another long range goal should be to effect real innovation in the notions of "laboratory courses" and finally, to instill in graduating students the appreciation that computers are for far more than "technical algorithm execution."

In more detail, some of the goals are to:

1. Plan, develop and produce computer-based course material for use in undergraduate science and engineering courses to supplement existing or newly proposed courses. Course matter developed will utilize a systems approach and consider entering behaviors, terminal objectives, test and evaluation schema, provisions for revision, and plans for validation in an institutional learner environment.
2. Produce documentation, reports, programs and descriptive literature on course matter developed to include design and instructional strategies, flow charts and common algorithms.
3. Test course matter developed on selected classes or groups for the purpose of evaluation, revision and validation. This is to determine individual learner characteristics through use of techniques such as item analysis, and latency response using interactive languages.
4. Promote and implement an in-depth faculty involvement and competency as to the nature, application and use of computer-based instructional languages, course matter and techniques.
5. Compile, produce and make provisions for mass distribution of such computer-based materials as are tested, validated and accepted for use by other colleges or institutions participating in similar programs.

In an academic environment the use of time-sharing computers for education is gradually being incorporated as a hard-core element, which in time will become so commonplace a resource for faculty and student use that it will serve as an "educational utility" on an interdisciplinary basis. To achieve a high degree of effectiveness of such systems, the faculty using computer-based teaching techniques as a supplement to laboratory type instruction must initially become involved. The pedagogical objectives of this whole approach are:

1. To correct a logistic imbalance: i.e., a condition marked by the lack of a qualified instructor and/

or the proper equipment and facilities to perform a specific teaching task (or project) being at a specific place at a specific time.
2. To provide more information per unit time so that, as time passes, the course will provide inincreasingly in-depth knowledge.
3. To allow new ranges of material to be covered when one specifically considers things currently omitted from the curriculum because of student safety or curriculum facility costs.
4. To increase academic cost effectiveness, because it is certainly expected that adroit information processing will free the faculty from many mundane tasks.
5. To provide both parties with more personal time and flexibility, because it is anticipated that considerable amounts of time are to be made free for both student and faculty.
6. To make a computer-based system the key to the individualization of mass instruction by utilizing dial-up (utility) techniques.
7. To develop laboratory simulation so that it is no longer a constriction in the educational pipeline because of limited hardware and current laboratory logistics.

Evaluation criteria

In general, there are two distinct phases to the evaluation of instructional modules. The first of these coincides with the actual authoring and coding of the materials, and the second takes place after the program has advanced to its penultimate stage. These two types of evaluation can be referred to as "developmental testing" and "validation testing," respectively.

Developmental testing

This testing is informal and clinical in nature and involves both large and small segments of the final program. The specific objective of this phase of the development is to locate and correct inadequacies in the module. It is particularly desirable at this point of development to verify the suitability of the criteria which are provided at the various decision points in the program. It is also anticipated that the program's feedback to the students can be improved by the addition of a greater variety of responses. Finally, testing at this stage should help to uncover ambiguous or incomplete portions of the instructional materials.

Relatively few students (about 25) will be required at this stage of evaluation; however, since this phase is evolutionary, the exact number will depend on the nature and extent of the changes that are to be, and have been, made. Materials which are rewritten to improve clarity, for example, must be retested on a small scale to establish whether an improvement has in fact been made. A final form of this program, incorporating the revisions based on this preliminary testing, will be prepared for use by a larger group of students.

Validation testing

The formal evaluation of the programs will occur in a section (or with a part of a section) of a regularly scheduled class.

It is assumed that a selection of students can be obtained that is representative of the target population for which the materials are designed. One of the following two methods for obtaining experimental and control groups is suggested, depending upon circumstances existing at the time of the study (i.e., number of sections available, whether they are taught by the same instructor, the willingness of instructor to cooperate, section size, etc.).

The preferred method is to arbitrarily designate one course section *experimental* and one course section *control* with the same instructor teaching both sections. An assumption here is that the registration procedure results in a random placement of students in the sections. The alternative method of selecting students follows. The instructional programs are explained to the total student group early in the semester, and a listing of those students who are willing to participate is obtained. Two samples of approximately equal size are randomly selected from this list. One sample of students is then assigned to work with the computer-assisted instructional facilities and is designated as the experimental group.

The criteria used to evaluate the programs are as follows:

1. The extent to which students engage in and/or attain the behavioral objectives stated for the program. For tutorial and remedial programs pre- and post-tests are the instruments for measuring attainment and will help answer the question: Does the computer-based supplement actually teach what it purports to teach? For experiment simulations, the successful completion of the program is *prima facie* evidence that the student has engaged in the desired behaviors.
2. Achievement as measured by the course final examination.

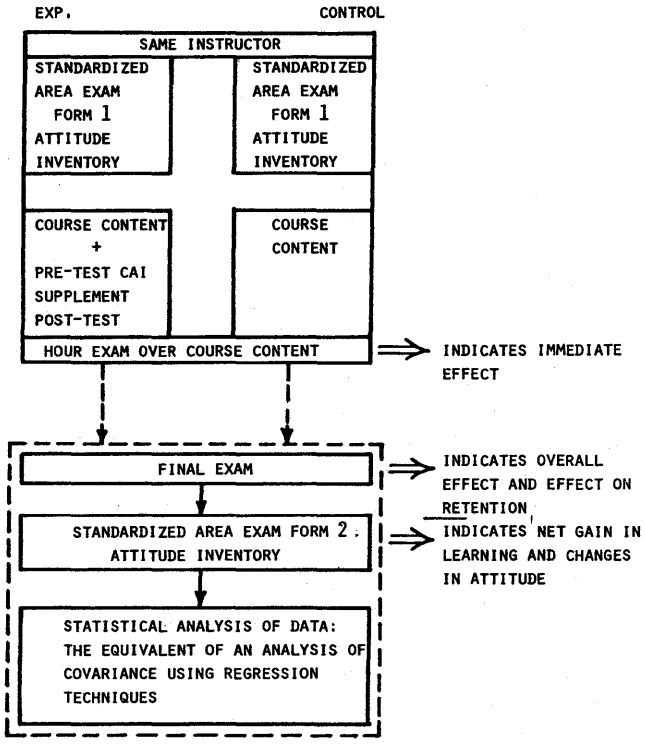


Figure 2—Test and evaluation schema for overall program

3. Net gain in achievement as determined by pre- and post-testing with the appropriate standardized area examination.
4. Changes in student attitudes as measured by pre- and post-attitude inventories.

The statistical treatment used to evaluate the programs in light of the above criteria will be a multiple linear regression technique equivalent to analysis of covariance for criteria 2 and 3. The covariables used will include, (1) A placement exam score (if available); (2) the beginning course grade (for students in advanced courses); (3) sex; (4) SAT* scores, high school GPA**; and (5) other pertinent information available in the various courses. The key elements of the test and evaluation schema are shown schematically in Figure 2.

IMPLEMENTATION

Planning is one of the most important aspects of CBI and when done properly yields what we might call a "systems approach" to curriculum or course design. This means not only the setting up of a course of instruction by following a comprehensive, orderly set of steps, but also a plan for involving other faculty.

No two curriculum designers will agree as to what constitutes the essentials in outlining a course in its entirety before beginning. However, there are certain sequential or logical steps that can be defined precisely. These steps have been designated below as the course planning "mold" and are based upon actual in-house experience in planning, developing, testing and evaluation of course matter. Examples of this planning as shown in the entries in Tables I and II are authentic

TABLE I—Curriculum Development Research Tasks

Task Title	Type Application	Research Investigator	Department
Machine Element Design	D, IG, MM, PrS, Res	Dr. J.J.A. Assistant Professor	Mech. Engr.
Aerospace Structural	D, IG, MM, PrS, Sim, Stat, Res	Dr. E.H.B. Assistant Professor	Aero. Engr.
Theoretical Chemistry	Sim, D, Rsim	Dr. F.A.M. Professor Dr. R.W.K. Faculty Associate	Chemistry
Biophysical Analysis	SIM, Rsim, Stat	Dr. J.L.F. Assistant Professor	Zoology

Application Code

D—Drill and Practice; G—Graphics; IG—Interactive Graphics; MM—Mathematical Modeling, Gaming; PrS—Problem Solving; OLME—On Line Monitoring of Experiments; Sim—Simulation; Rsim—Real Experiments Plus Simulation; Stat—Statistical Data Processing; Res—Research in Learning Processes.

TABLE II—Curriculum Development Research Task Cost

Task Title	Duration (Mo.)	Total Dollar Value*	Personnel Cost	Computer Time Cost
Elements of Design	27	$72,440.	$31,120.	$41,320.
Aerospace Structures	31	$111,720.	$45,920.	$65,800.
Theoretical Chemistry	24	$53,150.	$36,900.	$16,250.
Biophysical Analysis	19	$43,250.	$23,600.	$19,650.

* Not including hardware cost.

* Scholastic Aptitude Test.
** Grade Point Average.

and have been derived by requesting each new associate investigator to fit the approach to beginning his work into a "mold." Descriptive material for each prospective participating faculty member is shown in the following example,* and is organized as follows:

1. Title
2. Introduction
3. Objectives
4. Project Description
5. Method of Evaluation
6. Impact on the Curriculum
7. Pedagogical Strategy
8. Current or Past Experience
9. Related Research
10. Plans for Information Interchange
11. Detailed Budget*
12. Time Phasing Diagram
13. Application Cross-Reference Chart

EXAMPLE OF A PROSPECTIVE ASSOCIATE INVESTIGATOR'S PROPOSAL:

INTERACTIVE DESIGN OF MACHINE ELEMENT SYSTEMS

Introduction

Engineering design is a structured information/creative process. The synthesis of physically feasible solutions generally follows many design decisions based on processed available information. The structure of design information has been defined in the literature.[8] Processing this design information in a scientific and logical method is important to the understanding of how one teaches engineering design, particularly the part that computer augmentation might play in the process. Essentially, the computer can relieve the engineering designer of those aspects of problem manipulation and repetitive computation that might otherwise distract him from the context of his problem. The fundamental principle here is to have the designer making decisions, not calculations.

Objectives

There are several objectives to being able to put a student in an interactive situation for the design of machine elements. One of the primary goals is to allow students to consider machine element systems instead of the traditional approach of the machine elements. The object of this is to show the interplay of the deflections and stresses due to imposed forces on machine element systems. Another objective is to allow distributions to be attributed to parameters of systems normally considered as discrete. Latest in the series of engineering texts are those that consider probablistic approaches to design. Finally in the design process, by definition, a task performed at one instant in time does not give the observer of a designer any information about what the designer's next task might be. Hence, it is most important to construct an environment that will allow the student to "meander" through the myriad of possibilities of alterations of a nonfeasible solution of a design problem that might make it subsequently a feasible solution. Another objective of using an interactive system is so considerations not necessarily sequential in an algorithmic sense can be superposed for the student's consideration. That is, he can consider system reliability, system structural integrity, and system economics simultaneously and make design decisions based on any one of those fundamental considerations.

Project description

This project involves the integration into a conventional classroom of one or more reactive CRT displays. The addition of computer augmentation to the traditional teaching of engineering element systems design is proposed, in order to satisfy the needs of the undergraduate student who would like to be able to synthesize and analyze more realistic element systems than is now pedagogically possible. The following specific desired functions in the computer-based system are necessary to make it an easily accessible extension to the student designer's functional capabilities:

1. Information storage and retrieval—The system must be able to accept, store, recall, and present those kinds of information relevant to the designer. For example, material properties, algorithms for analyzing physical elements, conversion factors, etc.
2. Information Interchange—The system must be able to interact with the designer conveniently so that he can convey information with a minimum of excess information during the operation of the system.[8]
3. Technical Analysis Capabilities—The system must be able to act on the information pre-

* It should be emphasized that the courses listed in Tables I and II represent only a portion of the curriculum matter which is being or needs to be developed.

* Sample forms for items 11, 12, and 13 can be obtained from the authors. The form for item 11 in particular gives the details necessary to arrive at a project's estimated dollar cost.

sented, analyze it, and present concise results to the designer.
4. Teaching and Guidance Capabilities—The system must provide the student designer with information necessary to enable him to use the system and also to tutor him in the context of his problems.
5. Easy Availability—The system should be available to the student user and adaptable to his modifications.

In essence this project can be described as the design and implementation of a computer-based reactive element design system which would be as integral a part of a teaching environment as the blackboard, doors and windows. Further, in this project the amount of additional material in terms of reliability, value analysis, noise control and other modern engineering concepts that can be integrated into the standard curriculum can be conveyed more effectively and to a deeper degree.

Method of evaluation

The effectiveness of this method will be evaluated by presenting the students with more comprehensive design problems at the end of the semester. Several observations would be made. First, they would be able to take into account more readily the interaction between systems elements in a mechanical network. They should not require as many approximations with respect to fatigue life, method of lubricating bearings, etc.

Impact on the curriculum

The effect on the curriculum will be as follows. The Mechanical Engineering Department at The University of Texas has what is known as "block" options. In this way, a student in his upper-class years may select some particular concentrated area of study. Part of the material that he now receives in one of his block courses 366N could be moved back to his senior design course 366K. The effects on the curriculum should be to free, for additional material, one-half semester of a three-hour course at the senior undergraduate level for those majoring in design.

Pedagogical strategy

The strategy of the course at present is the solution of authentic design problems solicited from industry. However, all analysis techniques are currently confined to paper and pencil with computer algorithms where practical. The new strategy will add the ability to analyze mechanical systems networks on an interactive screen and have an opportunity to manipulate many possible ideas (both changes in topology and changes in parameters) per unit time.

Current or past experience

The investigator has spent several years on the development of interactive graphic machine design systems both in industry and in academic institutions. Further, the investigator has been teaching the design of machine element systems since 1963 and prior to that practiced design of machine element systems since 1958.

Related research

Professor Frank Westervelt at the University of Michigan is currently conducting research in the area of interactive graphic design. His major emphasis is on the design of the executive systems behind interactive computer graphic systems. Dr. Milton Chace also at the University of Michigan uses some software generated by Dr. Westervelt's group. His concentration is on using Lagrange's method to describe dynamic systems and be able to manipulate the problem's hardware configurations on the screen. Professor Daniel Roos of the MIT Urban Systems Laboratory is now putting a graphic front end on his ICES system. Professor Gear at the University of Illinois has a system running on a 338/360-67 system and his primary interest there is to work with people who are manipulating electrical network topologies and doing network analysis. Professor Garrett at Purdue University is using graphic terminal access to his university's computer for performing a series of analyses on mechanical engineering components.

The unique aspect of the work that is envisioned here, as opposed to all above mentioned relative works, is primarily that it will be an instructional tool. That is, the computer is recognized as an analysis tool, a teacher, and an information storage and retrieval device.

With respect to each of the above, there are two connotations. With respect to analysis, the computer will perform analysis in its traditional sense of engineering computation and it will also analyze the topology of mechanical systems as drawn on the screen such that the excess information between the designer and the computer will be reduced. Concerning teaching,

the computer will work with the student not only to teach him how to use the system, but also in another sense it will help him learn about his problem's context by techniques such as unit matching and unit conversion and checking topological continuity before allowing engineering computations. With respect to information storage and retrieval, traditional parameters such as material yield strengths and physical properties of fluids will be available.

Another concept in interactive design will be developed when the system can work with the student and help him optimize apparently continuous systems made up of objects such as bearings that only come in discrete quantities, for instance $\frac{1}{8}$ inch increment bores, etc.

Plans for information interchange

Information interchange, of course, has two meanings: (1) how to get information from other schools to the University of Texas, and (2) how to disseminate the results of my research to other interested parties. It is the responsibility of the principal investigator to insure that information from professional society meetings, personal visits to other campuses, research reports, published articles, personal communication, and "editorial columns" in both regional and national newsletters of societies such as ASME, IEEE, SID, SCI, ACM, and others are disseminated to his research associates.

Further, it is the researcher's responsibility to generate research reports, published articles, take the responsibility for keeping his latest information in "news items," attend meetings, and write personal letters to other people in the field who are interested to help keep them abreast of his current research activity.

EXPERIENCES TO DATE

Completed projects

During the academic year 1968–69 two pilot projects were conducted on the use of computer-based techniques in the instruction of undergraduate chemistry. Fifteen curriculum modules were written for the General Chemistry project and eight for Organic Chemistry. In each instance the modules covered subjects typical of the material found in the first semester of each course. Each module was "debugged" and polished by extensive use of student volunteers prior to use in the pilot studies. In each study, the modules were used to supplement one section of the respective courses under controlled conditions. Results from this two-semester study on computer-based applications in organic chemistry indicate a high degree of effectiveness. Not only was the study effective in terms of student score performance, but also in terms of outstandingly favorable attitudes towards computer-based instruction by students and professors.

Concurrently, faculty members of the College of Engineering have also conducted various courses using computer-based techniques. Dr. C. H. Roth of the Electrical Engineering Department has developed a computer-assisted instructional system that simulates a physical system and provides computer-generated dynamic graphic displays of the system variables.[9] An SDS 930 computer with a display scope is used for this project. The principal means of presenting material to the students is via means of visual displays on the computer output scope. A modified tape recorder provides audio messages to the student while he watches the scope displays. A flexible instructional logic enables the presentation of visual and audio material to the student and allows him to perform simulated experiments, to answer questions (based on the outcome of these experiments) and provides for branching and help sequences as required. Instructional programming is done in FORTRAN to facilitate an interchange of programs with other schools.

Dr. H. F. Rase, et al.,[10] of the Chemical Engineering Department have developed a visual interactive display for teaching process design using simulation techniques to calculate rates of reaction, temperature profiles and material balance in fixed bed catalytic reactors.

Other computer-based instructional programs developed include similar modules in petroleum engineering and mechanical engineering. In each instance, the curriculum modules are used to supplement one or more sections of formal courses. In several instances it is possible to measure the course effectiveness against a "control" group. Course modules generally contain simulated experiments, analysis techniques using computer graphics, and some tutorial material.

The general objectives for all of the aforementioned studies are as follows:

1. To give the students an opportunity to have simulated laboratory experiences which would otherwise be pedagogically impossible.
2. To provide the students with supplemental material in areas which experience has shown to be difficult for many students.
3. To provide a setting in which the student is allowed to do a job that professionals do; i.e., collect, manipulate and interpret data.
4. To give the student the feeling that a real, con-

cerned personality has been "captured" in the computer to assist him.
5. To individualize the student-computer interactions as much as possible by allowing the student to pace himself.
6. To give the student a one-to-one situation in which he can receive study aid.

Present state of computer-based learning systems

Curriculum development in engineering and the sciences is being accomplished within the University of Texas through the Computation Center CDC-6600 system using conversational FORTRAN within the RESPOND time-sharing system. Also, small satellite pre- post-processors such as a NOVA, a SIGMA 5 and an SDS-930 are linked to the CDC-6600. Of special significance is the Southwest Regional Computer Network (SRCN) which is now in the early stages of use and is built around the CDC-6600. Some eight intrastate colleges and universities are linked. Workshop and user orientation sessions are still under way concurrent with curriculum development. The integration of course matter for this network will be accomplished during the next two to three years. A discussion of the factors involved in curriculum software development and some evaluation aspects follows.

Cost factors and management considerations

When developing an accounting system for examining the cost of generating software for teaching, one readily realizes that there is the traditional coding and the subsequent assembly and listing. However, because academic institutions frequently have this class of endeavor funded from agency sources, a next step is very frequently hardware installation (if the terminals or the central processor have not been "on board" prior to the initiation of the research). Once the hardware is available, however, on-line debugging can take place and the first course iteration can begin. The next step is the first course revision in which the material is rewritten, possibly restructured, to take advantage of experiences on the first pass. Then, the second course iteration is performed and finally the second course revision.

The estimates for coding, assembly, listing, and debug time, etc., are a function of

1. the computer operating system and its constraints.
2. the coding proficiency of the professionals involved.
3. the way in which the computer is supposed to manifest itself, that is:
   a. as a simulation device in parallel with the real experiment,
   b. as purely a simulation device,
   c. as a base for computer-augmented design,
   d. for data acquisition,
   e. for computer-based quizzes,
   f. for computer-based tutorial and drill,
   g. or finally, for computation.
4. the degree of reaction required, that is in the physiological sense, how the computer must interact for the user's latent response.
5. the extent of the data base required and the data structure required to allow one to manipulate that data base.

The primary considerations are organization and control. In this work the organization consists of the following. At each college there is a project coordinator who is the line supervisor for the secretarial services, programming services, and technical implementation services, and who acts as the coordinator for consultants.

At the University level there exists a review panel, composed of members of the representative colleges and several people from other areas such as educational psychology and computing, which evaluates the works that have been conducted in any six-month period and also evaluates the request for continuation of funds to continue a project to its duration. The actual fiscal control of each project is with the project coordinator of that particular college. Also, the purchase of equipment is coordinated at the college level.

The control as expressed above is basically a two-step control; a bi-annual project review, plus a budgetary analysis, is accomplished by an impartial review panel. Their function is to act as a check on the context of the material presented and to recommend continuance of particular research tasks. As a further step, this research is coordinated in all colleges by the Research Center for Curriculum Development in Sciences and Mathematics.

SUMMARY

Dissemination of information

Dissemination of information is planned through (1) documentation of publications in the form of articles, reports or letters, (2) symposia to be held which cover procedures and fundamentals for developing curriculum in CBI and (3) furnishing of completed packages of

course matter on computer tapes, cards or disks to other colleges or institutions desiring, and able to use, this material. Wide publicity will be given completed course matter through such agencies as ERIC, ENTELEK and EDUCOM. The provision for continuing periodic inputs to the above agencies will provide for current availability of curriculum materials.

Future outlook

The future outlook for time-sharing in computer-based educational systems is extremely bright with respect to hardware development. The advent of the marketing of newer mini-computer models selling for five to ten thousand dollars—some complete with a limited amount of software—is already changing the laboratory scene. The configuration of direct-connect terminals, with the inherent advantage of installation within one building, further enhances the use of this type of system by eliminating the high expense of data lines and telephones.

Software in the form of a powerful CBI language for the sciences and engineering designed specifically for minicomputers is perhaps one of the most important needs. Rapid progress has been made in developing a low cost terminal using plasma display or photochromic technology, however, the promise of a low cost terminal has yet to be realized.

A small college, high school or other institution may be able to afford a small time-sharing computer system with ten terminals that could meet its academic needs for less than $36,000; however, still lacking is the vast amount of adequate curriculum matter. Educators using CBI are in the same position as a librarian who has a beautiful building containing a vast array of shelves but few books to meet academic needs of students or faculty. The task of curriculum development parallels the above situation and must be undertaken in much the same manner as any other permanent educational resource.

Educational benefits

Benefits that result from implementing this type plan are a function of the synergistic interplay of (1) personnel with expertise in the application of computer-based techniques, (2) computer hardware including interactive terminal capability, (3) faculty-developed curriculum materials, and (4) the common information base into which the entire research program can be embedded.

The program can provide students with a learning resource that serves many purposes. That is, the computer can be the base of a utility from which the user can readily extract only that which he needs, be it computation, data acquisition, laboratory simulation, remedial review, or examination administration. At all times the computer can simultaneously serve as an educator's data collection device to monitor student interaction. This modularized dial-up capability can give the students an extremely flexible access to many time-effective interfaces to knowledge.

Administrative benefits

When this type project has been completed, all users may have access to the results. This unified approach can yield modules of information on cost accounting which can be used as a yardstick by the University.

Further, this type project insures that data-yielding performance is of a consistently high quality, and that regardless of the subject matter on any research task the depth of pedagogical quality is relatively uniform.

The subject matter developed can serve as the basis for conducting experiments in teaching and continuing curricular reform. Indeed, the association of faculty in diverse disciplines can serve as a catalyst for curricular innovations in the course of preparing materials for computer-based teaching techniques.

The instructional efforts described here can serve as the basis for displaying the unity of science and engineering to the undergraduate student in an indirect way. For example, if a student is taking a set of science courses in a particular sequence because it is assumed each course contributes to an understanding of the next one, it is possible to use programs developed in one course for review of remedial work in the next higher course. Thus, an instructor in biology can assume a certain level of competence in chemistry without having to review the important points in the latter area. Should some students be deficient in those aspects of chemistry that are important to the biology course, the instructor can assign the suitable material that had been developed for practice and drill *for the preceding chemistry course.* With a little imagination, the instructional system can make a significant impact on the student by giving him a unified view of science or enginering. It is possible to develop both a vertical and horizontal structure for common courses which can be used on an interdisciplinary basis for integration in the basic core curriculum in the various departments where computer-based techniques are used. The revision problem of inserting new and deleting old material in such a system is considerably simplified for all concerned.

There is a vast, largely unexplored area of applications. As time-sharing methods become more widespread, terminal hardware becomes less complex, and teleprocessing techniques are improved, the potential usefulness of computers in the educational process will increase. With the technology and hardware already in existence, it is possible to build a computer network linking industries and universities. Such a network would (1) allow scientists and engineers in industry to explore advanced curriculum materials, (2) allow those who have become technically obsolete to have access to current, specifically prepared curriculum materials, training aids and diagnostic materials, (3) allow local curriculum development as well as the ability to utilize curriculum materials developed at the university, (4) allow engineers to continue their education, (5) provide administrators with an efficient and time-saving method of scheduling employees and handling other data processing chores such as inventories, attendance records, etc., and (6) provide industrial personnel with easily obtainable and current student records to aid in giving the student pertinent and helpful counseling and guidance.

Although complaints have been voiced that computer-based techniques involve the dehumanization of teaching, we argue to the contrary—the judicious use of these methods will individualize instruction for the student, help provide the student with pertinent guidance based upon current diagnostic materials and other data, and allow instructors to be teachers.

ACKNOWLEDGMENTS

Special thanks are due Dr. Sam J. Castleberry, Dr. George H. Culp, and Dr. L. O. Morgan (Director), of the Research Center for Curriculum Development in Science and Mathematics, The University of Texas at Austin. Also, appreciation is extended to numerous colleagues who have contributed to our approaches with their ideas.

REFERENCES

1 *To improve learning*
US Government Printing Office March 1970

2 N HANSEN
Learning outcomes of a computer-based multimedia introductory physics course
Semiannual Progress Report Florida State University Tallahassee Florida 1967 p 95

3 S CASTLEBERRY J LAGOWSKI
Individualized instruction in chemistry using computer techniques
J Chem Ed 47 pp 91-97 February 1970

4 L RODEWALD et al
The use of computers in the instruction of organic chemistry
J Chem Ed 47 pp 134-136 February 1970

5 R E LAVE JR D W KYLE
The application of systems analysis to educational planning
Comparative Education Review 12 1 39 1968

6 C V BUNDERSON
Computer-assisted instruction testing and guidance
W H Holtzman Ed Harper & Row New York New York 1970

7 R GAGNE
The analysis of instructional objectives for the design
In Teaching Machines and Programmed Learning II The National Education Association Washington DC 1965 pp 21-65

8 J J ALLAN
Man-computer synergism for decision making in the system design process
CONCOMP Project Technical Report #9 University of Michigan July 1968

9 E A REINHARD C H ROTH
A computer-aided instructional system for transmission line simulation
Technical Report No 51 Electronics Research Center The University of Texas Austin 1968

10 H F RASE T JUUL-DAM J D LAWSON L A MADDOX
The use of visual interactive display in process design
Journal Chem Eng Educ Fall 1970

The telecommunications equipment market—Public policy and the 1970's

by MANLEY R. IRWIN

University of New Hampshire
Durham, New Hampshire

INTRODUCTION

The growing interdependence of computers and communications, generally identified with developments in digital transmission and remote data processing services, has not only broadened the market potential for telecommunication equipment but has posed several important public policy issues as well. It is the purpose of this paper to explore the relationship between the telecommunications equipment market and U.S. telecommunication policy. To this end we will first survey the traditional pattern of supplying hardware and equipment within the communications common carrier industry and second, identify recent challenges to that industry's conduct and practices. We will conclude that public policy holds a key variable in promoting competitive access to the telecommunication equipment market—access that will redound to the benefit of equipment suppliers, communication carriers, and ultimately the general public.

THE COMMUNICATION COMMON CARRIER

As a prelude to our discussion it is useful to identify the major communications carriers in the United States. The largest U.S. carrier, AT&T, provides upwards to 90 percent of all toll or long distance telephone service in the country. In addition to its long line division, AT&T includes some 24 telephone operating companies throughout the U.S.; the Bell Telephone Laboratory, the research arm of the system; and Western Electric, the manufacturing and supply agent of the system.[1] These entities make up what is commonly known as the Bell System.

The non-Bell telephone companies, include some 1800 firms scattered throughout the U.S. These firms render service in 85 percent of the geographical sector of the country and account for 15 percent of the remaining telephones in the U.S. The independents are substantially smaller than AT&T and in decreasing size include the General Telephone and Electronics System, United Utilities, Central Telephone Company and Continental Telephone respectively. Although General is by far the largest, the independents have experienced, over the past two decades, corporate merger and consolidation.

Western Union Telegraph Company provides the nation's message telegraph service, the familiar telegram and Telex, a switched teletypewriter service. Recently, Western Union has negotiated with AT&T to purchase TWX thus placing a unified switched network under the single ownership of the telegraph company.[2] In addition to their switched services, the carriers provide leased services to subscribers on a dedicated or private basis. In this area, both Western Union and AT&T find themselves offering overlapping or competitive services.

The carriers, franchised to render voice or non-voice service to the general public, reside in an environment of regulation. Licenses of convenience and necessity are secured from either Federal or state regulatory bodies and carry with it a dual set of privileges and responsibilities. In terms of the former, the carriers are assigned exclusive territories in which to render telephone or telegraph services to the public at large—a grant tendered on the assumption that competition is inefficient, wasteful, and unworkable. In terms of the latter, the carriers must serve all users at non-discriminatory rates, submitting expenses, costs and revenue requirements for public scrutiny and overview. In the United States, the insistence that utility firms be subject to public regulation rests on the premise that economic incentives and public control are neither incompatible nor mutually exclusive.

THE POLICIES AND PRACTICES OF THE CARRIER

Given the carriers and their environmental setting, three traits have tended to distinguish the communication industry in the past. These include, first, a practice of owning and leasing equipment to subscribers; second, the policy of holding ownership interest in equipment suppliers and manufacturers; and finally, a practice of taking equipment and hardware requirements from their supply affiliates. Each of these policies has conditioned the structure of telecommunication equipment for several decades and thus merits our examination.

Tariffs

In the past at least the carriers provided what they term a total communication service. That service embraced loops or paired wires, switching equipment, interoffice exchange trunks and terminal or station equipment. Prior to 1968, subscribers were prohibited from linking their own telephone equipment to the telephone dial network. This policy was subsumed under a tariff generally termed the foreign attachment tariff—the term "foreign" in reference to equipment not owned or leased by the telephone company. Users who persisted in attaching equipment not supplied by the carrier incurred the risk of service disconnection.

Carrier non-interconnection policy extended to private communication systems as well as user equipment. In the former case, the denial of interconnection rested on the carriers apprehension that customers would tap profitable markets. Accordingly, customer interconnection would lead to profit cream skimming, loss of revenues to carriers, and ultimately require the telephone company to increase rates to the general public or postpone rate reductions.

Interconnection was also said to pose problems of maintenance for the utility. With equipment owned partly by subscribers and owned partly by utility, who would assume the burden of service and who would coordinate the introduction of new equipment and new facilities? Whether these problems were real or imaginary, public policy sanctioned carrier ownership and control of related terminal equipment under the presumption that divided ownership would compromise the systemic integrity of a complex and highly sophisticated telephone network.

Telephone subscribers had little choice but to adjust to the foreign attachment prohibition. This meant that of necessity equipment choices were restricted to hardware supplied and leased by the carrier. Competitive substitutes were by definition minimal or nonexistent. In fact the carrier's policy of scrapping old equipment removed a potential used market from possible competition with existing telephone equipment.

Integration

In addition to certain practices embodied as filed tariffs, the carriers owned manufacturing subsidiaries. The integration or common ownership of utility and supplier thus marked a second characteristic of the industry. Obviously Western Electric, the Bell System's affiliate dominated the industry, and over the years, accounted for some 84 to 90 percent of the market. General Telephone's supply affiliates, acquired since 1950, accounted for some 8 percent of the market. Together the two firms approached what economists term a duopoly market; that is, two firms supplying in excess of 90 percent of the market for telecommunication equipment.[3]

Despite the persistence of integration, the efficacy of vertical integration experienced periodic review. In 1949, for example, the Justice Department filed an antitrust suit to divest Western Electric from the Bell System.[4] The suit premised on a restoration of competition to the hardware market, asserted that the equipment market could grow and evolve under conditions of market entry and market access. In 1956, a consent decree permitted AT&T to retain Western Electric as its wholly owned affiliate on the apparent assumption that divestiture served as an inappropriate means to achieve the goal of competition.[5] Instead, market access was to be achieved by making available Bell's patents on a royalty free basis.

Still later the Justice Department embarked on another antitrust suit. This time the antitrust division challenged General Telephone's acquisition of a West Coast independent telephone company on grounds that the merger would tend to substantially lessen competition in the equipment market.[6] The General suit felt the weight of the Bell Consent Decree. So heavy in fact was the Bell precedent that the Department cited the 1956 Decree as grounds for dropping its opposition to General Telephone's merger.[7]

Procurement

A third practice inevitably followed the carrier's vertical relationship; namely the tendency to take the bulk of their equipment from their own manufacturing affiliates. Perhaps such buying practices were inevitable.

Certainly, in the carriers' judgment, the price and quality of equipment manufactured in-house was clearly superior to hardware supplied by independent or nonintegrated suppliers.

Indeed, the courts formalized the procedure of determining price reasonableness by insisting that the carrier rather than the regulatory agency assume the burden of proof in the matter.[8] The result saw the arbitor of efficiency pass from the market to the dominant firm. Over the years the integrated supplier firm has accorded itself rave reviews. Consultant's under carrier contract repeated those reviews. But under the existing rules of the game, one would have hardly expected the firm to act differently.

However long standing, the triad of tariffs, integration and procurement has held obvious implications for independent suppliers of equipment and apparatus. First, non-integrated suppliers found it difficult to sell equipment to telephone subscribers given the enforcement of the foreign attachment tariff. Second, the non-integrated supplier was not particularly successful in selling directly to integrated operating companies. No policy insisted that arms-length buying be inserted between the utility and its hardware affiliate, and indeed the integrated affiliate assumed the role as purchasing agent for its member telephone carriers. Little surprise then that the percentage of the market penetrated by independent firms has tended to remain almost constant for some forty years.

Having said this it must be noted that the carriers insisted that the quality, price and delivery time of equipment from their in-house suppliers was clearly superior to alternative sources of supply. It was as if a competitive market was, by definition less, efficient in allocating goods and services. The private judgment of management was never put to an objective test. Indeed, the carriers resisted formal buying procedures as unduly cumbersome and unwieldy.[9] That resistance has tended to carry the day.

Third, vertical integration skirted the problem of potential abuse inherent in investment rate-base padding. Utilities, for example, operate on a cost plus basis, i.e., they are entitled to earn a reasonable return on their depreciated capital—a derivation of profits that stands as the antithesis of the role of profits under competition. Vertical integration compounded utility profit making by providing an incentive to transplant cost plus pricing to the equipment market. Certainly, the penalty for engaging in utility pricing was difficult to identify much less discipline. The affiliate occupied the best of all possible worlds.

In all fairness one must note the institutional constraint erected to prevent corporate inefficiency on the equipment side. The argument ran that the regulatory agency monitored the investment decision of the carrier. By scrutinizing the pricing practices of the integrated affiliate indirect regulation prevented exorbitant costs on the supply side from entering the utility rate base and passed forward to the subscriber. Indirect regulation allegedly protected both the public and the utility.

As an abstract matter, indirect regulation held an element of appeal. Translating that theory into practice was obviously another matter. On the federal level, the FCC has never found an occasion to disallow prices billed by Western Electric to AT&T.[10] Yet, 65 percent of AT&T's rate base consists of equipment purchased in-house and absent armslength bargaining.[11]

Finally, vertical integration placed the independent equipment supplier in an awkward if not untenable position. As noted, the independent firm could secure equipment subcontracts from its integrated counterpart. The dollar value of those subcontracts was not unimportant. The problem was that the non-integrated supplier was still dependent upon its potential competitor—a competitor who exercised the discretion to make or buy. Little wonder then, that the viability of independent equipment suppliers was controlled and circumscribed by the tariffs, structure, and procurement practices of telephone utilities. Patently, no market existed for the outside firms; and without a market, the base for technical research and development, much less the incentive for product innovation, was effectively constrained, not to say anesthetized. Access to the telecommunication equipment was, in short, limited to suppliers holding a captive market. The government asked the monopoly firm to judge itself; and after dispassionate inquiry the firm equated the public interest with preservation of its monopoly position.

MARKET CHANGES AND PUBLIC POLICY

Market changes

All this has been common knowledge for decades. Whatever the pressures for reassessment and change, those pressures were sporadic, ad hoc and often inconsistent. Today, however, the telecommunications industry is experiencing a set of pressures that marks a point of departure from the triad of policies described above. In a word, the pace of technology is challenging the status quo. More specifically, new customized services tend to be differentiated from services rendered by the common carriers. Time sharing and remote computer based services, for example, provide and impetus for a host of specialized information ser-

vices. The rise of facsimile and hybrid EDP/communication services is equally significant as a trend toward specialization and sub-market development. Subscribers themselves, driven by the imperatives of digital technology, seek flexibility and diversity in their communication facilities. Segmentation and specialization is gaining momentum.

At the same time, the telecommunication hardware market is experiencing a proliferation of new products that pose as substitutes to carrier provided equipment. In the terminal market, for example, modems, multiplexors, concentrators, teleprinters, CRT display units are almost a daily occurrence. Indeed, the computer itself now functions as a remote outstation terminal; and many go so far as to assert that the TV set holds the potential as the ultimate terminal in the home and the school.

The carriers, of course, have not stood idly by. In the terminal market, touch-tone hand sets, picturephones and new teletypewriters signal an intent to participate in remote access input output devices as well. But the point remains—carrier hardware no longer stands alone. The proliferation of competitive substitutes and the potential rise of competitive firms is now a process rather than an event.

The same technological phenomena is occurring in the transmission and switching area as well. Cable TV provides a broad link directly to the home, the school, or the business firm. Satellite transmission poses as a substitute for toll trunking facilities and the FCC has recently licensed MCI as a new customized microwave carrier. Furthermore, carrier switching technology is challenged by hardware manufactured and supplied by firms in the computer industry.

All of this suggests that carrier affiliates no longer possess the sole expertise in the fundamental components that make up telecommunication network and services. It is perhaps inevitable then, that the growing tension between the existing and the possible has surfaced as questions of public policy. These questions turn once again on matters of tariff, procurement and integration.

Public policy decisions

Tariffs

Undoubtedly, the FCC's 1968 Carterphone Decision marks one significant change in the telecommunication equipment industry.[12] Here the Commission held that users could attach equipment to the toll telephone network. Indeed, the Commission insisted that carriers establish measures to insure that customer-owned equipment not interfere with the quality and integrity of the switched network. Subsequently, the Commission entrusted the National Academy of Science to evaluate the status of the network control signalling device. Although somewhat cautious, the Academy has suggested that certification of equipment may pose as one feasible alternative to noncarrier equipment.[13]

The implications of Carterphone bear repetition. For one thing the decision broadens the users option in terms of equipment selection. The business subscriber no longer must lease from the telephone company, but may buy hardware from other manufacturers as well. For another, an important constraint has been softened with respect to suppliers of terminals, data modems and multiplexors as well as PBX or private branch exchanges. Indeed, some claim that the decision has established a two billion dollar market potential for private switching systems.[14] Ironically, the carriers themselves, to meet the demand for PBX's, have turned to independent or nonaffiliated firms supplying such equipment.

Nevertheless, the Carterphone in softening previous restraints continues to pose an interesting set of questions. For example, what precisely is the reach of Carterphone? Will the decision be extended to the residential equipment market? Is the telephone residential market off limits to the independent suppliers of telecommunications equipment? These questions are crucial if for no other reason than terminal display units are already on stream and the carriers themselves are now introducing display phones on a commercial basis. Indeed, the chasm between Carterphone's reality and promise will bulk large if public policy decides that the residential subscriber cannot be entrusted with a freedom of choice comparable to the business subscriber.

Vertical integration

The vertical structure of the carriers has also been subject to reexamination. A relatively unknown antitrust suit involving ITT and General Telephone system is a case inpoint.[15] The suit erupted when General Telephone purchased controlling interest in the Hawaiian Telephone Company—a company that was formerly a customer of ITT and other suppliers. ITT has now filed an antimerger suit on grounds that General forcloses ITT's equipment market. The ITT suit represents a frontal assault on General's equipment subsidiaries, for ITT is seeking a ban on all of General Tel's vertical acquisitions since 1950. In a

word, the suit seeks to remove General Telephone from the equipment manufacturing business.

While the suit is pending, it is obviously difficult to reach any hard conclusions, but one can speculate that anything less than total victory for General Telephone will send reverberations throughout the telephone industry and the equipment market. Certainly, if General Telephone is required to give up its manufacturing affiliate, then the premise of the Western Electric-AT&T consent decree will take on renewed interest.

Another development in the equipment market focuses on Phase II of the FCC's rate investigation of AT&T.[16] This phase is devoted to examing the Western Electric-AT&T relationship. Presumably, the Commission will examine Bell's procurement policies as well as the validity of the utility-supplier relationship. What conclusions the Commission will reach are speculative at this time. In terms of market entry for the computer industry, the implications of Phase II are both real and immediate.

Still another facet of integration is the relationship of communication to EDP and carrier CATV affiliates. In the former, the FCC has ruled that with the exception of AT&T, carriers may offer EDP on a commercial basis via a separate but wholly owned corporation.[17] Nothing apparently prohibits a carrier from offering an EDP service to another carrier—note the current experiments in remote meter reading. By contrast, the FCC has precluded carriers from offering CATV service in areas where they currently render telephone service.[18] Both moves, to repeat, hold important market implications for manufacturers of telecommunication equipment.

Procurement

Finally, equipment procurement has surfaced once again as a policy issue. Consider Carterphone, domestic satellites and specialized common carriers as symptomatic of the procurement theme.

The premise supporting Carterphone is that the user is entitled to free choice in his equipment selection. Once that principle has been established, and that may well be debatable, someone is bound to pose an additional question. Should suppliers be permitted to sell to the Carriers directly rather than through carrier-owned supply affiliates? Perhaps that precedent has already been made. Bell System companies may buy computer hardware directly from computer suppliers, thus permitting the computer industry to bypass Western Electric's traditional procurement assignment.[19] The point may well be asked, does this policy merit generalizing across the board?

Access to the equipment market in the domestic satellite field poses as a second issue. A White House memorandum has advised the FCC that the problems of spatial parking slots and frequency bands do not bar the number of competitive satellite systems.[20] And in return, the FCC has reopened its domestic satellite docket for reevaluation. If the Commission adopts only segments of the White House memo, domestic satellites will presumably raise the issue of competitive bidding in one segment of the hardware market. As it stands now, all satellite equipment, whether secured be Com Sat or the international carriers, must be secured through competitive bids.[21] These rules apply not only to the prime contractor, but all sub-contracting tiers at a minimum threshold of $24,000. The equestion persists, if domestic satellites evolve within the continental U.S., will competitive bidding procedures attend such an introduction whether in satellite bird or in earth terminal stations. These issues will likely gain momentum as the carriers move into the production of ground station equipment.

Finally, an FCC proposed rule made in the area of specialized carriers, bears directly on the equipment market.[22] The docket is traced to the FCC's MCI decision which authorized a specialized carrier to offer service between Chicago and St. Louis.[23] Since the MCI decision, the FCC has received over 1700 applications for microwave sites. In its recent docket, the Commission has solicited views in proposed rulemaking that would permit free access of any and all microwave applicants. As the Commission noted, "Competition in the specialized communications field would enlarge the equipment market for manufacturers other than Western Electric. . . ."[24] If this policy becomes implemented and the FCC can prevent the carriers from engaging in undue price discrimination, it is clear that the one constraint to the growth of specialized common carriers will be the output capacity of firms who manufacture such equipment.

CONCLUSION

In sum the premises supporting the tariffs, structure and practices of the carriers have been exposed to erosion and subject to revision. That change has in turn spilled into the policy arena. Firms in the telecommunication equipment industry—and this includes the computer industry—will find it increasingly difficult to avoid the policy issues of a market whose potential exceeds $5 billion.

One might argue that questions dealing with market entry are in one sense peripheral issues. That is, public policy should direct its attention to existing

structures as well as potential entry. In this context competitive buying practices may well pose as a workable solution to the vertical integration problem. But that solution is obviously of short term duration. The pace of technology is suggesting that something more fundamental must give. Over the next decade, the nation's supply of telecommunication equipment must expand by an order of magnitude and that goal stands in obvious conflict with monopoly control of telecommunication equipment suppliers.

REFERENCES

1 W H MELODY
Interservice subsidy: Regulatory standards and applied economics
Paper presented at a conference sponsored by Institute of Public Utilities Michigan State University 1969

2 *New York Times*
July 29 1970

3 *Final Report*
President's Task Force on Communications Policy
December 7 1968

4 *United States v Western Electric Co*
Civil No 17-49 DNJ filed Feb 14 1949

5 *Consent Decree US v Western Electric Co*
Civil No 17-49 DNJ January 23 1956

6 *United States v General Telephone and Electronics Corp*
Civil No 64-1912 SD NY file June 19 1964

7 In the US District Court District of Hawaii International Telephone and Telegraph Corporation v General Telephone and Electronics Corporation and Hawaiian Telephone Company Civil No 2754 *G T&E's motion under Rule 19 Points and Authorities in Support Thereof* April 21 1970

8 *Smith v Illinois Bell Telephone*
282 US 1930

9 *Telephone investigation*
Special Investigation Docket No. 1, Brief of Bell System Companies on Commissioner Walker's Proposed Report on the Telephone Investigation 1938

10 *The domestic telecommunications carrier industry*
Part I Presidents Task Force on Communications Policy Clearinghouse pp 184-417 US Department of Commerce June 1969

11 *Moody's public utility manual*
Moody's Investors Service Inc New York August 1969

12 Before the FCC in the Matter of Use of the Carterphone Device in Message Toll Telephone Service FCC No 16942 In the Matter of Thomas F Carter and Carter Electronics Corporation Dallas Texas Complainants v American Telegraph and Telephone Company Associated Bell System Companies Southwestern Bell Telephone Company and General Telephone of the Southwest FCC Docket No 17073 *Decision* June 26 1968

13 *Report of a technical analysis of common carrier/user interconnections*
National Academy of Sciences Computer Science and Engineering Board June 10 1970

14 *New York Times*
July 12 1970

15 In the US District Court for the District of Hawaii International Telephone and Telegraph Corp v General Telephone and Electronics Corp and Hawaiian Telephone Company *Complaint for Injunctive Relief* Civil Action No 2754 October 18 1967 Also *Amended Complaint for Injunctive Relief* December 14 1967

16 Before the FCC In the Matter of American Telephone and Telegraph Company and the Associated Bell System companies charges for Interstate and Foreign Communication Service 1966
Stanford Research Institute *Policy Issues Presented by the Interdependence of Computer and Communications Services* Docket No 19979 Contract RD-10056 SRI Project 7379B Clearinghouse for Federal Scientific and Technical Information US Department of Commerce February 1969

17 Before the FCC in the Matter of Regulatory and Policy Problems Presented by the Interdependence of Computer and Communication Service and Facilities Docket No 16979 *Tentative Decision* 1970

18 Before the Federal Communications Commission In the Matter of Applications of Telephone Companies for Section 214 Certificates for Channel Facilities Furnished to Affiliated Community Antenna Television Systems Docket No 18509 *Final Report and Order* January 28 1970

19 *A systems approach to technological and economic imperatives of the telephone network*
Staff Paper 5 Part 2 June 1969 PB184-418 President's Task Force on Communications Policy

20 Memorandum White House to Honorable Sean Burch Chairman Federal Communications Commission January 23 1970

21 Before the FCC In the Matter of Amendment of Part 25 of the Commission's Rules and Regulations with Respect to the Procurement of Apparatus Equipment and Services Required for the Establishment and Operation of the Communication Satellite System, and the Satellite Terminal Stations Docket No 15123 *Report and Order* April 3 1964

22 Before the FCC In the Matter of Establishment of Policies and Procedures for Consideration of Applications to Provide Specialized Common Carrier Services in the Domestic Public Point-to-Point Microwave Radio Service and Proposed Amendments to Parts 21 43 and 61 of the Commission's Rules *Notice of Inquiry to Formulate Policy Notice of Proposed Rulemaking 1 and Order* July 1970 (Cited as FCC Inquiry on Competitive Access)

23 Federal Communications Commission In re Application of Microwave Communications Inc for Construction Permits to Establish New Facilities in the Domestic Public Point to Point Microwave Radio Service at Chicago Illinois St Louis Missouri and Intermediate Points Docket No 16509 *Decision* August 14 1969

24 FCC Inquiry on Competitive Access *op cit* July 1970 p 22

Digital frequency modulation as a technique for improving telemetry sampling bandwidth utilization

by G. E. HEYLIGER

Martin Marietta Corporation
Denver, Colorado

INTRODUCTION

A hybrid of Time Division Multiplexing (TDM) and Frequency Division Multiplexing (FDM), both well-established in theory and practice is described herein. While related to TDM and FDM, the particular combinations of techniques and implementations are novel and, indeed, provide a third alternative for signal multiplexing applications. The essence of the idea is to perform all band translation and filtering via numerical or digital techniques.

Signal multiplexing techniques are widely employed as a means of approaching the established theoretical limitations on communication channel capacity. In general, multiplexing techniques allow several signals to be combined in a way which takes better advantage of the channel bandwidth. FDM systems accomplish this by shifting the input signal basebands by means of modulation techniques, and summing the results. Judicious choice of modulation frequencies allows non-overlapping shifted signal bands, and permits full use of the channel bandwidth. Refinements such as "guard bands" between adjacent signal bands and the use of single sidebands can further affect the system design, but, in general, the arithmetic sum of the individual signal bandwidths must be somewhat less than half the composite channel bandwidth.

TDM systems achieve full utilization of channel bandwidth in quite a different way. Several signals are periodically sampled, and these samples are interleaved so that the individual signal must be sampled at least twice per cycle for the highest signal frequency present in accordance with Nyquist's sampling theorem. In this case, also, the number of signals that can be combined depends upon the sum of individual signal bandwidths and the bandwidth of the channel itself.

The sampling theorem states that only two samples per cycle of the highest frequency component of a strictly band-limited signal are required for complete recovery of that signal. Nevertheless, 5 to 10 samples per cycle are widely employed. There are reasons, practical and otherwise, for the resulting bandwidth extravagance:

1. Many times it is difficult, if not impossible, to place a specific upper limit on "significant'. frequency components. Safe estimates are made.
2. Interpretation of real-time or quick-look plots is simpler and more satisfying if more samples per cycle are available.
3. Aliasing or folding of noise is more severe for relatively low sampling rates and inadequate prefiltering.

This paper acknowledges the practice of over-sampling but avoids the difficulties previously described. Full use is made of the sampling bandwidth by packing several signals into that bandwitdh utilizing a form of FDM. The novelty lies in the use of FDM and the way modulation is achieved for periodically sampled signals.

SYSTEM DESCRIPTION

Before describing the system, it is useful to briefly consider some theoretical background. The following discussion should clarify the basic ideas. Consider a source signal with the spectrum shown in Figure 1(a). It is well known that sampling signals at a frequency $f_s = 1/T$ where T is the time between samples, results in a periodic replication of the original spectrum as shown in Figure 1(b). Modulation of the original signal by frequency f_0 produces the usual sum and difference frequencies, and sampling then results in the replicated pattern shown in Figure 1(c).

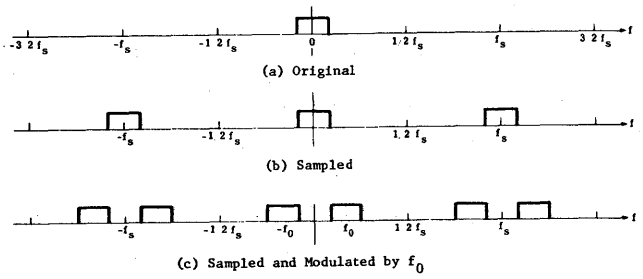


Figure 1—Spectral effects of sampling and modulating

Now consider three source signals with the spectra shown in Figure 2(a), all with roughly the same bandwidth. Modulating the second and third signals with the frequencies $f_s/2$ and $f_s/4$, respectively, results in the shifted spectra shown in Figure 2(b). Summing yields the composite spectrum shown in Figure 2(c). This composite signal now makes full use of the sampling bandwidth.

Figure 3 shows the inverse process of obtaining the original spectra. Demodulating by the same frequencies used for modulation successively brings each signal band to the origin where low pass filtering eliminates all but the original signal.

Since few signals are strictly band-limited, it is evident that crosstalk *noise* will appear in the received signal. This noise can be controlled by the degree of pre- and postfiltering. For certain relatively inactive signals, the crosstalk may be no penalty at all. In general, however, crosstalk presents the same problems here as with any FDM system. The important point to be made is that tracking of the mod/demod oscillators is not relevant since these operations are obtained directly by operating on successive samples, i.e., there are no local oscillators *per se*.

In general, modulation is accomplished by multiplying the signal source by a single sinusoidal frequency or carrier. Sampled signals are modulated in the same way, but the modulating frequency multiplier is required only at successive sample times.

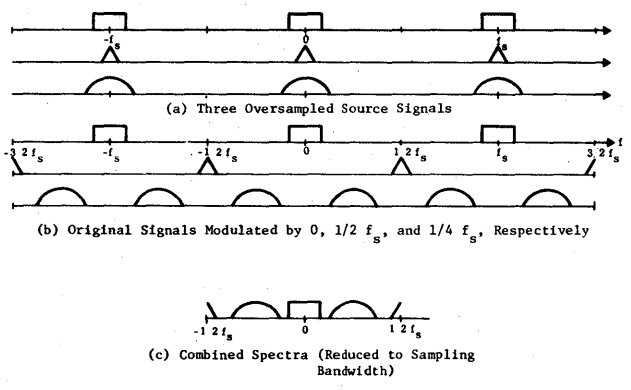


Figure 2—Combinations of oversampled signals

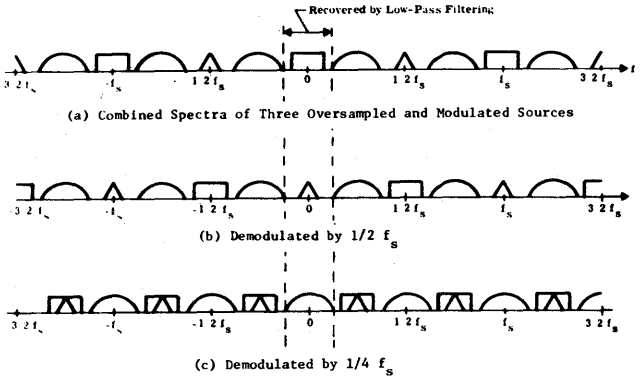


Figure 3—Prefiltered separation of combined signals

Modulation (i.e., multiplication) by integer fractions of the sampling frequency is particularly simple if appropriate sample times are chosen. For example, certain modulation frequency amplitudes are quite easily obtained as shown in Table I. The phase shift of $\pi/4$ for $1/4\ f_s$ was chosen to avoid multiplication by zero yet retain natural symmetry. All the modulation factors may be easily obtained by modifying the sign of the signal magnitude and/or multiplying by a factor of 1/2. Furthermore, the majority of interesting cases are handled by these modulation frequencies, packing two, three, or four FDM channels within the sampling bandwidth. This degree of packing nicely accommodates practical oversampling systems encountered in practice. For particular applications, it may be useful to employ arbitrary modulation frequencies and the corresponding sequence of numerical multipliers (nonrepeating or repeating).

A hybrid form of implementation is shown in Figures 4 and 5. Figure 4 is the modulator, and Figure 5 is the demodulator. Not explicitly shown, but implied,

Table I Modulation Factor

Modulation Frequency	General Expression, k = 0,1,2,...	Periodic Sequence
$\frac{1}{2} f_s$	$\cos\left(\frac{k}{2} f_s 2\pi T\right) = \cos k\pi$	1, -1, ...
$\frac{1}{4} f_s$	$\cos\left(k \frac{f_s}{4} 2\pi T\right) = \cos k\frac{\pi}{2}$	0, 1, 0, -1, ...
	or $\cos\left(k\frac{\pi}{2} + \frac{\pi}{4}\right)$	$\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}$, ... (Note Constant Amplitude)
$\frac{1}{6} f_s$	$\cos\left(k \frac{f_s}{6} 2\pi T\right) = \cos k\frac{\pi}{3}$	$1, \frac{1}{2}, -\frac{1}{2}, -1, -\frac{1}{2}, \frac{1}{2}, \ldots$
$\frac{1}{3} f_s$	$\cos\left(k \frac{f_s}{3} 2\pi T\right) = \cos k\frac{2}{3}\pi$	$1, -\frac{1}{2}, -\frac{1}{2}, \ldots$

TABLE I—Modulation Factor

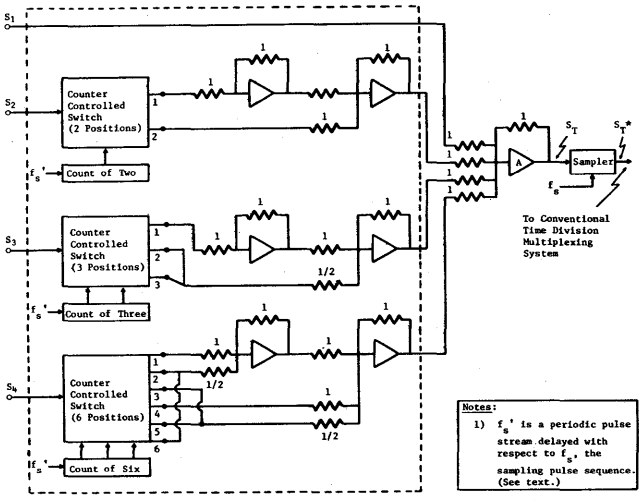


Figure 4—Sampled FDM modulator

is the use of the combined signal output as a single sampled source for conventional TDM systems. The system diagram assumes the case of four signals of roughly equal bandwidth to be combined into a single signal. Subfunctions such as sampling, counting, digital decoding trees, and operational amplifiers can be implemented in a variety of ways utilizing conventional, commercially available functional blocks or components. Details of the subfunction implementations themselves are incidental to the concept but important to the particular application. Referring to Figure 4, the multiplexer modulator works as follows:

Four independent signals (S_1, S_2, S_3, and S_4) are accepted as inputs. One, shown as S_1, goes directly to the summing amplifier, A. Each of the other signals is switched periodically under control of the appropriate binary counter which is synchronized and driven by the sampling frequency pulses. As shown, S_2 is alternately switched from the first to the second of a two-stage cascade of operational amplifiers. The effect of this chain is to alternately multiply S_2 by the factors plus one and minus one, i.e., the modulation factor $\cos k\pi$; = 0, 1, 2, . . . in accordance with Table I and considering the *modulation signal valid at the sample times only*. Similarly, S_3 is multiplied by the periodic sequence (1, $-\frac{1}{2}$, $-\frac{1}{2}$) again in accordance with the third line of Table I. The effect, considered at sample times only, is to modulate S_3 by 1/3 f_s. Finally, S_4 is modulated by 1/6 F_s, by periodically switching this signal to one of six inputs of the operational amplifier chain with the gains (1, $\frac{1}{2}$, $-\frac{1}{2}$, -1, $-\frac{1}{2}$, $\frac{1}{2}$) in accordance with line four of Table I.

All four outputs are summed by the operational amplifier A, and the summed signal sampled at the output of A at the sampling frequency, f_s. It should be noted that the switching counters can be changed at any time after a sample is taken from the output of A; therefore, the design of the system provides that the pulse driving the counters is delayed slightly more than the aperture time of the sampled output. This mechanization provides ample time for switching operations prior to the subsequent sampling. The sampled output signal, S_t*, can be used as an input to a conventional TDM system.

The demodulator shown in Figure 5 is very similar to the modulator. In fact, within the dotted lines it is identical. Here, the appropriate output from a conventional TDM system, S_t*, is used as input to *all four* counter-controlled switches. A sample and hold operation is employed at the input in order to drastically reduce the time response requirements of the operational amplifiers.

Again, sequentially switching the input effectively demodulates S_t by the frequencies 1/6 f_s, 1/3 f_s, and 1/2 f_s. Since this modulation is effective only at the sampling instants, a sample and hold circuit is required at each output. The low-pass filter eliminates components of all but the demodulated signal. Note that for the demodulator, the signals f'_t should precede f_s in phase by the aperture (or pulse width) of S_t*, to allow a maximum time for change in S_t* to be accommodated by the amplifier and switching chain. Since f_s is derived from S_t* and is a periodic signal, any desired relative phasing is readily achieved.

SYSTEM ADVANTAGES AND CAPABILITIES

Several useful and interesting features are inherent in the system:

1. Numerical Modulation of Sampled Signals—Because the modulation signal is required only

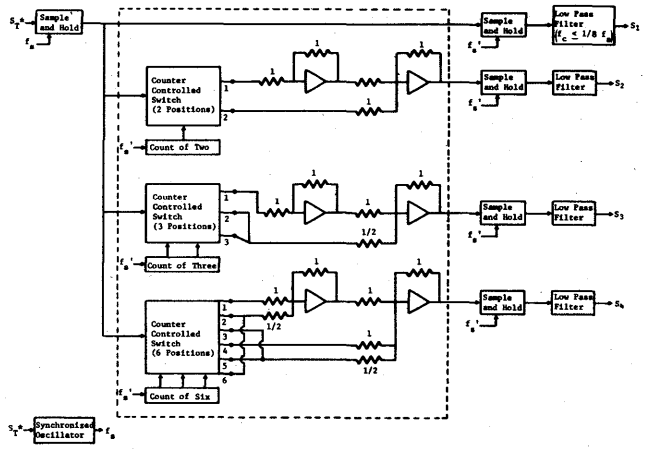


Figure 5—Sampled FDM demodulator

at the sampling instants, a periodic sequence of numerical multipliers substitutes for the *local oscillator* of conventional frequency modulation systems. Conventional oscillator accuracy and stability problems do not arise, and very low frequency modulation is readily achieved.

2. Coincident Sampling of Several Signals—Conventional TDM systems may combine signals sampled at the same rate, but at different instants of time. This approach provides for combining signals sampled at the same rate and *same times*. Full use of conventional TDM techniques can be employed on the combined signal.
3. Full Utilization of Sampling Bandwidth—The sampling rate chosen defines the unaliased bandwidth in a sampled data system. Here, a way of combining several independent signals is employed so that the total sampling bandwidth can be utilized for transmission of information.
4. Signal Independent Choice of Sampling Rate—As a corollary to 3, this system permits, even promotes, oversampling of individual signals. Oversampling is attractive and widely practiced as previously noted. The system described here avoids the usual oversampling penalties by packing several independent signals within the sampling bandwidth.
5. Noise Aliasing Avoidance—Some source signals must be heavily prefiltered or oversampled in order to avoid the noise signal folding effects of sampled data systems. Again, oversampling can be employed without the usual penalties. It should be noted that wideband noise will of course result in crosstalk among the combined channels.

In summary, the system described gives a new dimension in the design of signal multiplexed systems. Combination of these techniques with the conventional TDM and FDM techniques allows the designer to tailor a sampled data system to the peculiarities of a specific set of source signals, while making full use of the available sampled bandwidth.

ALTERNATIVE IMPLEMENTATIONS

The hybrid system described herein uses pulse amplitude modulation (PAM). However, pulse code modulation (PCM) can be employed as well, in one of several attractive alternative implementations. The following system functions can be identified:

1. Sampling, timing, and switching;
2. Analog/digital (A/D) conversion;
3. Sample modulation/demodulation.

The modulator/transmitter also requires an adder for combining the signals, while the demodulator/receiver requires a suitable lowpass filter for each output. Conversion to a digital representation of the signals can be performed at most any point in the system. Following conversion, the subsequent functions are performed via conventional digital arithmetic and logic operations.

Exclusively digital implementation

As an extreme example, consider an implementation that provides A/D conversion at the source (modulator/transmitter input).

Modulation is accomplished by arithmetic multiplication of the source sequence values by the desired modulation sequence, $\cos k\theta_0$ where $\theta = sT$. Note that in this case, the modulation *sequence* need not be a periodic sequence if a means is provided for generating the values $\cos k\theta_0$ for all integers, k.

Independent signals are combined after modulation simply by arithmetic addition of corresponding modulated sequence values. The summed sequence is the output. The combined PCM samples are then handled as with a conventional TDM system.

At the demodulator/receiver, the input is the digital sampled sequence as derived from a conventional PCM system.

Demodulation is performed as before; arithmetic multiplication of the input sequence by the appropriate sequence of values, $\cos k\theta_0$.

Each resulting output must be filtered to eliminate the other signal components. Filtering can be accomplished numerically using either recursive or nonrecursive techniques.

The outputs then are available as separate signals corresponding to those first transmitted. The digital output sequence may be used directly for listing, further processing, or as an input to an incremental plotter. Alternatively, D/A, conversion and hold operations convert the signal to its analog equivalent.

Mixed analog/digital implementations

Evidently, a number of obvious combinations of PAM and PCM are possible. Thus, operational amplifier (op-amp) modulation can be used in combination

with a time-shared A/D converter and arithmetic summation with the result handled as a conventional PCM signal. Similarly, at the receiver, D/A conversion may take place at the output of the PCM arithmetic modulator, and the result passed through a conventional low-pass analog filter for signal recovery.

Analog system simplifications

Figure 4 presents the system in a way that aids description and understanding. Good design practice would permit combination of the modulation and summing functions in a single op-amp stage. Similarly, various combinations of cascades 2- and 3-way switches might be advantageous instead of the single stage 6-way switch shown in Figure 4.

Modulation sequence considerations

The op-amp modulator implementation requires that the modulation sequence, cos k θ_0, be a repeating or periodic sequence. From a practical point of view, only a small number of modulation values should be employed, since each requires additional switching and input to the op-amp. While the only theoretical limitation on the number of values is that θ_0 be some rational fraction of 2π, the simple ratios of the examples shown should prove most useful in practice.

Arithmetic implementation of the modulation and demodulation function imposes no constraint on the number of distinct modulation values, cos k θ_0. Successive values may be generated arithmetically using some equivalent of the following algorithm:

$$\sin k\theta_0 = \cos(k-1)\theta_0 \sin\theta_0 + \sin(k-1)\theta_0 \cos\theta_0$$

$$\cos k\theta_0 = \cos(k-1)\theta_0 \cos\theta_0 - \sin(k-1)\theta_0 \sin\theta_0$$

Only the initial values cos θ_0 and sin θ_0 are required to start. If θ_0 is some rational fraction of 2π, the sequence will be repeating; otherwise, not. In this case any desired modulation frequency (ω_0) may be realized.

Bandwidth packing variations

While roughly equal bandwidths were assumed for the combined signals of the system described, the fundamental constraint is that the sum of signal bandwidths plus guard bands must be less than the sampling frequency. As usual with FDM systems, both upper and lower sidebands for each signal must be included in this consideration. Choice of a suitable modulation frequency then depends upon the placement of each signal band within the sampling bandwidth. Clearly, many variations of center frequencies and bandwidth are feasible and useful.

Variations in digital system

A general purpose digital computer can perform all operations required for modulating, summing, demodulating, and filtering. Where such a computer is already employed in the data system for switching, comparison, calibration, and control, the additional functions described here become particularly attractive. Standard programming practices can be used to perform the essential functions described here.

Alternatively, for the system example the arithmetic operations required are quite simple. Multiplications of ½ and −1 are readily realized by right shift and sign change operations, respectively. A special purpose digital computer with few storage registers and capability for "right shift," "add," "sign change," and conventional register transfers, will provide the required functions.

CONCLUSION

The digital frequency modulation technique described herein permits combination of several signals into a single signal having a sampled bandwidth equal to the sum of the original signal bandwidths. Utilization of this technique to reduce the penalties of oversampled telemetry channels appears particularly attractive.

THE ALOHA SYSTEM—Another alternative for computer communications*

by NORMAN ABRAMSON

University of Hawaii
Honolulu, Hawaii

INTRODUCTION

In September 1968 the University of Hawaii began work on a research program to investigate the use of radio communications for computer-computer and console-computer links. In this report we describe a remote-access computer system—THE ALOHA SYSTEM—under development as part of that research program[1] and discuss some advantages of radio communications over conventional wire communications for interactive users of a large computer system. Although THE ALOHA SYSTEM research program is composed of a large number of research projects, in this report we shall be concerned primarily with a novel form of random-access radio communications developed for use within THE ALOHA SYSTEM.

The University of Hawaii is composed of a main campus in Manoa Valley near Honolulu, a four year college in Hilo, Hawaii and five two year community colleges on the islands of Oahu, Kauai, Maui and Hawaii. In addition, the University operates a number of research institutes with operating units distributed throughout the state within a radius of 200 miles from Honolulu. The computing center on the main campus operates an IBM 360/65 with a 750 K byte core memory and several of the other University units operate smaller machines. A time-sharing system UHTSS/2, written in XPL and developed as a joint project of the University Computer Center and THE ALOHA SYSTEM under the direction of W. W. Peterson is now operating. THE ALOHA SYSTEM plans to link interactive computer users and remote-access input-output devices away from the main campus to the central computer via UHF radio communication channels.

* THE ALOHA SYSTEM is supported by the Office of Aerospace Research (SRMA) under Contract Number F44620-69-C-0030, a Project THEMIS award.

WIRE COMMUNICATIONS AND RADIO COMMUNICATIONS FOR COMPUTERS

At the present time conventional methods of remote access to a large information processing system are limited to wire communications—either leased lines or dial-up telephone connections. In some situations these alternatives provide adequate capabilities for the designer of a computer-communication system. In other situations however the limitations imposed by wire communications restrict the usefulness of remote access computing.[2] The goal of THE ALOHA SYSTEM is to provide another alternative for the system designer and to determine those situations where radio communications are preferable to conventional wire communications.

The reasons for widespread use of wire communications in present day computer-communication systems are not hard to see. Where dial-up telephones and leased lines are available they can provide inexpensive and moderately reliable communications using an existing and well developed technology.[3,4] For short distances the expense of wire communications for most applications is not great.

Nevertheless there are a number of characteristics of wire communications which can serve as drawbacks in the transmission of binary data. The connect time for dial-up lines may be too long for some applications; data rates on such lines are fixed and limited. Leased lines may sometimes be obtained at a variety of data rates, but at a premium cost. For communication links over large distances (say 100 miles) the cost of communication for an interactive user on an alphanumeric console can easily exceed the cost of computation.[5] Finally we note that in many parts of the world a reliable high quality wire communication network is not available and the use of radio communications for data transmission is the only alternative.

There are of course some fundamental differences

between the data transmitted in an interactive time-shared computer system and the voice signals for which the telephone system is designed.[6] First among these differences is the burst nature of the communication from a user console to the computer and back. The typical 110 baud console may be used at an average data rate of from 1 to 10 baud over a dial-up or leased line capable of transmitting at a rate of from 2400 to 9600 baud. Data transmitted in a time-shared computer system comes in a sequence of bursts with extremely long periods of silence between the bursts. If several interactive consoles can be placed in close proximity to each other, multiplexing and data concentration may alleviate this difficulty to some extent. When efficient data concentration is not feasible however the user of an alphanumeric console connected by a leased line may find his major costs arising from communication rather than computation, while the communication system used is operated at less than 1 percent of its capacity.

Another fundamental difference between the requirements of data communications for time-shared systems and voice communications is the asymmetric nature of the communications required for the user of interactive alphanumeric consoles. Statistical analyses of existing systems indicate that the average amount of data transmitted from the central system to the user may be as much as an order of magnitude greater than the amount transmitted from the user to the central system.[6] For wire communications it is usually not possible to arrange for different capacity channels in the two directions so that this asymmetry is a further factor in the inefficient use of the wire communication channel.

The reliability requirements of data communications constitute another difference between data communication for computers and voice communication. In addition to errors in binary data caused by random and burst noise, the dial-up channel can produce connection problems—e.g., busy signals, wrong numbers and disconnects. Meaningful statistics on both of these problems are difficult to obtain and vary from location to location, but there is little doubt that in many locations the reliability of wire communications is well below that of the remainder of the computer-communication system. Furthermore, since wire communications are usually obtained from the common carriers this portion of the overall computer-communication system is the only portion not under direct control of the system designer.

THE ALOHA SYSTEM

The central computer of THE ALOHA SYSTEM (an IBM 360/65) is linked to the radio communication channel via a small interface computer (Figure 1). Much of the design of this multiplexor is based on the design of the Interface Message Processors (IMP's) used in the ARPA computer net.[4,7] The result is a Hawaiian version of the IMP (taking into account the use of radio communications and other differences) which has been dubbed the MENEHUNE (a legendary Hawaiian elf). The HP 2115A computer has been selected for use as the MENEHUNE. It has a 16-bit word size, a cycle time of 2 microseconds and an 8K-word core storage capacity. Although THE ALOHA SYSTEM will also be linked to remote-access input-output devices and small satellite computers through the MENEHUNE, in this paper we shall be concerned with a random access method of multiplexing a large number of low data rate consoles into the MENEHUNE through a single radio communication channel.

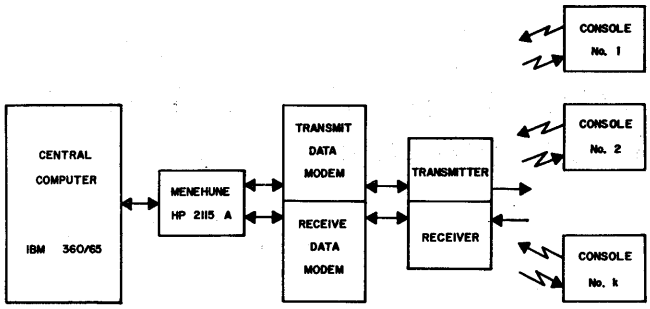


Figure 1—THE ALOHA SYSTEM

THE ALOHA SYSTEM has been assigned two 100 KHZ channels at 407.350 MHZ and 413.475 MHZ. One of these channels has been assigned for data from the MENEHUNE to the remote consoles and the other for data from the consoles to the MENEHUNE. Each of these channels will operate at a rate of 24,000 baud. The communication channel from the MENEHUNE to the consoles provides no problems. Since the transmitter can be controlled and buffering performed by the MENEHUNE at the Computer Center, messages from the different consoles can be ordered in a queue according to any given priority scheme and transmitted sequentially.

Messages from the remote consoles to the MENEHUNE however are not capable of being multiplexed in such a direct manner. If standard orthogonal multiplexing techniques (such as frequency or time multiplexing) are employed we must divide the channel from the consoles to the MENEHUNE into a large number of low speed channels and assign one to each console, whether it is active or not. Because of the fact that at any given time only a fraction of the total number of consoles in the system will be active and because of the burst nature of the data from the con-

soles such a scheme will lead to the same sort of inefficiencies found in a wire communication system. This problem may be partly alleviated by a system of central control and channel assignment (such as in a telephone switching net) or by a variety of polling techniques. Any of these methods will tend to make the communication equipment at the consoles more complex and will not solve the most important problem of the communication inefficiency caused by the burst nature of the data from an active console. Since we expect to have many remote consoles it is important to minimize the complexity of the communication equipment at each console. In the next section we describe a method of random access communications which allows each console in THE ALOHA SYSTEM to use a common high speed data channel without the necessity of central control or synchronization.

Information to and from the MENEHUNE in THE ALOHA SYSTEM is transmitted in the form of "packets," where each packet corresponds to a single message in the system.[8] Packets will have a fixed length of 80 8-bit characters plus 32 identification and control bits and 32 parity bits; thus each packet will consist of 704 bits and will last for 29 milliseconds at a data rate of 24,000 baud.

The parity bits in each packet will be used for a cyclic error detecting code.[9] Thus if we assume all error patterns are equally likely the probability that a given error pattern will not be detected by the code is[10]

$$2^{-32} \approx 10^{-9}.$$

Since error *detection* is a trivial operation to implement,[10] the use of such a code is consistent with the requirement for simple communication equipment at the consoles. The possibility of using the same code for error *correction* at the MENEHUNE will be considered for a later version of THE ALOHA SYSTEM.

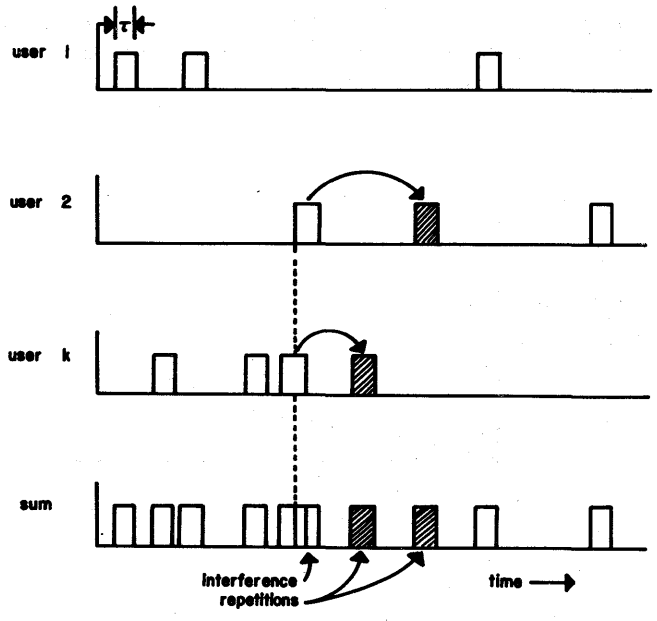


Figure 2—ALOHA communication multiplexing

The random access method employed by THE ALOHA SYSTEM is based on the use of this error detecting code. Each user at a console transmits packets to the MENEHUNE over the same high data rate channel in a completely unsynchronized (from one user to another) manner. If and only if a packet is received without error it is acknowledged by the MENEHUNE. After transmitting a packet the transmitting console waits a given amount of time for an acknowledgment; if none is received the packet is retransmitted. This process is repeated until a successful transmission and acknowledgment occurs or until the process is terminated by the user's console.

A transmitted packet can be received incorrectly because of two different types of errors; (1) random noise errors and (2) errors caused by interference with a packet transmitted by another console. The first type of error is not expected to be a serious problem. The second type of error, that caused by interference, will be of importance only when a large number of users are trying to use the channel at the same time. Interference errors will limit the number of users and the amount of data which can be transmitted over this random access channel.

In Figure 2 we indicate a sequence of packets as transmitted by k active consoles in the ALOHA random access communication system.

We define τ as the duration of a packet. In THE ALOHA SYSTEM τ will be equal to about 34 milliseconds; of this total 29 milliseconds will be needed for transmission of the 704 bits and the remainder for receiver synchronization. Note the overlap of two packets from different consoles in Figure 2. For analysis purposes we make the pessimistic assumption that when an overlap occurs neither packet is received without error and both packets are therefore retransmitted.* Clearly as the number of active consoles increases the number of interferences and hence the number of retransmissions increases until the channel clogs up with repeated packets.[11] In the next section we compute the average number of active consoles which may be supported by the transmission scheme described above.

Note how the random access communication scheme of THE ALOHA SYSTEM takes advantage of the nature of the radio communication channels as opposed to wire communications. Using the radio channel as we have described each user may access the same

* In order that the retransmitted packets not continue to interfere with each other we must make sure the retransmission delays in the two consoles are different.

channel even though the users are geographically dispersed. The random access communication method used in THE ALOHA SYSTEM may thus be thought of as a form of data concentration for use with geographically scattered users.

RANDOM ACCESS RADIO COMMUNICATIONS

We may define a random point process for each of the k active users by focusing our attention on the starting times of the packets sent by each user. We shall find it useful to make a distinction between those packets transmitting a given message from a console for the first time and those packets transmitted as repetitions of a message. We shall refer to packets of the first type as *message packets* and to the second type as *repetitions*. Let λ be the average rate of occurrence of message packets from a single active user and assume this rate is identical from user to user. Then the random point process consisting of the starting times of message packets from all the active users has an average rate of occurrence of

$$r = k\lambda$$

where r is the average number of message packets per unit time from the k active users. Let τ be the duration of each packet. Then if we were able to pack the messages into the available channel space perfectly with absolutely no space between messages we would have

$$r\tau = 1.$$

Accordingly we refer to $r\tau$ as the *channel utilization*. Note that the channel utilization is proportional to k, the number of active users. Our objective in this section is to determine the maximum value of the channel utilization, and thus the maximum value of k, which this random access data communication channel can support.

Define R as the average number of message packets plus retransmissions per unit time from the k active users. Then if there are any retransmissions we must have $R > r$. We define $R\tau$ as the *channel traffic* since this quantity represents the average number of message packets plus retransmissions per uni ttime multiplied by the duration of each packet or retransmission. In this section we shall calculate $R\tau$ as a function of the channel utilization, $r\tau$.

Now assume the interarrival times of the point process defined by the start times of all the message packets plus retransmissions are independent and exponential. This assumption, of course, is only an approximation to the true arrival time distribution. Indeed, because of the retransmissions, it is strictly speaking not even mathematically consistent. If the retransmission delay is large compared to τ, however, and the number of retransmissions is not too large this assumption will be reasonably close to the true distribution. Moreover, computer simulations of this channel indicate that the final results are not sensitive to this distribution. Under the exponential assumption the probability that there will be no events (starts of message packets or retransmissions) in a time interval T is $\exp(-RT)$.

Using this assumption we can calculate the probability that a given message packet or retransmission will need to be retransmitted because of interference with another message packet or retransmission. The first packet will overlap with another packet if there exists at least one other start point τ or less seconds before or τ or less seconds after the start of the given packet. Hence the probability that a given message packet or retransmission will be repeated is

$$[1 - \exp(-2R\tau)]. \tag{1}$$

Finally we use (1) to relate R, the average number of message packets plus retransmissions per unit time to r, the average number of message packets per unit time. Using (1) the average number of retransmissions per unit time is given by

$$R[1 - \exp(-2R\tau)]$$

so that we have

$$R = r + R[1 - \exp(-2R\tau)]$$

or

$$r\tau = R\tau e^{-2R\tau}. \tag{2}$$

Equation (2) is the relationship we seek between the channel utilization $r\tau$ and the channel traffic $R\tau$. In Figure 3 we plot $R\tau$ versus $r\tau$.

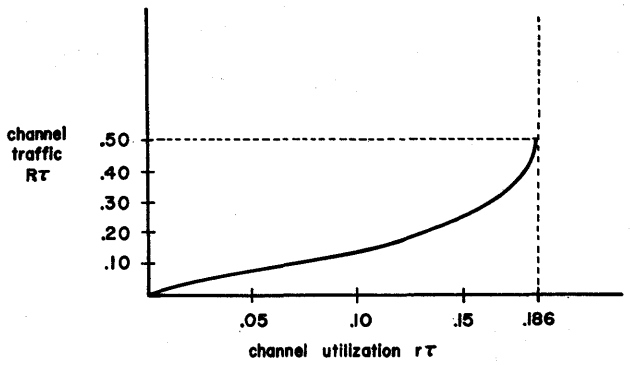


Figure 3—Channel utilization vs channel traffic

Note from Figure 3 that the channel utilization reaches a maximum value of $1/2e=0.186$. For this value of $r\tau$ the channel traffic is equal to 0.5. The traffic on the channel becomes unstable at $r\tau=1/2e$ and the average number of retransmissions becomes unbounded. Thus we may speak of this value of the channel utilization as the *capacity* of this random access data channel. Because of the random access feature the channel capacity is reduced to roughly one sixth of its value if we were able to fill the channel with a continuous stream of uninterrupted data.

For THE ALOHA SYSTEM we may use this result to calculate the maximum number of interactive users the system can support.

Setting

$$r\tau=k\lambda\tau=1/2e$$

we solve for the maximum number of active users

$$k_{\max}=(2e\lambda\tau)^{-1}.$$

A conservative estimate of λ would be 1/60 (seconds)$^{-1}$, corresponding to each active user sending a message packet at an average rate of one every 60 seconds. With τ equal to 34 milliseconds we get

$$k_{\max}=324. \tag{3}$$

Note that this value includes only the number of active users who can use the communication channel simultaneously. In contrast to usual frequency or time multiplexing methods while a user is not active he consumes no channel capacity so that the total number of users of the system can be considerably greater than indicated by (3).

The analysis of the operation of THE ALOHA SYSTEM random access scheme provided above has been checked by two separate simulations of the system.[12,13] Agreement with the analysis is excellent for values of the channel utilization less than 0.15. For larger values the system tends to become unstable as one would expect from Figure 3.

REFERENCES

1 N ABRAMSON et al
1969 annual report THE ALOHA SYSTEM
University of Hawaii Honolulu Hawaii January 1970

2 M M GOLD L L SELWYN
Real time computer communications and the public interest
Proceedings of the Fall Joint Computer Conference pp 1473-1478 AFIPS Press 1968

3 R M FANO
The MAC system: The computer utility approach
IEEE Spectrum Vol 2 No 1 January 1965

4 L G ROBERTS
Multiple computer networks and computer communication
ARPA report Washington D C June 1967

5 J G KEMENY T E KURTZ
Dartmouth time-sharing
Science Vol 162 No 3850 p 223 October 1968

6 P E JACKSON C D STUBBS
A study of multiaccess computer communications
Proceedings of the Spring Joint Computer Conference pp 491-504 AFIPS Press 1969

7 *Initial design for interface message processors for the ARPA computer network*
Report No 1763 Bolt Beranek and Newman Inc January 1969

8 R BINDER
Multiplexing in THE ALOHA SYSTEM: MENEHUNE-KEIKI design considerations
ALOHA SYSTEM Technical Report B69-3 University of Hawaii Honolulu Hawaii November 1969

9 W W PETERSON E J WELDON JR
Error-correcting codes—Second edition
John Wiley & Sons New York New York 1970

10 D T BROWN W W PETERSON
Cyclic codes for error detection
Proceedings IRE Vol 49 pp 228-235 1961

11 H H J LIAO
Random access discrete address multiplexing communications for THE ALOHA SYSTEM
ALOHA SYSTEM Technical Note 69-8 University of Hawaii Honolulu Hawaii August 1969

12 W H BORTELS
Simulation of interference of packets in THE ALOHA SYSTEM
ALOHA SYSTEM Technical Report B70-2 University of Hawaii Honolulu Hawaii March 1970

13 P TRIPATHI
Simulation of a random access discrete address communication system
ALOHA SYSTEM Technical Note 70-1 University of Hawaii Honolulu Hawaii April 1970

Computer-aided system design*

by E. DAVID CROCKETT, DAVID H. COPP, J. W. FRANDEEN, and CLIFFORD A. ISBERG

Computer Synectics, Incorporated
Santa Clara, California

PETER BRYANT and W. E. DICKINSON

IBM ASDD Laboratory
Los Gatos, California

and

MICHAEL R. PAIGE

University of Illinois
Urbana, Illinois

INTRODUCTION

This paper describes the Computer-Aided System Design (CASD) system, a proposed collection of computer programs to aid in the design of computers and similar devices. CASD is a unified system for design, encompassing high-level description of digital devices, simulation of the device functions, automatic translation of the description to detailed hardware (or other) specifications, and complete record-keeping support. The entire system may be on-line, and most day-to-day use of the system would be in conversational mode.

Typically, the design of digital devices requires a long effort by several groups of people working on different aspects of the problem. The CASD system would make a central collection of all the design information available through terminals to anyone working on the job. With conversational access to a central file, many alternative designs can be quickly evaluated, proven standard design modules can be selected, and the latest version of the design can be automatically documented. The designer works only with high-level descriptions, which reduce the number of trivial errors and ensure the use of standard design techniques.

From October, 1968, through December, 1969, the authors participated in a study at the IBM Advanced Systems Development Laboratory in Los Gatos, California, which defined the proposed CASD system and looked into the problems of building the various component programs. Details of several prototype programs which were implemented are given elsewhere.[1] There are no present plans to continue work in this area. This paper is essentially a feasibility report, describing the overall system structure and the reasons for choosing it. It includes descriptions of the data forms in the system and of the component programs, discussions of the overall approach, and an example of a device described in the CASD design language.

THE SYSTEM IN GENERAL

The (proposed) Computer-Aided System Design (CASD) system is a collection of programs to aid the computer designer in his daily work, and to coordinate record-keeping and documentation. It offers the designer five major facilities:

High-level description

The designer describes his device in a high-level, functional language resembling PL/I, but tailored to his special needs. This is the only description he enters into the system, and the one to which all subsequent modifications, etc., refer.

* This work was performed at the IBM Advanced Systems Development Laboratory, Los Gatos, California.

High-level simulation

An interpretive simulator allows the designer to check out his design at a functional level, before it is committed to hardware. The simulation is interactive, allowing the designer to "watch" his design work and evaluate precisely design alternatives.

Translation to logic specifications

The high-level design, after testing by simulation, is automatically translated to detailed logic specifications. These specifications may take a variety of forms, such as (1) input to conventional Design Automation (DA) systems, or (2) microcode for an existing machine.

On-line, conversational updating

The designer makes design changes and does his general day-to-day work at a terminal, in a conversational mode. Batch facilities are also available.

Complete file maintenance and documentation

Extensive record-keeping is provided to keep track of different machines, different designs of machines, different versions of designs, results of simulation runs, and so forth. High-level documentation of designs (analogous to that produced at lower levels by today's design automation systems) is a natural by-product of the CASD organization.

The CASD system can thus be viewed as an extension to higher levels of current systems for design, in roughly the same way that compilers are functional extensions of assemblers to higher levels.

The general organization of the system is pictured in Figure 1. The designer describes his device in a *source design language*, which is translated by a compiler-like program called the *encoder* to an *internal form*. The internal form is the input both to the high-level simulator (called the *interpreter*) and to a series of *translators* (two are shown in Figure 1) which convert it to the appropriate form of logic specifications. Different series of translators give different kinds of final output (e.g., one series for DA input, another series for microcode). The entire system is on-line, operating under control of the CASD monitor, which handles communication to and from the terminals. The user interface programs handle the direct "talking" to the user and invoke the proper functional programs.

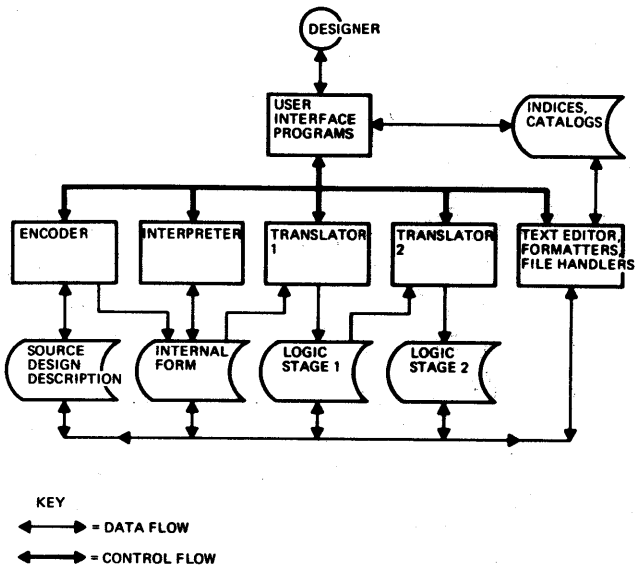


Figure 1—The CASD system

DATA FORMS IN THE CASD SYSTEM

Source design description

The CASD design language the designer uses is a variant of PL/I, stripped of features not needed for computer design and enriched with a few specialized features for such work. PL/I[2] and CASD's language[3] are described more fully elsewhere.

Procedures

The basic building block in a CASD description is the procedure. A procedure consists of: (1) declarations of the entities involved in the procedure, and (2) statements of what is to be done to these entities. A procedure is written as a PROCEDURE statement, followed by the declarations and statements, followed by a matching END statement, in the usual PL/I format:

```
PROC1:    PROCEDURE;

              declarations and statements

          END PROC1;
```

defines a procedure whose name is PROC1.

A procedure represents some logical module of the design, e.g., an adder. A complete design, in general, would have many such procedures, some nested within

others. The adder procedure, for example, may contain a half-adder as a subprocedure.

Data items

Each procedure operates on certain *data items*, such as registers or terminals. These items are defined by DECLARE statements, which have the general format:

DECLARE name attribute, attribute, . . .;

The name is used to refer to the item throughout the description. The attributes describe the item in more detail, and are of two types—logical and physical. Logical attributes describe the function of the item (it is bit storage, or a clock, say); physical attributes describe the form the item is to take in hardware (magnetic core, for example). Logical attributes influence the encoding, interpreting, and translating functions. Physical attributes, on the other hand, are ignored by the interpreter, giving a truly functional simulation.

Like any block-structured language, the CASD language has rules about local and global variables, and scope of names. These have been taken directly from the corresponding rules for PL/I.

Statements

The basic unit for describing what is to be done to the data items is the *expression*, defined as in PL/I but with some added Boolean operators, such as exclusive or (⅋), and some modifications to the bit string arithmetic.

The basic statement types for describing actions on data items are the assignment, WAIT, CALL, GO TO, IF, DO, and RETURN statements. These are basically as they are in PL/I, except as described below.

1. The assignment statement is extended to allow concatenated items to appear on the left-hand side. Thus:

   XREG || YREG:=ZREG;

   where XREG and YREG are 16 bits each and ZREG is 32 bits, means to put the high 16 bits of ZREG into XREG and the low 16 bits into YREG. In combination with the SUBSTR built-in function,[4] this assignment statement offers convenient ways to describe shifting and similar operations. The assignment symbol itself is the ALGOL ":=" rather than "=" as in PL/I.
2. The WAIT statement takes the form

   WAIT(expression);

   It thus differs from PL/I in that it allows one to specify a wait until an arbitrary expression is satisfied. This is useful for synchronizing tasks (see below).
3. The GO TO statement includes the facility of going to a label variable, and the label variable may be subscripted. This is useful for describing such operations as op-code decoding—for example: GO TO ROUTINE (OP).

Sequencing

The differences in motivation between CASD's language and PL/I are most evident in matters of sequence control and parallelism. PL/I, as a programming language, does not emphasize the use of parallelism. Programs are described and executed sequentially, which is not adequate for a design language.

The basic unit of work in CASD is the *node*. A node is a collection of actions which can be performed at the same time. For example, XREG:=YREG; and P:=Q; can be performed together if all the items involved are distinct. On the other hand, XREG:=YREG; ZREG:=XREG; cannot be performed (as written) at the same time, since the result of the first written operation is needed to do the second. The basic CASD rules are:

1. Operations are written as sequential statements.
2. However these operations are performed (sequentially or in parallel), the end results will be the same as the results of performing them sequentially.
3. Sequential statements will be combined into a single node (considered as being done in parallel) whenever this does not violate rule 2. That is, CASD assumes you mean parallel unless there's some "logical conflict."[5]

Of course, the designer may want to override rules 2 and 3. Another rule gives him one way to do this:

4. A labelled statement always begins a new node. Another way is by specifying parallelism explicitly. If the DO statement is written as DO CONCURRENTLY, all statements within the DO will be executed in parallel. Finally, the TASK option of the CALL statement makes it possible to set several tasks operating at once.

Preprocessor facilities

Some of the PL/I preprocessor facilities have been retained. These include the iterative %DO, which is particularly useful in describing repetitive operations, and the preprocessor assignment statement, useful for specifying word lengths, etc.

No defaults

Unlike PL/I, the CASD language follows the principle that nothing should be hidden from the designer. In particular, it has no default attributes, and everything must be declared. Similarly, it does not allow subscripted subscripts, subscripted parameters passed to subroutines, or anything else that might force the encoder to generate temporary registers not specified by the designer. Such restrictions might be relaxed in a later version, but we feel that until we have more experience with such systems, we had better hide as little as possible.

Internal form

Before the source description can be conveniently manipulated by other programs, it must be translated to an internal form. This form is designed to be convenient for both the translator programs and the interpreter. Compromises are necessary, of course—a computer program might be the most convenient form for simulation, but would be of no use at all to the translator.

The CASD internal form resembles the tabular structure used for intermediate results in compilers for programming languages. It consists of four kinds of tables: descriptors, expressions, statements and nodes.

The descriptor table records the nature of each item (taken from its DECLARE statement). The entries are organized according to the block structure of the source description and the scope-of-names rules of the language.

The expression table contains reverse Polish forms of all expressions in the source description, with names replaced by pointers to descriptors. Each expression appears only onee in the expression table, although it may appear often in the source description. In effect, the expression table lists the combinational logic the translator must generate.

The statement table consists of one entry for each statement in the source description, with expressions replaced by pointers to entries in the expression table, and a coded format for the rest of the statement (statement type plus parameters).

The node table tells which statements in the statement table belong in the same node, and the order in which various nodes should be executed.

The internal form has thus extracted three things from the source description—data items, actions to be taken on those items, and the timing of the actions—and recorded them in three separate tables—the descriptor, the statement, and the node tables. The expression table is added for convenience.

Simulation results

The high-level simulation involves three forms of data: values of the variables, control information, and run statistics.

Before a simulation run begins, the variables of the source design description (corresponding to registers, etc.) must be assigned initial values. One way to do this is with the INITIAL attribute in the DECLARE statement, which makes initialization of the variables at execution time a fundamental part of the description. Sometimes, though, the designer may want to test a special case, and simulate his design starting from some special set of initial values. CASD permits him to store one or more sets of initial values in his files; and for a given simulation run, to specify the set of initial values to be used. In this way, he can augment or override the INITIAL attribute.

At the end of a simulation run, the final values of the variables may be saved and used for print-outs, statistics gathering, or as initial values for the next simulation run. That is, a simulation run may continue where the last one left off.

The high-level, interpretive simulation in CASD is perhaps most useful because of its control options. As an interpreter, operating from a static, tabular description of the device, the CASD simulator can give the user unusually complete control over the running of the simulation. Through a terminal, he can at any time tell the system which variables to trace, how many nodes to interpret at a time, when to stop the simulation (e.g., stop if XREG ever gets bigger than 4 and display the results), and so forth. These control conditions may be saved just as the data values may be, and a simulation run may use either old or new control conditions.

Permanent records of a simulation also include summaries of run statistics (the number of subprocedure calls, number of waits, etc.).

Translator output

Different translators produce different kinds of output. Assembly-language level listings of mircocode might be needed for some lower-level systems, the coded equivalent of ALD sheets for others. Typically, output would include error and warning messages.

File structure

In an on-line, conversational system, it is particularly important that the working data be easily accessible to the user and the control language seem natural to him. CASD attempts to facilitate user control in two ways: through the user interface programs, and the structure of the data files.

The basic organizational unit in the CASD files is called the *design*. A design consists of all the data pertinent to the development of some given device. A design may have many *versions*, representing current alternatives or successive revisions. Each version has some or all of the basic forms of data associated with it: source description, internal form, simulation results, translator output, and so on.

Two catalogs, one for designs and one for versions, are the basic access points to CASD data. A typical entry in the design catalog (a *design record*) contains a list of pointers to the version descriptors for each version of every design in the system. The version descriptor contains pointers to each of the various forms of data for that version (source description, . . .) plus control information telling which set of translators has been applied to the design in this version, and so on.

These descriptors give the user interface programs efficient access to needed data. For example, if the user asks to translate a given design, the interface finds the version descriptor, and can then tell if the design has been encoded, and if not, inform the user and request the input parameters for encoding.

PROGRAMS IN THE CASD SYSTEM

CASD monitor and support programs

All the CASD component programs are under control of a monitor program, which provides the basic services for communicating with terminals and allocates system resources. In the prototype version[6] the environment was OS/360 MVT, and it was convenient to set up the monitor as a single job, attaching one subtask for each CASD terminal. The CASD files were all in one large data set, and access to them was controlled by service routines in the monitor. The monitor also controlled the allocation of CPU time to various CASD terminals within the overall CASD job. This approach makes it easier to manage the various interrelated data forms within the versions, and would probably work in environments other than OS/360 as well.

Besides the monitor and the data access routines, the support programs include a text-editing routine to use in editing the source description.

User interface programs

CASD system control is not specified in some general language. Rather, each CASD function has its own interface *program*, which has the complete facilities of the system available to it.

The design records and version descriptors give precisely the information needed by user interface programs. A typical user interface program might be one for encoding and simulating a source design description already in the CASD files. The version descriptor shows, for example, whether or not the source description has already been encoded. The interface may then give the user a message like "Last week you ran this design for 400 nodes. Should the results of that run be used as initial values for this run?" The point is that the conversation is natural to the task at hand. The tasks under consideration are well defined, and each natural combination of them has its own interface program.

Encoder

Since the CASD encoder is roughly the first half of a compiler, it may be built along pretty standard lines. Care must be taken only in providing some sort of conversational compilation facility. Conversational interaction is an important part of the CASD approach to design, and some sort of line-by-line feedback is required. Similarly, since modification is so common in design work, recompilation must be as efficient as possible. Incremental compilation—translating each source statement as far as possible on input, independently of other statements—is one answer. Then only those statements which have changed since the last compilation need be recompiled. The approach used in the CASD prototype is described elsewhere.[7,8]

Interpreter

The basic unit that the interpreter simulates is the node table, the various statements which comprise

the node are identified. These statments are then "executed" in two steps: First, all the expressions in the statements are evaluated; second, the results are stored. By this two-step procedure, the parallelism inherent in the definition of the node is correctly simulated.

The interpreter steps from node to node, as they appear in the node table, with several exceptions. One is the conditional branch, where some (usually just one) statment within the node must be evaluated or executed to determine what the next node should be. Another exception is when wait, halt, or trace conditions have been met. Such "values" as "stop if this item is referenced" may be stored with the item's descriptor in the internal form. If this kind of condition is encountered in a node, the interpreter takes the action indicated before going to the next node. Control conditions like these may be altered dynamically by the user, who may, when a "halt" condition is satisfied, not only observe the variables and their values, but alter the control conditions.

Translators

The translator used in the prototype system converts the internal form to a list structure of the machine logic. Techniques for translating from this to DA input or actual circuits for any given circuit family are straightforward. The elements of the list structure are: hash cells, part cells, subexpression cells, assignment cells, action cells, condition cells, and clock cells. Hash (as in "hash code") cells contain index entries and cross-references to the rest of the cells. Part cells contain all the information declared about each item; subexpression cells indicate how the various items are to be combined to form circuits. Assignment cells tell what data is to be transferred to where. Action cells and condition cells are lists of which actions (e.g., assignments) are to be taken and under which conditions. Clock cells contain labels and other information about sequencing. Most of the information in these cells comes fairly directly from the appropriate tables in the internal form, but the translator links the cells in a way that corresponds to the hardware that must be generated. For example, all assignments to a given register are linked together, and this might correspond (for a particular circuit family) to a single storage bus.

Essentially, the translator reduces the high-level description to a form which currently known procedures[9,10,11,12] can handle, by breaking up the information in the internal form and linking it up again in several different ways. Details of the various linking schemes and how they relate to the source description are given elsewhere.[13]

Other programs

The general structure of the CASD system is flexible enough to permit addition of other programs. A few possibilities have been considered.

One obvious drawback of interpretive simulation is the overhead. Simulation by compilation to machine code would be perhaps 50 or 100 times as fast. This is a significant difference on long runs, after the design is basically checked out (e.g., runs to get firm performance figures).

A generalized assembler program to prepare program input to the interpreter would allow larger quantities of software to be tested by "running" it on the machine being simulated.

Cost-estimating programs operating directly from the internal form would give quick-and-dirty estimates without going through the entire hardware translation process. Translation from the internal form to microcode is another possible extension.

COMMENTS

History

Others—most notably Gorman and Anderson,[14] Schorr,[15] Franke,[16] Duley and Dietmeyer,[17,18] Friedman and Yang,[20] and Metze and Seshu[21]—have described languages and systems for logic translation or simulation, and occasionally for both. Typically, in logic translation systems, the design is described in a special-purpose procedural language similar to programming languages. The description is usually at a lower level than in CASD and is translated to Boolean equations, or some similar form, by programs written for the purpose.

In most simulation systems, on the other hand, designs are described in some high-level, general-purpose language—either a general simulation language, or an existing programming language augmented with timing subroutines and the like. The description is translated by an existing compiler to a program which performs the simulation.

There is good reason for this difference. Until recently, no existing programming or simulation language was really adequate to describe logic, and no general-purpose simulation system was so deficient as to justify creating a special system for simulating computer

designs. But the advantages of integrating logic translation and simulation into the same system outweigh these factors, in our judgment.

Integration of the two functions is achieved in CASD by translating a single, high-level, special-purpose language to a common internal form, providing input to both logic translation programs and an interpretive simulator. The interpretive simulation is also a key point in making the system on-line.

Another innovation in CASD is the way in which descriptions incorporate timing. Timing is included rather explicitly in typical existing languages. At lower levels, every statement or action is accompanied by an indication of when it is to take place (at which clock pulse, say). At higher levels, actions are simply recorded sequentially, with some indication of how long they take and what resources they require. (Simulators operating from these descriptions usually construct "future events" lists, ordered by increasing time of occurrence, and simulate whichever event is on top of the list at the moment.)

Timing in the CASD descriptions is based on the use of asynchronous design as proposed by Metze and Seshu.[22] Multiple tasks are synchronized by using shared variables and referring to them with WAIT statements. This approach has several advantages. Asynchronous design at the functional level, as offered by the CASD system, allows reasonable hardware independence, since synchronizing conditions refer to elements of the functional design rather than to its physical implementation. (An asychronous description may, of course, be implemented in either synchronous or asynchronous logic circuits.) Perhaps most important, especially for an on-line system, is that the PL/I multitasking scheme, from which the CASD timing approach is derived, and techniques like DO CONCURRENTLY make it possible to describe timing relationships in a quick and natural manner.

Advantages of an on-line system

Conventional design work is slowed by turn-around time (in the model shop as well as in the computation center) and an elaborate hierarchy of system architects, engineers, and technicians. One result is that few alternatives are considered in designing a system, and fewer still are evaluated in any systematic way. The CASD system bypasses these limitations by putting the designer directly in touch with a design system by a terminal, having the system take over many of the bookkeeping functions of design, and giving him immediate feedback at each stage of the design process. Immediate feedback is important in:

a. Encoding, where descriptions are entered line by line, and syntax is checked immediately, allowing immediate correction and modification.
b. Simulation, in which the designer may "converse" with the system as his design is simulated. He may change control conditions as the simulation progresses, look at values of data items, and so forth.
c. Selection of different translation procedures based on the results of simulation, cost estimating programs, or other translations.

Except for (a), these could be done with a batch system, of course, but they are much more effective in an on-line environment. Suppose, for example, that a design for a computer is stored in the system, and it contains special hardware for floating point operations. The designer wants to know just what difference it would make if he eliminated this hardware and did all floating point operations with programmed subroutines. With the CASD text-editing programs, the designer would remove the description of the hardware for floating point, and change the floating point operation code descriptions to trap these operations to a specified location. He would re-encode the description and correct any errors. By simulating and translating both this new description and the old one, he would obtain precise figures on the exact difference in hardware and running time. An on-line system can reduce this complicated maneuver to a one-day job.

Advantages of an integrated system

Most of the advantages of integrating all aspects of design in a single system can be summed up in one word: control. Consider how important it is that the simulation model accurately reflect the hardware that is being built. Under the CASD system, this is automatic: the design description is the simulation model.

A necessary part of the design process is low-level checking of logic circuits both for logical correctness and for race and hazard conditions. In CASD, the system always uses proven methods. Besides reducing the necessary tests, this controlled logic synthesis ensures the use of standard techniques and building blocks. Different optimality criteria can be used and the results compared. For example, the different effects of restricting the logic to one chip type, or allowing more freedom, might be compared. Criteria such as these are often more important than minimizing the

total number of circuits; and under the CASD system, the correct criteria can be enforecd.

A good design must be reliable and allow ready diagnosis of problems that occur. The CASD controlled synthesis ensures that the resulting logic is diagnosable. Indeed, the required diagnostic tests can be produced as an integral part of the translation process by at least one method.[23] It is easy to see how translators could be made to produce either duplicated logic, triple-modular-redundant logic, or unduplicated logic (say) if the designer wants to compare their relative costs.

Finally, the advantages of a unified file system, providing documentation automatically, are fairly clear. Accurate, consistent, up-to-date documentation may be the most important single feature of the CASD system.

EXAMPLE

This section contains an example of a computer described in the CASD design language. The computer and the way it is described have been chosen to illustrate the features of the language, rather than for any intrinsic merit. The computer is a simple binary machine called SYSTEM/0. It contains 65,536 32-bit words of memory and 16 general-purpose and index registers called XREG(0) through XREG(15). XREG (0) always contains all zeros. It may be stored, tested, and shifted, but not altered.

The instructions of SYSTEM/0 are one word (32 bits) long. The first 8 bits contain the operation code. The next 8 bits contain two four-bit fields, the M (for modifier) and X (for index register) specifications. The last 16 bits are used for an address.

The following instructions are described in the following CASD description:

ST	M,X,ADDR	Store the contents of XREG(M) into memory location [ADDR+ contents of XREG(X)].
CLA	M,X,ADDR	Load the contents of memory location [ADDR+contents of XREG(X)] into XREG(M). M may not equal zero.
BC	M,X,ADDR	Branch to location [ADDR+ contents of XREG(X)] if and only if the contents of XREG (M) is zero. (Since XREG(0) is always zero, BC 0, X,ADDR is an unconditional branch.)
RR	M,X,ADDR	Rotate XREG(M) right [contents of XREG(X)+ADDR] places. The number of places to rotate is always assumed to be modulo 32.
BAL	M,X,ADDR	Branch and Link to location [ADDR+contents of XREG(X)] storing the return address (= next location) in the low-order 16 bits of XREG(M), setting the high-order 16 bits of XREG(M) to zero. M may not equal zero.
SIO	M,X,ADDR	Start an input-output operation on device number [ADDR + contents of XREG(X)]. The M field specifies which input-output operation is to be performed.

Figure 2 shows the data flow the designer might expect CASD to generate, after entering the functional description given in Figures 3 through 7. (The order of the figures is for illustration only. The designer need have only a shadowy outline of the data flow in mind at the time he prepares his functional description.)

Figures 3 through 7 are annotated to highlight interesting features of the CASD language. Also note that there are a few places where the designer did choose to dictate the data flow. For example, the only link to the XREG's is constrained to be through the Y register by specifying Y:=MSDATA; XREG(M):= Y; rather than just XREG(M):=MSDATA; . So, the designer can exercise as much or as little direct influence on the final data flow as he chooses.

ACKNOWLEDGMENTS

We wish to thank George T. Robinson and Dr. Eugene E. Lindstrom for their guidance and advice.

REFERENCES

1 E D CROCKETT et al
Computer-aided system design
Advanced Systems Development Division IBM Corporation Los Gatos California Technical Report ※16.198 1970
2 *IBM System/360 PL/I reference manual*
IBM Corporation White Plains New York Form C28-8201
3 CROCKETT Appendix A
4 IBM Corporation page 237
5 CROCKETT Appendix G
6 IBID Appendix H
7 IBID Appendices I, J, K

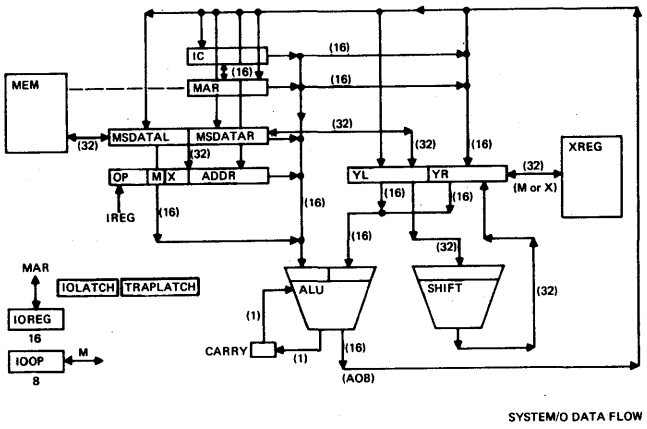


Figure 2—Flow of data in SYSTEM/0

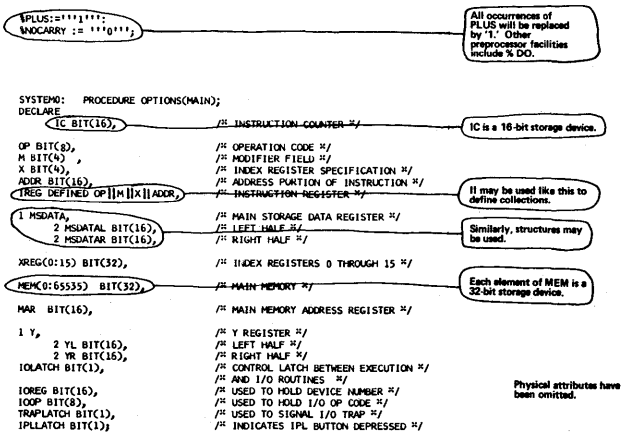


Figure 3—CASD description of SYSTEM/0, page 1

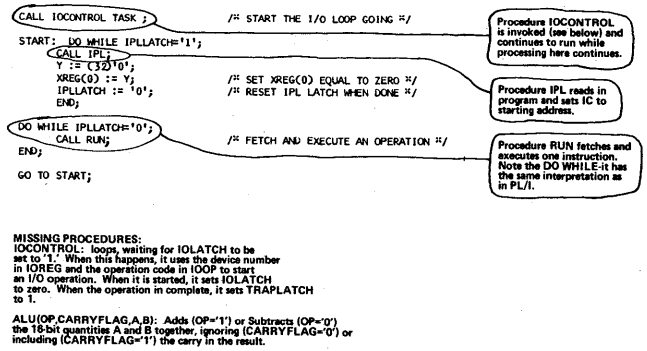


Figure 4—CASD description of SYSTEM/0, page 2

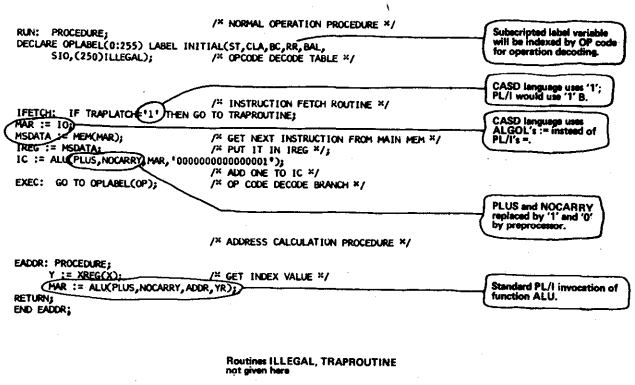


Figure 5—CASD description of SYSTEM/0, page 3

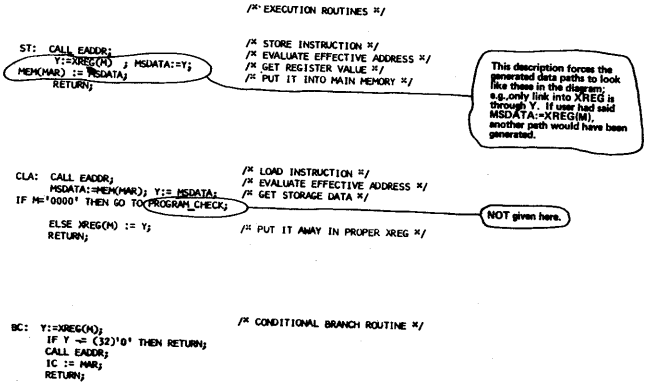


Figure 6—CASD description of SYSTEM/0, page 4

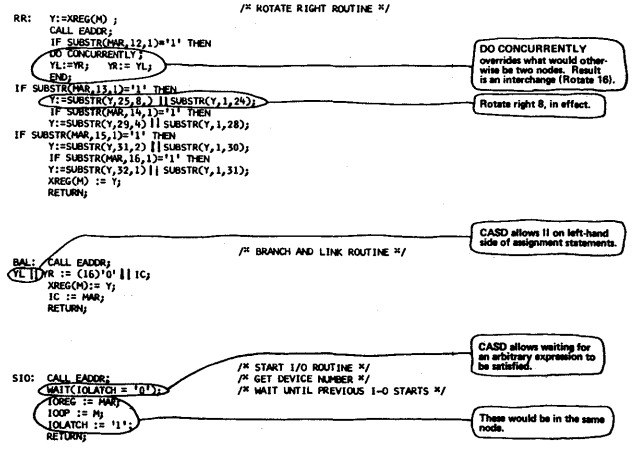


Figure 7—CASD description of SYSTEM/0, page 5

8 P BRYANT
A note on designing incremental compilers
Submitted to CACM August 1970
9 J R DULEY D L DIETMEYER
Translation of a DDL digital system specification to Boolean equations
IEEE Transactions Vol C-18 April 1969
10 T D FRIEDMAN
ALERT: a program to produce logic designs from preliminary machine descriptions
Research Division IBM Corporation Yorktown Heights New York Research Report No RC-1578 March 1966
11 T D FRIEDMAN S C YANG
Methods used in an automatic logic design generator (ALERT)
Research Division IBM Corporation Yorktown Heights New York Research Report No RC-2226 October 1968
12 D F GORMAN J P ANDERSON
A logic design translator
Proceedings AFIPS Fall Joint Computer Conference 1962
13 CROCKETT Appendix M
14 GORMAN and ANDERSON
15 H SCHORR
Computer-aided digital system design and analysis using a register-transfer language
IEEE Transactions Vol EC-13 December 1964
16 E A FRANKE
Computer-aided functional design of digital systems
IEEE Southwestern Conference Record April 1968
17 DULEY and DIETMEYER *op cit*
18 J R DULEY D L DIETMEYER
A digital system design language (DDL)
IEEE Transactions Vol C-17 September 1968
19 FRIEDMAN
20 FRIEDMAN and YANG
21 G METZE S SESHU
A proposal for a computer compiler
Proceedings AFIPS Spring Joint Computer Conference 1966
22 IBID
23 CROCKETT Appendix N

Integrated computer aided design systems

by ROGER C. HURST

North American Rockwell
Los Angeles, California

and

ALLEN B. ROSENSTEIN

University of California
Los Angeles, California

INTRODUCTION

Computer Aided Design, the initial phase of Engineering Design Automation, has been characterized by the development of individual computer programs for engineering design tasks. These programs specifically describe design tasks that are identified as repetitive in nature and, as such, are appropriately interspersed into the overall design process.

Design Automation is now entering a new phase characterized by the need for an integrated design approach to the modeling, synthesis, analysis, optimization, documentation, and production functions of entire engineering projects and of complete engineering fields. Concurrent with this need has come a recognition of the generalized Morphology and Anatomy of the Design Process which in turn has led to a rapid development of formal procedures for many of the basic functions of the Design Matrix (Figure 1).[1] Since the Decision Making process which we call Design and its fundamental elements apply without restriction to all engineering activities, it is not surprising to find that very general computerized design programs are beginning to appear which can be applied to whole classes of engineering problems without regard to particular disciplines.

Computers have long been applied to the analysis phase of the design process with the great majority of existing engineering computer programs, in fact, devoted to analysis. A study of the limited number of generalized models available to the engineer and of the mathematics applicable to these models makes obvious the utility of an analysis language such as Fortran and the possibilities for further generalization.[2] But analysis and simulation are not the only phases of the Design Process undergoing change. In response to the increasing need to search for an optimum solution have come a number of optimization programs, and very recently, a powerful generalized optimization program (General Electric's AID)[3] that may be coupled to existing analysis routines to close the Design Process loop, Figure 2.

The statement that the digital computer offers extended assistance in many if not most stages of the Design Process presents nothing new. We should like to examine, however, some of the emerging advances on the road from Computer Aided Design to Design Automation. It should be obvious that we are now ready to seriously consider Integrated Computer Aided Design systems that tie together individual stages of the design process, many of which have already been programmed.

INTEGRATED COMPUTER AIDED DESIGN—ICES AND ELECSYS

The requirements for an Integrated Computer Aided Design system can be outlined as follows:

1. *Generality* must be extensive enough to provide a substantial design capability for a complete class of problems or major areas in engineering fields such as Civil Engineering or Electrical Engineering. System capabilities would of a necessity include generalized optimization routines and strategies, extensive data handling and file capacity, and solution of simulation programs as well as comprehensive analytical capability—all to operate upon a common data set.

Anatomy of Design / Morphology of Design	Problem Statement Needs Formulation	Information Collection Organization Retrieval usage	Modeling Parameters Variables	Value Statement Criteria Constraints	Synthesis of Alternative Physical Realizability / Economic " / Financial " / Producibility / Reliability / Maintainability	Analysis and/or Test	Evaluation	Decision	Optimization	Iteration	Communication Implementation
Needs Analysis											
Feasibility Study											
Preliminary Design											
Detailed Design											
Production											
Distribution											
Consumption											
Retirement											

Figure 1

2. *Flexibility* is required to allow selection and coupling of preferred programs and the substitution of data at will with negligible effort.
3. *Problem Oriented Languages* that allow the engineer to communicate with the computer in the same engineering language that he would use with fellow engineers are a necessity if the Integrated Computer Aided Design system is to earn wide user acceptance.
4. A *Programming* mechanism should be available to allow the insertion of new design routines or the modification of existing programs with a minimal investment in programming time.

In spite of the identification of the computer with electrical and electronic engineering, and the early extensive use of computers for the design of electrical components such as transformers and electronic subsystems including computer subsystems, the first operational Integrated Computer Aided Design system has appeared in Civil Engineering with the MIT developed **I**ntegrated **C**ivil **E**ngineering **S**ystem (ICES).[4] It is the intent of this paper to briefly describe ICES and then obtain some review of its capabilities by describing a new Integrated **ELEC**tronic Engineering **SYS**tem (ELECSYS) that utilizes the basic ICES concepts and programs for internal control of the computer software and hardware.

Generality of the Integrated Computer Aided Design concept will be demonstrated by the conversion from the Civil Engineering environment of ICES to the Electronic Engineering of ELECSYS. *Flexibility* and *Programming* capability are explored by utilizing an MIT modified version* of ICES as the programming structure to activate the first subsystem of ELECSYS.

* Modified to drop off Civil Engineering subsystems and add programming routines to simplify the task of adding new subsystems.

In this case, we have modified an existing **E**lectronic **C**ircuit **A**nalysis **P**rogram (ECAP)—the DC analysis portion only—to reflect the external subsystem requirements of ICES. Neither ICES or ELECSYS can ever be considered complete systems in that they have been deliberately designed for the almost unlimited addition of engineering subsystems as desired.

ICES

Let us briefly examine the capabilities and structure of ICES while reserving a more detailed explanation for Appendix A.[5]

ICES has been designed to allow engineers with little or no computer programming experience to apply the computer to a wide range of engineering problems. The user can view ICES as a set of engineering subsystems in a particular discipline of engineering such as structural analysis, soil mechanics, etc. , . . . , which the user can call upon individually or collectively to solve his problems. Each subsystem has its own problem oriented language that is derived from the natural language of the discipline.

ICES is a modular, dynamic system that also has been designed to simplify the programmer's task. Programming languages have been incorporated to enable the programmer to readily make modifications and additions. To provide a full ICES system capability, a computer will ultimately contain a total of six languages or perhaps we should say six language levels if we count the final machine language itself (Job Control Language, FORTRAN, ICETRAN, Command Definition Language, and Problem Oriented Language). However, the user need only master the simple problem oriented languages of the subsystems and disciplines that interest him. He communicates with a subsystem only through the commands of the subsystem **P**roblem **O**riented **L**anguage (POL).

The subsystem programmers, on the other hand, will

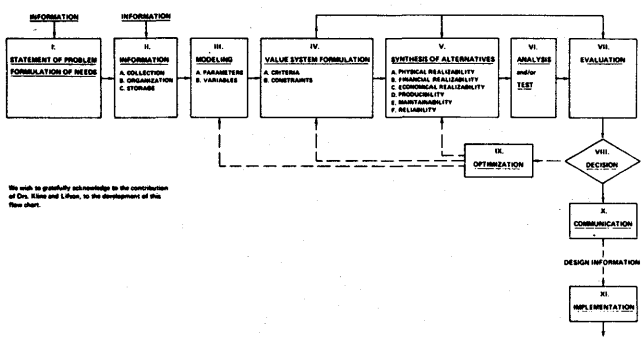


Figure 2—The design-planning process flow chart

require two additional languages besides Fortran. The first which has been called the **C**ommand **D**efinition **L**anguage (CDL) is used to write a program for each subsystem to interpret each command in the POL and provide direction for the processing of each command. The CDL specifies what information should be provided in the engineer's commands, how and where that information should be stored, and what operations (ICETRAN programs) shall be used to perform the desired operations upon the data.

ICETRAN, the second language of the subsystem programmer, is an extension and expansion of FORTRAN that is used by the subsystem programmer to develop the ICETRAN building block programs that

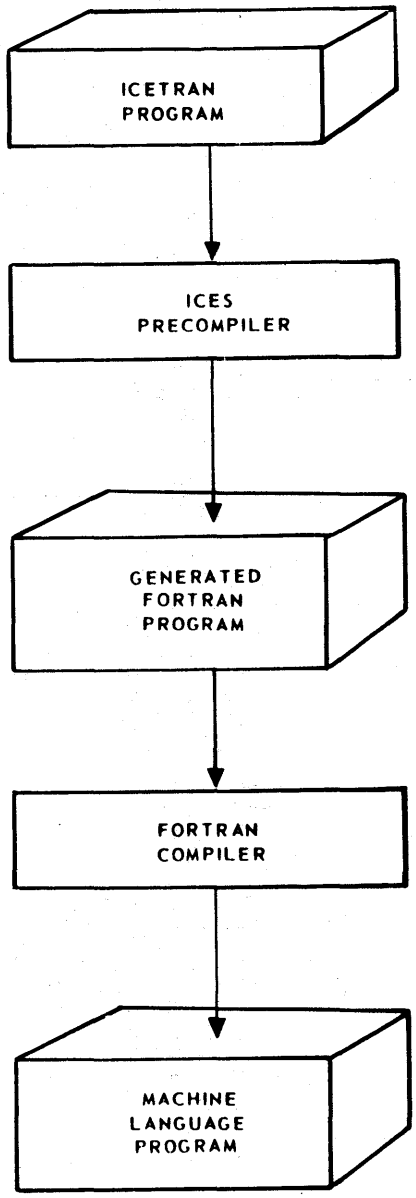


Figure 3—ICETRAN precompilation and compilation

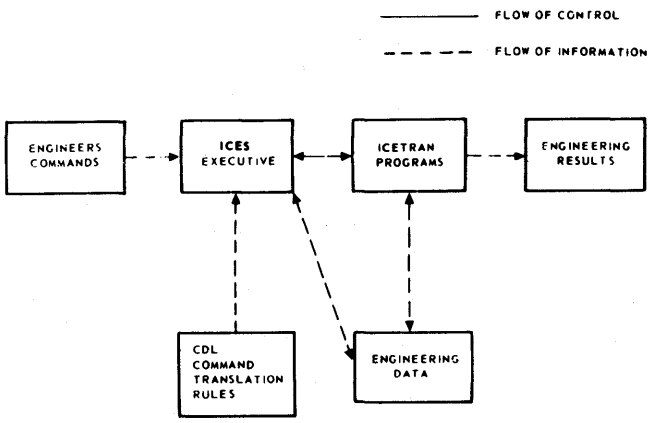


Figure 4—Subsystem execution

operate upon the engineering data. To facilitate the translation from ICETRAN to FORTRAN and finally to machine language, a special translator program is provided called the ICETRAN Compiler, Figure 3.

Operation of an integrated ICES type system can be partially explained and the remaining ICES system elements introduced by referring to Figures 4, 5, and 6. Stored in computer primary memory, we find the Execu-

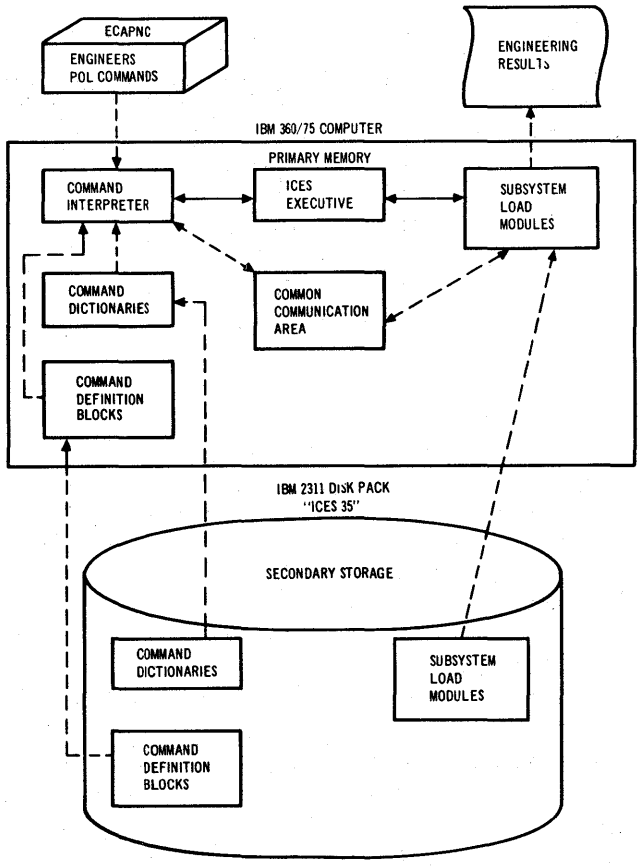


Figure 5—ICES system organization

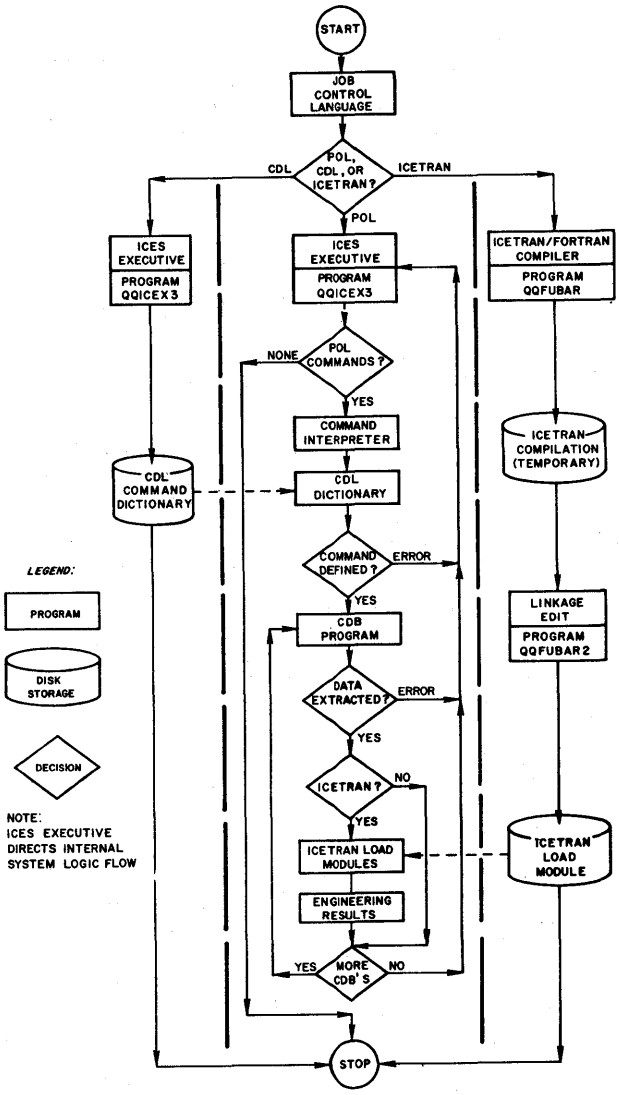


Figure 6—System logic diagram

tive program that supervises and coordinates all ICES jobs. The multi-subsystem capability of ICES demands a dynamic data structure and extensive use of secondary storage. Stored in secondary memory for each subsystem are Subsystem Load Modules (constructed from ICETRAN building block programs), Subsystem Command Dictionary, and Subsystem Command Definition Blocks.

The engineer's POL commands are directed to an ICES Command Interpreter that proceeds on a command-by-command basis. From a primary command dictionary, the first subsystem requested by the engineer will be identified and the corresponding Command Dictionaries, Command Definition Blocks, and Load Modules called up from secondary storage by the Executive program. The Command Interpreter can now refer to the subsystem command dictionary to process subsequent commands. Each subsystem command calls for a specific Command Dictionary Block which the Command Interpreter uses to analyze the full command and transmit the correct information to the required subprograms.

Data and information extracted from the commands and the command definition blocks are stored in the subsystem Common Communications Area where they are available for processing by the ICETRAN programs of the subsystem load modules.

ELECSYS

Keeping in mind the built-in provisions for expansion and change, let us examine the requirements for an Integrated Electronic Engineering Design System (ELECSYS) and its implementation, Figures 7 and 8. We wish to provide in a single computer aided design system all of the analytical methods, design rules, manufacturing standards, component characteristics, tabular data, simulation routines, and optimization

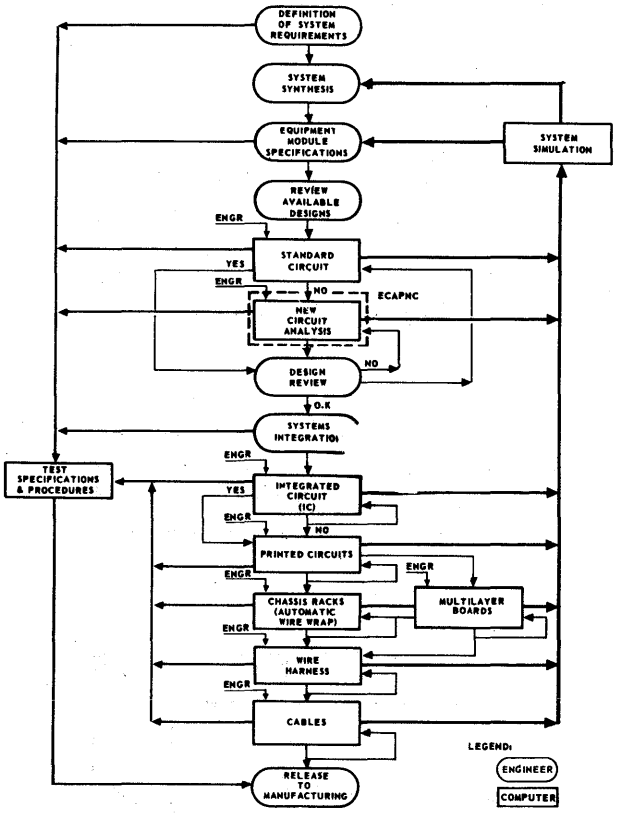


Figure 7—ELECSYS system flow diagram

programs necessary for the computer aided design of electronic equipment. All of the foregoing should be available to the user in problem oriented languages with a common vocabulary and similar formats. It should be noted that ELECSYS would be classified as a computer aided design system instead of design automated since the engineer will intervene at frequent intervals during the design process to establish desired system behavior.

ELECSYS, Figure 7, has been laid out to follow the iterative decision making pattern of the design process, Figure 2. Design normally progresses from problem recognition to problem definition where design requirements, constraints, and performance specifications are established. A field of feasible system solutions is next synthesized which in turn can be broken down into subsystem, module, and finally circuit specifications. At some function level, it becomes convenient to review the applicability of existing designs.

Electronic subsystems readily lend themselves to classification and evaluation, i.e., amplifiers, flip flops, shift registers, etc. ELECSYS would include an extensive library of proven circuits and designs evaluated in accordance with standard performance/cost criteria. Full listings would also be provided of standard components along with their cost and performance characteristics. As the analytical capabilities of ELECSYS are expanded, it has been suggested that a stage will eventually be reached when the computer can use its off hours to read the technical literature in order to independently evaluate published circuits and update its own library.

Both the engineer and the computer would review their respective libraries for standard circuits capable of meeting the required specifications. The ELECSYS Standard Circuit subsystem would provide the engineer with a thorough computerized search of existing designs to meet specific performance specifications and design criteria. If an acceptable circuit is not found, the engineer will design a new circuit with the aid of the computer circuit analysis programs. Final detailed electronic designs are subject to many design reviews to insure system performance and reliability. System stress and reliability analysis can be done by the computer along with system simulation.

Since a common circuit data base is necessary for both electrical and mechanical design, ELECSYS has been conceived to provide POL routines for both the electronic design and much of the mechanical design required for manufacturing. After the electrical performance requirements are satisfied, the resultant circuits are re-examined for fabrication. A set of programs should be incorporated to carry manufacturing from printed and integrated circuit layouts through the

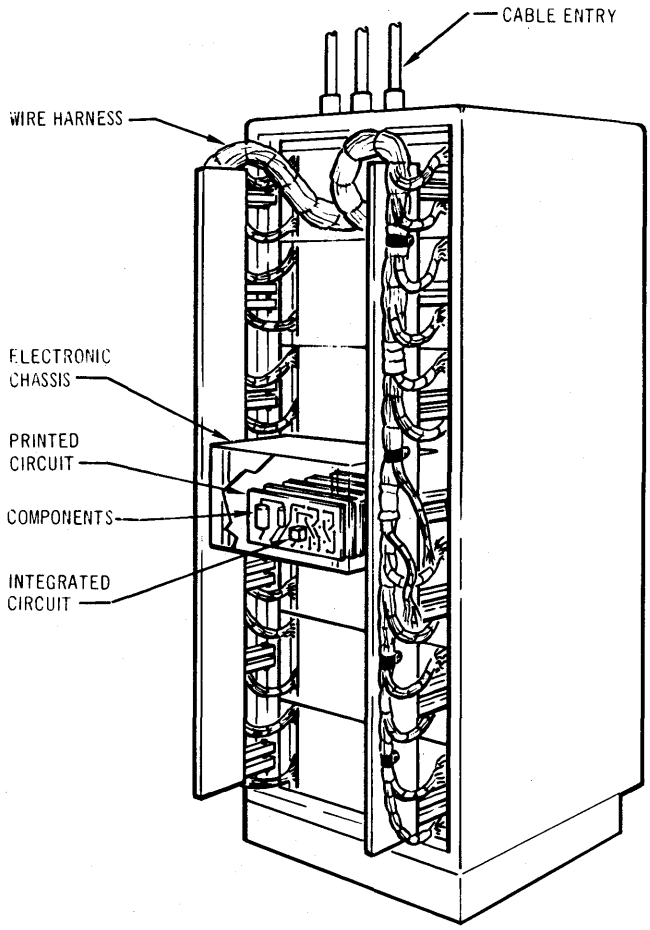


Figure 8—Electronic equipment

wire harnessing and cabling necessary to interconnect circuits and components.

A very general purpose optimization program that can be coupled to analysis routines (such as G.E.'s AID) is to be included to provide computer assistance in the determination of improved circuit and electromechanical system design.

INITIAL IMPLEMENTATION—ECAPNC

ELECSYS has been conceived as a flexible integrated electronic design system that incorporates as subsystems existing or new computer programs in part or in wholew herever desirable. To test the feasibility of this concept, we decided, as mentioned earlier, to build IBM's ECAP (**E**lectronic **C**ircuit **A**nalysis **P**rogram) into ELECSYS as a subsystem for **N**ew **C**ircuit analysis, ECAPNC.

Implementation of ELECSYS began with the acquisition from IBM of the ICES Type C Release version containing all system programs necessary for ICES

utilization including the programming capability for adding new subsystems. Modification of the ICES Job Control Language was required for the installation of ICES on the UCLA Campus Computer Network.

With ICES operational on campus as "ICES35" five major steps were taken to implement and verify ECAPNC. The first three tasks were concerned with the *computer software needed by an engineer setting up a subsystem.* The next step employed the *computer software provided to an engineer using a subsystem* while the final step became that of *operational verification.* Thus, the first three tasks dealt with subsystem generation whereas the last two specifically relate to subsystem utilization in a problem oriented language environment. Typical subsystem installation and operation can be presented by reviewing those five tasks.

1. *Precompilation, Compilation, and Object Load:* Programs are written in ICES FORTRAN (ICETRAN) by the engineer designing an engineering subsystem which temporarily creates and stores computer programs for engineering operations. This task utilizes the ICES procedure "Precompile, Compile, and Object Load" (Refer to Figure 3) to process the written ICETRAN programs in preparation for the next task of Load Module Generation.
2. *Load Module Generation:* This task employs the ICES procedure called "Load Module Generation" to combine or link edit the ICETRAN programs of the previous task into Subsystem Load Modules that satisfy the computer hardware requirements. The Subsystem Load Modules are computer programs permanently stored for the performance of engineering operations.

To initiate ECAPNC, the DC Analysis portion of the standard ECAP program was modified in accordance with tasks 1 and 2. In particular, six subprograms were programmed in ICETRAN and added to the original DC Analysis subprograms in order to provide the final subscripted variable translation of ECAP problem data. ICES CDL software cannot translate POL data directly into Fortran subscripted variables, thus, these extra ICETRAN programs provide the necessary processing of ECAP compatible data. The total package of ICETRAN subprograms (1400 cards) were processed by tasks 1 and 2 to establish the engineering load module for the ECAPNC subsystem.

3. *Subsystem Command Definition Execution:* Utilizing the CDL of ICES, a subsystem vocabulary must be developed to provide the **P**roblem **O**riented **L**anguage (POL) required for task 4. Permanent subsystem computer programs must be created to interpret the engineer's English language (POL) input and select the appropriate building block programs for execution. ICES CDL procedures are used to create a Subsystem Command Dictionary and the individual Subsystem Command Definition Block associated with each command in the subsystem dictionary.

Command **D**efinition **L**anguage (CDL) programs were designed to translate the Problem Oriented Language commands of ECAPNC, extract the problem data, and convert that data into the appropriate ICETRAN variables to be operated upon by the ICETRAN subprograms for DC Analysis (task 2). A total of 400 cards were required to generate the Subsystem Command Definition Dictionary and its Command Definition Blocks for the ECAPNC subsystem.

4. *POL Program Execution:* An individual program must be written for each new circuit requiring a DC performance analysis with ECAPNC. However, the POL provided by task 3 completely routinizes this step making it feasible to train technicians or secretaries to convert the engineer's schematics into the required punched cards or tape.
5. *Circuit Execution:* With a program written for the desired circuit topology, the program can be submitted to a computer with the initial values of individual components and the allowable range of component values to be investigated. The ELECSYS system presently utilizes batch computer processing techniques to obtain solutions of circuit problems. Since the POL is conversational in nature, this is an unnecessary restriction. Future systems will utilize a remote CRT terminal or teletype to improve ICES man-machine communication as well as job turnaround time.

To verify the successful implementation of ECAP in the ELECSYS subsystem, a sample electronic circuit was run through tasks 4 and 5 of ECAPNC and compared with the results of submitting the same circuit to the original ECAP program. The results were, of course, identical.

The detailed explanation of the necessary ICES modifications and required ECAP conversion under tasks 1, 2, and 3 is given in Reference 7. In Appendix B, the POL of task 3 is described and its application to a simple circuit illustrated for tasks 4 and 5.

RESULTS AND CONCLUSIONS

This work was undertaken to obtain operational experience with Integrated Computer Aided Design Systems and to provide a basis for consideration of further automation of the Design Process. We wished in particular to test the concept of a very generalized Integrated Computer Aided Design System that would be independent of its field of engineering application. Also, although ELECSYS has been proposed (Figure 7) with human intervention at every major step, it should be apparent that further formalization of the engineer's decision rules and data resources will allow significant reductions in the number of unmechanized phases. This, of course, would appear practical only if the system can operate upon a common design problem data base that is derived from the original problem statement.

Allowing for the present limited ELECSYS implementation, our experiences to date have been quite satisfying. No unusual difficulties were encountered in utilizing the structure of ICES, designed as an Integrated Civil Engineering System, for ELECSYS, an Integrated Electronic Engineering System. The feasibility and economics of incorporating existing design subprograms into an integrated design system has been clearly established. Once the details of ECAP and ICES had been mastered, the programming chores required to generate the load modules (tasks 1 and 2) and the language translator (task 3) were found to be quite reasonable.

The ECAP DC Analysis program was converted to ICETRAN in two steps. The first step required a compatibility study of the language interface between ECAP FORTRAN and ICES FORTRAN that resulted in modification of several ECAP FORTRAN statements which would have created ICES system errors. A second step was required to produce a set of subprograms necessary to overcome a subscripted variable limitation of the ICES Command Definition Language. Completion of both steps produced a permanently stored set of DC Circuit Analysis subprograms. Utilization of the existing ECAP routines saved, of course, the substantial effort that would have been required to create a new circuit analysis program.

It should be observed that problem oriented language programs such as ECAP, STRESS, CIRC, and STRUDL in reality solve the same general class of problem. In each of these cases the computer is given the problem topology and component lumped parameter characteristics. With this data, the computer sets up and solves the appropriate matrix. It should be feasible, therefore, to design into the Integrated Design System a very general, lumped parameter, equation writing, and analysis routine that would provide suitable load modules and eliminate the need to repeat tasks 1 and 2. When general, lumped parameter, load modules have been created for an integrated design system, incorporation of engineering fields utilizing lumped parameter models will require only the production of the problem oriented language from task 3.

Utilization of the ICES Command Definition Language to generate the CDL subprograms for ICETRAN/POL interface control and a set of CDL subroutines for data extraction and storage (task 3), gave particularly interesting results. The programming effort created a total of nine subprograms and five subroutines with a final program source deck of 370 CDL cards. A significant difference in programming effort and time is realized when the 370 cards of the CDL program are compared to the 1610 Fortran cards required by the original ECAP Language Processor to perform the same language translation. This comparison would indicate that the ICES CDL provides a simplified programming capability for data translation of complicated POL such as ECAP. With the adoption of Integrated Computer Aided Design Systems somewhat dependent upon the programming investment, the efficiency of the CDL will be of increasing importance particularly if generalized load modules can be developed to reduce the programming effort of tasks 1 and 2.

Review of ECAPNC revealed that the POL and flexibility criteria were satisfied. The POL notation of ECAP was retained with small modification. However, whereas the original ECAP language requires text in a fixed order as well as a rigid format that tends to increase language programming errors, the ECAPNC ordering of the text supplying problem data is very flexible. The integrated system concept offers still further opportunity for increasing the flexibility and utility of ECAP. Recently while reviewing ELECSYS with Mr. Howell Tyson of IBM, who is one of the originators of ECAP, Tyson pointed out the advantages that could be obtained by interfacing the ECAPNC subsystem with the existing ICES TABLE subsystem.[6] A permanent file of electronic circuit components, characteristics, particularly transistor models, could be maintained by the TABLE subsystem and utilized as required by the ECAPNC subsystem in conjunction with the ICES system programs. By combining the TABLE capabilities with ECAPNC, a more powerful circuit analysis program could be achieved with circuit lookup capability similar to the NET-1 program.

On the negative side, the average time for four ECAPNC problem solutions was 21 seconds compared to an average of 11 seconds for the identical solutions with ECAP. The difference is caused by the separate

processing that is performed by the CDL when extracting problem data. For offices with a large circuit analysis load, the differences in computing time could become prohibitive. On the other hand, when analysis is considered as only part of the computational load of the design process, the advantages of the common problem data base of the Integrated Computer Aided Design System should more than compensate for the analysis computing inefficiencies.

BIBLIOGRAPHY

1 A B ROSENSTEIN
The concept of engineering design as a formal discipline
Proc ASME 8th Annual Technical Symposium Albuquerque New Mexico November 17 1967

2 A B ROSENSTEIN
Modeling, analysis, synthesis, optimization, and decision maker in engineering curricula
UCLA Department of Engineering EDP2-68 April 1968

3 J K CAREY R L RUSTAG
AID—A general purpose computer program for optimization
Recent Advances in Optimization Techniques edited by Love & Vogl 1966

4 ———
Introduction to ICES
Department of Civil Engineering Massachusetts Institute of Technology Cambridge Massachusetts Report R67-47 C 1967

5 ———
ICES System—General description
Department of Civil Engineering Masachusetts Institute of Technology Cambridge Massachusetts Report R67–49 C. 1967

6 ———
ICES Programmers Reference Manual
Department of Civil Engineering Massachusetts Institute of Technology Cambridge Massachusetts Report R67-50 C. 1967

7 R C HURST
The feasibility of an electrical/electronic engineering computer system (ELECSYS) which utilizes the integrated civil engineering system (ICES)
University of California at Los Angeles 405 Hilgard Avenue Los Angeles, California 90024 Master's Thesis C 1969

8 ———
1620 electronic circuit analysis program (ECAP)
1620-EE-20X user's manual
International Business Machines Corp Technical Publications Department C 1965

APPENDIX A

INTEGRATED CIVIL ENGINEERING SYSTEM (ICES)

The ICES system is now ready to perform its prescribed system tasks (Refer to Figure 6). These tasks are externally controlled by the IBM 360 Job Control Language (JCL) and internally by the ICES System Programs. The external control process provides information to the computer system concerning the ICES data flow between the IBM 360/75 central computer and the IBM 2311 secondary storage disk pack. Internal control is provided by an appropriate ICES system program for the complete performance of a related system task. External and internal controls are combined for each system task in order for the system to be fully operational.

The following paragraphs illustrate the ICES control process for the first four system tasks. The fifth system task concerns circuit solutions and the design iteration process of the engineer.

The first system task for subsystem implementation concerns the translation of ICETRAN programs into machine language (Figure 3). The Job Control Language directs information flow between computer hardware units such as secondary storage devices (disk pack), and tape units. After appropriate units have been accessed, program QQFUBAR, internal controlling program designated by the Task 1 JCL, is executed and successively linked to Program QQVOLO, the ICETRAN precompiler; to Program IEJFAAAO, the E-level FORTRAN compiler; and to Program QQNIX3, the object program loader.

The second system task for subsystem generation concerns the permanent creation and storage of the machine coded programs from Task 1. After similar job initialization and secondary storage accessing, the JCL designates the Program QQFUBAR2 to provide internal control and successively link to Program QQSETGEN, the load module interface program generator; and program ASMBLER, the E-Level Linkage Editor. Program QQSETGEN generates an interface program called QQSETUP which is tailor-made for every subsystem load module, and is used whenever the generated load module is entered for the performance of Task 4, Problem Oriented Language (POL) Program Execution. In particular, the Load Module Description input for this task provides the necessary information to generate a unique QQSETUP program.

The third and fourth system tasks have radically different functions. Task 3 involves the subsystem generation of Command Definition Language dictionaries, while Task 4 utilizes the data generated from the previous tasks to perform overall subsystem activities for the solution of engineering problems.

Both tasks utilize Program QQICEX3, the internal controlling ICES executive. For the engineer setting up a subsystem, the ICES Executive program will process and store Command Definition Language programs. For the engineer using ICES in a problem solving mode, the ICES Executive program will

process Problem Oriented Language commands as well as control input/output, secondary storage file management, core memory allocation, and program loading of subsystem load modules and command definition dictionaries.

The POL Execution sequence starts with the initialization of the ICES Executive program. This interrogates the input data for the existence of POL commands. If there are no commands, the program terminates. If commands are present in the input data, the ICES Executive calls the Command Interpreter and an appropriate Subsystem Command Definition Block (CDB) and Dictionary. When a POL command is not defined in the CDB, a POL error is given by the ICES Executive program. If the POL statement is a valid command, an appropriate Command Definition Language program is directed to extract problem data from the POL command. Unless errors occur in the data or data identifiers, the ICES Executive directs further processing of the Command Definition Language program which may include either more CDB's or the execution of CDL designated ICETRAN programs. When an ICETRAN program is performed, engineering results are calculated and tabulated for engineering analysis. At this point, the ICES Executive can either process more CDB programs or more ICETRAN programs; or return to process subsequent POL commands. After all POL commands have been processed, the ICES Executive terminates the specific problem solution.

It can be readily seen that there is no limit to the number of subsystems that can be executed. Each subsystem is started when the POL contains a system command that specifies the subsystem name. The presence of this command causes the previous subsystem, if any, to be closed and the specified subsystem to be processed by the ICES Executive program. The case of any one computer run is specified by the system command FINISH, which causes the last subsystem to be closed, as well as the ICES system itself. Thus, for the first time, an integrated system concept is evolving for the solution of engineering problems as complete discipline, rather than the present scheme of disassociated, fragmented computer programs.

APPENDIX B

THE ECAPNC PROBLEM ORIENTED LANGUAGE

Initial results from a generalized study of ECAP indicated that the Problem Oriented Language design should be similar in format to the original ECAP input language. This was advantageous for two reasons:

1. The POL design would utilize existing ECAP language notation for the identification of electronic circuit data. Thus, most of the data identifier information required in the design of the ECAPNC Command Definition Language programs was readily determined.
3. The instruction of engineers in the use of ECAPNC would not be substantially different than that of the original input language. Also, since ECAP is a widely used circuit analysis program, engineers already familiar with ECAP could easily convert to ECAPNC.

Based on the previous design criteria, the POL design was developed to incorporate all existing ECAP problem statements while satisfying related design constraints imposed by the design of the CDL and ICETRAN programs.

The completed design provides a detailed summary of usage instructions for the engineer utilizing the ECAPNC subsystem for DC circuit analysis.

BASIC COMMAND NOTATION

The engineer communicates problem data to the circuit analysis subsystem through a POL program consisting of a series of statements known as *Commands*. Each command utilizes the following two types of elements:

1. Alphanumeric words that are used as labels for commands, command modifiers, and data identifiers.
2. Integer or decimal numbers that make up the engineering problem data.

The alphanumeric words or single letters identify the command statements according to the rules set forth in the design of the ECAPNC Command Definition Language. Each word in a command statement uniquely specifies the problem data with respect to engineering terminology. Usually the first word in the statement defines the command, and subsequent words further describe and label the type of data required for problem solution. It is also emphasized that in addition to supplying problem data, the command words can instruct the subsystem to perform calculations on data specified in previous command statements.

After appropriately identifying the engineering

problem data, it must be specified in the following data modes:

1. Integer data
   These are numbers that do not contain a decimal point, or commas.
   Examples: 1 38 −1000 +10000
   Note: If the sign is omitted, it is assumed positive (+).
   The command notation for integer data is given by i_n, where n is a subscript that uniquely identifies integer data within a command statement.
2. Decimal data
   These numbers must contain a decimal point, but no commas. There are two basic types of decimal numbers, normal and exponential. The normal decimals consist of digits preceded by an optional sign.
   Examples: 1.0 38. −1000. +10000.
   The exponential numbers are decimals multiplied by the power of ten. They have the form of a decimal, but are followed by the letter E and a signed or unsigned integer representing the power of ten.
   Examples: $10.E-1$ $3.8E1$ $-1.0E3$ $+10E3$
   The command notation for decimal data is given by v_n, where n is a subscript that uniquely identifies decimal data within a command statement.

The command text is now defined as the logical meaning and ordering of the command elements (alphameric words, and integer or decimal numbers). For convenience and simplicity, the meaning and ordering of the command text is abbreviated through the use of special symbols such as underlines, braces, parentheses, and brackets. The following abbreviations are summarized for subsequent reference:

1. Underlines are used to indicate the required portion of an alphanumeric word in a command statement. For example, if the word INCREMENT is used in a command, the engineer need only give the letters INCR or any other word starting with these letters such as INCREASE.
2. Braces are used to indicate that a choice exists among the enclosed elements in the command text.
   Example:

$$\left\{\begin{array}{l}\text{SENSITIVITY}\\ \text{WORST}\\ \text{STANDARD}\end{array}\right\}$$

   The engineer may choose one of the three words within this command statement.
3. Parentheses are used to indicate that certain words may be omitted in the command text.
   Example:

   DC (SOLUTION)

   The engineer can specify the DC solution to a problem and add the word SOLUTION if clarity is desired. It is noted that if parentheses enclose the example for braces, the entire set of elements may be omitted.
4. Brackets are used to indicate the alphanumeric words that are used as labels for integer or decimal data.
   Example:

   [BRANCH] i_1 [NODES] i_2 i_3

   It is noted that an internal ICES programming capability allows the engineer to omit the labels and just insert the data in the order specified by the subscripts, or to give the data any order provided that they are labeled.
   Example:

   i_1 i_2 i_3

   [NODES] i_2 i_3 [BRANCH] i_1

   In the last example, it should be emphasized that i_2 and i_3 are a group of related data with the common label of NODES. Thus, this capability stipulates that problem data may be given in any order within the command text provided that the data is labeled.

In essence, then, the POL command text consists of alphanumeric words that define commands and label data in conjunction with special symbols that specify the language order and meaning.

GENERAL COMMANDS FOR THE ECAPNC SUBSYSTEM

The design of the ECAPNC Problem Oriented Language is organized into nine separate commands. Each command performs a unique function in the overall problem solution. For instance, some commands specify certain types of electronic circuit data while others dictate the type of analysis that must be performed on the specified data. This section provides a detailed explanation of all POL commands used in describing electronic circuit analysis problems to the ECAPNC subsystem.

The Problem Oriented Language for the ECAPNC

subsystem is summarized by the following command names:

1. Subsystem Specification (ECAPNC)
2. Solution Specification (DC)
3. Analysis Specification (SENSITIVITY, WORST CASE, STANDARD DEVIATION)
4. Circuit Description (DESCRIPTION)
5. Circuit Topology (INPUT)
6. Transconductance (TRANSCONDUCTANCE)
7. Execute Print (EXECUTE)
8. Branch Modification (MODIFY)
9. Transconductance Modification (TMODIFY)

It should be noted that an ICES capability in the Command Definition Language allows POL commands 3 through 6 to be specified in any order when included between the DC and EXECUTE commands. Thus, the POL commands are partially order-independent which enables the engineer to formulate problem solutions in a language environment oriented towards normal engineering practice.

In order to add clarity to the command descriptions, a sample electronic circuit will be used to derive problem data. The circuit schematic diagram (Figure 9)[8] shows the DC equivalent of a single-stage common emitter transistor amplifier. The transistor has been replaced at the Base, Emitter, and Collector nodes by a DC model containing a transconductance or current gain ($T1$). The circuit is prepared for computer solution by numbering each electrical branch (B) and corresponding electrical nodes (V). The electronic components that make up the circuit are given their respective precalculated values.

The circuit can be described as follows: In Branch 1, between Nodes 2 and 0 (Ground), there is a Resistor of 2000 ohms in series with a voltage source of 20 volts; In Branch 2, between Nodes 1 and 0, there is a Resistor of 6000 ohms in series with a voltage source of 20 volts; In Branch 3, between Nodes 1 and 0, there is a Resistor of 1000 ohms; In Branch 4, between Nodes 1 and 3, there is a Resistor of 350 ohms in series with a voltage source of 0.5 volts; In Branch 5, between Nodes 3 and 0, there is a Resistor of 500 ohms; In Branch 6, between Nodes 2 and 3, there is a Resistor of 11.1E3 ohms; and between Branches 4 and 6, there is an equivalent current gain ($1/h_{fe}$) of 50. This circuit problem is now defined in a form which the ECAPNC Problem Oriented Language can readily utilize.

The following commands are available to the engineer for communicating problem definitions to the ECAPNC subsystem:

The subsystem Specification command specifies the

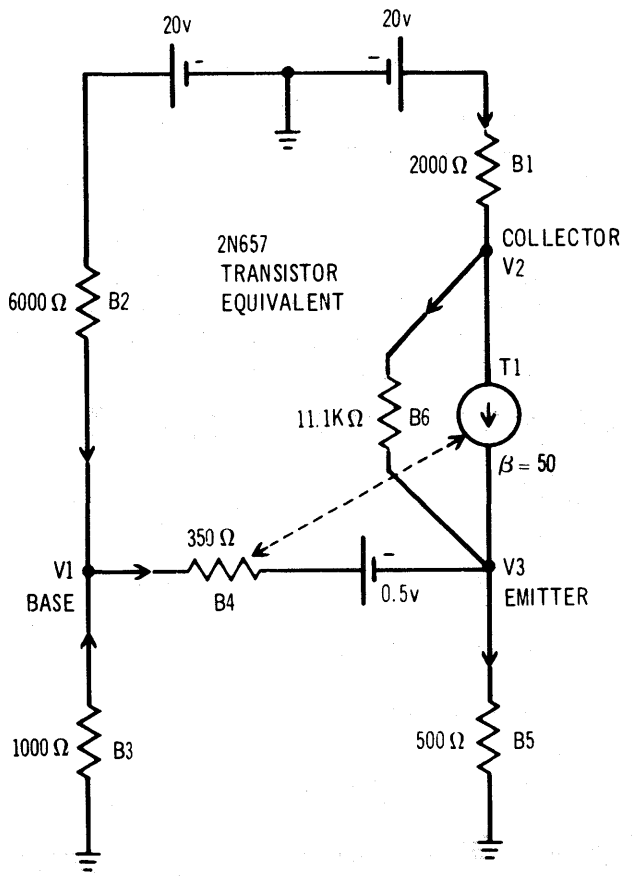


Figure 9—DC equivalent circuit for the single-stage common emitter amplifier

required ELECSYS subsystem and is the first command of any problem solution.

General form:*

ECAPNC**

Example:

ECAPNC

This is the name of the subsystem (ECAP New Circuit Analysis) that performs DC electronic circuit analysis.

The Solution Specification command is used to direct the ECAPNC subsystem to perform the nominal DC solution. It also prepares the subsystem for subsequent problem data, and therefore, must follow the subsystem Specification command.

General form:

DC (SOLUTION)

* A double asterisk, **, has been used to point out differences between ECAP and ECAPNC notation, and commands unnoted are identical. Also, this double asterisk in not part of the command notation.

Example:

DC SOLUTION

In the example, the word SOLUTION may be omitted from the command text. It is noted that any number of problems may be specified in one entry of the subsystem simply by starting each problem with this command.

The Analysis Specification command directs the ECAPNC subsystem to override the DC solution command and instead perform one of three available analysis routines. However, to reach this mode, the DC solution command must be given and followed by the desired Analysis Specification command. Without an Analysis command, a nominal DC solution will be provided.

General form:

$$\left\{\begin{array}{ll} \underline{\text{SE}}\text{NSITIVITY} & \text{(ANALYSIS)} \\ \underline{\text{WO}}\text{RST (CASE)} & \text{(ANALYSIS)} \\ \underline{\text{ST}}\text{ANDARD} & \text{(DEVIATION)} \end{array}\right\}$$

Example:

SENSITIVITY

The example shows that the subsystem is directed to perform the Sensitivity Analysis, which is a measure of circuit output stability when each circuit component is subjected to a one percent variation. The Worst Case Analysis could have been specified instead of the Sensitivity Analysis, and allows the engineer to obtain the minimum and maximum DC solutions assuming linear variation in component parameters. The Standard Deviation Analysis could also have been specified and would yield the three-sigma standard deviations for all electrical node voltages subject to the worst case component variations. The extreme parentheses indicate that all three analyses are optional, and do not have to be specified.

The Circuit Description command specifies the total dimensions of the circuit topology. In the sample problem, there are six electrical branches, three electrical nodes, and one current gain (also known as transconductance or dependent current source).

General form:

$\underline{\text{DESCRIPTION}}$ (TOTALS) [$\underline{\text{BRANCHES}}$] i_1

[$\underline{\text{NODES}}$ i_2 ([$\underline{\text{DCUR}}$] i_3)**

where i_1—integer number representing total circuit branches,**

i_2—integer number representing total circuit nodes,**

i_3—integer number representing total dependent current sources.**

Example:

DESCRIPTION TOTALS BRANCHES 6

NODES 3 DCUR 1

The engineer must specify the number of circuit branches and electrical nodes, otherwise, the DC solution cannot be calculated. The number of current gains is optional since some circuits may not contain transistors requiring this modeling technique.

The Circuit Topology command is used to specify the electrical topology (Branches and Nodes) as well as the component parameter data. This command is identified by the word INPUT which is followed by subcommands of similar format in a tabular arrangement. Each subcommand represents the complete circuit data for each of the electrical branches, and may be given in any arbitrary order as long as all branches are tabulated. The label ENDB is used to signal the end of tabulated circuit branch data.

General form (Tabular):

$\underline{\text{INPUT}}$**

$$\begin{array}{llll} [\underline{\text{B}}]i_1 & [\underline{\text{N}}]i_2 i_3 & \left\{\begin{array}{l} [\underline{\text{R}}]v_1(v_2 v_3) \\ [\underline{\text{G}}]v_1(v_2 v_3) \end{array}\right. & \left(\left\{\begin{array}{l} [\underline{\text{E}}]v_1(v_2 v_3) \\ [\underline{\text{I}}]v_1(v_2 v_3) \end{array}\right\}\right) \\ — & — & — & — \\ — & — & — & — \\ — & — & — & — \end{array}$$

$\underline{\text{ENDB}}$**

Where i_1—integer number identifying each electrical branch,

i_2—integer number for the initial node of branch,

i_3—integer number for the final node of branch,

v_1—decimal number for nominal component value,

v_2—decimal number for worst case minimum component value,

v_3—decimal number for worst case maximum component value,

[B]—Circuit branch label,

[N]—Circuit nodes label,

[R]—Resistor Impedance label (data in ohms),

[G]—Resistor Admittance label (data in mhos),

[E]—Voltage source label in series with Resistor (data in volts),

[*I*]—Current source label in parallel with Resistor (data in amps).

Example:

INPUT

```
B 1  N 0 2  R 2000.      E   20.
B 2  N 0 1  R 6000.      E   20.
B 3  N 0 1  R 1000.
B 4  N 1 3  R  350.      E  - 0.5
B 5  N 3 0  R  500.
B 6  N 2 3  R  11.1E3
```

ENDB

It is noted that resistor component data may be specified as impedance or admittance while independent voltage or current sources are respectively placed in series or parallel with the resistor.

The Transconductance command is used to specify current gains or dependent current sources such as those occurring in transistor models. It utilizes a tabular format to describe problems containing more than one transconductance.

General form (Tabular):

TRANSCONDUCTANCE**

[T] i_1	[B] i_2 i_3	{[BETA] v_1 ($v_2 v_3$) / [GM] v_1 ($v_2 v_3$)}
—	—	—
—	—	—
—	—	—

ENDT**

Where i_1—integer number identifying each transconductance,
i_2—integer number of the "from" electrical branch,
i_3—integer number of the "to" electrical branch,
v_1—decimal number of nominal gain value,
v_2—decimal number of worst case minimum gain value,
v_3—decimal number of worst case maximum gain value,
[*T*]—Transconductance label,
[*B*]—Current gain direction ("from" and "to" branches) label,
[BETA]—Current gain magnitude label (dimensionless),
[*GM*]—Current gain BETA divided by the Impedance of "from" branch (data in mhos).

Example:

TRANSCONDUCTANCE

```
T 1  B 4 6  BETA 50.
```

ENDT

It is noted that the transconductance values may be given as BETA or GM, and any number of transconductances may be specified between the TRANSCONDUCTANCE command and the label ENDT.

The Execution Print command signals the end of a problem specification and specifies the performance of calculations by the ECAPNC subsystem as prescribed in the Solution and Analysis commands. After the appropriate analysis has been performed, the subsystem is ready to accept further commands.

General form:

EXECUTE (AND) (PRINT)

[NV] i_1 ([CA] i_1 [CV] i_1 [BA] i_1 [BV] i_1 [BP] i_1 [SE] i_1 [MI] i_1)**

Where i_1—integer number equal to 1 (one), **
[NV]—Print Node Voltages label,
[CA]—Print Element Currents label,
[CV]—Print Element Voltages label,
[BA]—Print Branch Currents label,
[BV]—Print Branch Voltages label,
[BP]—Print Branch Power Losses label,
[SE]—Print Sensitivities label,
[MI]—Print Miscellaneous label (nodal and conductance matrices used in calculations).

Example:

EXECUTE AND PRINT

```
NV 1  CA 1  CV 1  BA 1  BV 1  BP 1
```

If an output label is not followed by the integer one (1), the corresponding output is not printed.

After the circuit has been analyzed for the steady-state solution, the previously specified circuit data can be changed utilizing two Data Modification commands. The first command changes circuit component values while the second changes transconductance values. Both commands prepare the circuit specification for reanalysis toward an optimum solution.

The Branch Modification command is used to change impedance, admittance, voltage source, and current source data originally specified in the INPUT command. The branch data is modified in one of two ways:

1. Discrete changes in nominal and worst case data, or

2. Incremental changes of each component value in turn.

The command also utilizes a tabular format in which the word MODIFY starts the tabulation and the label ENDMB signals the end of data.

General form (Tabular):

MODIFY

i_1 [B] i_2 { [R] v_1 ($v_2 v_3$) | [R] v_4 $v_5 v_6$ INCREMENT** | [G] v_1 ($v_2 v_3$) | [G] v_4 $v_5 v_6$ INCREMENT** | [E] v_1 ($v_2 v_3$) | [E] v_4 $v_5 v_6$ INCREMENT** | [I] v_1 ($v_2 v_3$) | [I] v_4 $v_5 v_6$ INCREMENT** }

— — — —
— — — —
— — — —

ENDMB**

Where i_1—integer number sequentially identifying each subcommand in the MODIFY command,**
i_2—integer number of electrical branch to be modified,
v_1—decimal number of changed nominal component value,
v_2—decimal number of changed worst case component minimum value,
v_3—decimal number of changed worst case component maximum value,
v_4—decimal number of starting component value for the increment modification,
v_5—decimal number of increment steps in modification range,
v_6—decimal number of final component value for the increment modification,
[B]—Circuit Branch label,
[R]—Resistor Impedance label (data in ohms),
[G]—Resistor Admittance label (data in mhos),
[E]—Voltage source label (data in volts),
[I]—Current source label (data in amps).

Example:

MODIFY

1 B 1 R 1000. 10. 3000. INCR

ENDMB

The example shows that the Resistor in Branch one (1) is to be modified in ten increments from a value of 1000 ohms to a value of 3000 ohms, and is the only branch to be modified.

The Transconductance Modification command is very similar in construction to the Branch Modification command. In this case, the tabulation starts with the word TMODIFY and ends with the label ENDMT. This command also has a tabular format which can specify either discrete or incremental changes in current gains.

General form (Tabular):

TMODIFY**

i_1 [T] i_2 { [BETA] v_1 ($v_2 v_3$) | [BETA] v_4 $v_5 v_6$ INCREMENT** | [GM] v_1 ($v_2 v_3$) | [GM] v_4 $v_5 v_6$ INCREMENT** }

— — — —
— — — —
— — — —

ENDMT**

Where i_1—integer number sequentially identifying each subcommand in the TMODIFY command, **
i_2—integer number of transconductance to be modified,
v_1—decimal number of changed nominal gain value,
v_2—decimal number of changed worst case minimum gain value,
v_3—decimal number of changed worst case maximum gain value,
v_4—decimal number of starting gain value for the increment modification,
v_5—decimal number of increment steps in modification range,
v_6—decimal number of final gain value for the increment modification,
[T]—Transconductance label,
[BETA]—Current gain magnitude label (dimensionless),

[GM]—Current gain BETA divided by the impedance of "from" branch (data in mhos).

Example:

```
TMODIFY
        1  T 1  BETA 60.
ENDMT
```

The example shows that the current gain is changed from 50 in the original problem to 60 for the next problem solution.

SUBSYSTEM RESPONSE TO POL COMMANDS

To use the ECAPNC subsystem, the engineer writes a Problem Oriented Language program consisting of the predefined ECAPNC subsystem commands. The POL program represents a unique problem which specifies electronic circuit data, analysis, and results. Each POL program is then transcribed on standard 80 column FORTRAN coding sheets for keypunch. The engineer can use all 80 columns of the sheet to program commands, and should be careful to start each command name on a new line. Also, all words and numbers must be separated by spaces within every command.

The subsystem's response to the POL commands is given by the Nominal DC Solution which provides the engineer with a steady-state response for a linear lumped-parameter electronic circuit. The following POL program illustrates a problem solution for the circuit described in Figure 9:

```
cc. 1                                                    cc. 80
  ECAPNC
  DC SOLUTION
  INPUT
  B 1  N 0 2  R  2000.     E   20.
  B 2  N 0 1  R  6000.     E   20.
  B 3  N 0 1  R  1000.
  B 4  N 1 3  R   350.     E  - 0.5
  B 5  N 3 0  R   500.
  B 6  N 2 3  R    11.1E3
  ENDB
  TRANSCONDUCTANCE
  T 1  B 4 6  BETA 50.
  ENDT
  DESCRIPTION TOTALS  BRANCHES 6
                      NODES 3  DCUR 1
  EXECUTE AND PRINT
    NV 1  BV 1  CV 1  CA 1  BP 1  BA 1
```

```
ECAPNC

*****************************************
*                                       *
*          ICES  ECAPNC                 *
*    NEW  CIRCUIT  DESIGN  SUBSYSTEM    *
*                                       *
*                                       *
*                                       *
*            DC  ANALYSIS               *
*                                       *
*                                       *
*        SEPTEMBER,1968   MOD 1         *
*                                       *
*****************************************

DC SOLUTION
INPUT
B 1  N  0  2   R  2000.    E  20.
B 2  N  0  1   R  6000.    E  20.
B 3  N  0  1   R  1000.
B 4  N  1  3   R  350.     E -0.5
B 5  N  3  0   R  500.
B 6  N  2  3   R       11.1E3
ENDB
TRANSCONDUCTANCE
T 1  B  4  6   BETA     50.
ENDT
DESCRIPTION TOTALS       BRANCHES  6  NODES  3  DCUR  1
EXECUTE  AND  PRINT   NV 1  CA 1  CV 1  BA 1  BV 1  BP 1

NODE VOLTAGES

NODES              VOLTAGES
  1-  3      0.27942208D 01     0.11072700D 02     0.22685277D 01

BRANCH VOLTAGES

BRANCHES          VOLTAGES
  1-  4     -0.11072700D 02    -0.27942208D 01    -0.27942208D 01    0.52569314D 00
  5-  6      0.22685277D 01     0.88041720D 01

ELEMENT VOLTAGES

BRANCHES          VOLTAGES
  1-  4      0.89273003D 01     0.17205779D 02    -0.27942208D 01    0.25693135D-01
  5-  6      0.22685277D 01     0.88041720D 01

ELEMENT CURRENTS

BRANCHES          CURRENTS
  1-  4      0.44636488D-02     0.28676258D-02    -0.27942206D-02    0.73408957D-04
  5-  6      0.45370551D-02     0.44636162D-02

ELEMENT POWER LOSSES

BRANCHES          POWER LOSSES
  1-  4      0.39848333D-01     0.49339806D-01     0.78076695D-02    0.18861063D-05
  5-  6      0.10292435D-01     0.39298445D-01

BRANCH CURRENTS

BRANCHES          CURRENTS
  1-  4      0.44636488D-02     0.28676298D-02    -0.27942206D-02    0.73408957D-04
  5-  6      0.45370551D-02     0.44636162D-02
FINISH
```

Figure 10—Nominal DC solution

This program is now ready for processing by the ICES procedure called "Subsystem Execution." The computer results for the Nominal DC Solution (Figure 10) are tabulated according to the circuit node voltages, branch voltages, element voltages, element currents,

element power losses, and branch currents. It is important to note that the POL program is actually a dialogue of the engineer's commands and the computer system's solution results. Within the program, the engineer can specify:

1. Any number of problem solutions on one computer run as long as each solution is initiated by a DC or MODIFY command and terminated by an EXECUTE command.
2. Any arbitrary order of the analysis and circuit data commands (3 through 6) when included between the DC and EXECUTE commands.
3. Any arbitrary order of the tabulated data within the INPUT, TRANSCONDUCTANCE, MODIFY commands as long as the data is appropriately labeled.

APPENDIX C

MODIFICATION OF ECAP

The study of ECAP as the first ELECSYS subsystem began with the gathering of source information on the external and internal program description. The capabilities of ECAP were analyzed for conversion feasibility as determined by ICES limitations and requirements. The following results became the boundary conditions for the ICETRAN conversion of ECAP:

1. The ECAP program has satisfied the ICES requirement specifying that engineering computer programs must be written in FORTRAN IV, E Level Subset;
2. The size of ECAP and time allotted for the conversion process restricted the programming effort to the DC Analysis portion of ECAP; and
3. The modifications will primarily consist of adding ICETRAN subprograms for the transfer of engineering data to the appropriate FORTRAN language variables utilized in ECAP.

The study also indicated that the design of the ICETRAN language subprogram is subject to the concurrent designs of two other interactive data translation programs, the Command Definition Language and the Problem Oriented Language. In essence, then, the design considerations in all three language programming areas must reflect a mutual satisfaction of constraints before the subsystem can be declared operational.

ECAP DC ANALYSIS

The ICETRAN conversion process started with the acquisition of a disposable magnetic tape reel containing the latest version of ECAP. Using an IBM 360/20 computer, the contents of the tape were displayed in the form of a program listing and punched-card program deck. The program listing yielded information on the internal program logic and structure, while the card deck served as the physical means of implementing ICETRAN language modifications.

The DC Analysis card deck contained approximately 1200 FORTRAN IV source cards which were divided into the following twelve subprograms (subroutines).

1. Subroutine ECA19—Passes control of program from the ECAP Language Processor (replaced by ICES) to Subroutine ECA20.
2. Subroutine ECA20—Controls processing of DC Analysis, which includes DC nominal solution, Sensitivity, Worst Case, Standard Deviation, and component parameter modification solution.
3. Subroutine ECA22—Calculates DC nominal solution.
4. Subroutine ECA23—Calculates DC nominal solution.
5. Subroutine ECA24—Calculates DC Nodal Impedance Matrix.
6. Subroutine ECA26—Calculates DC Nodal Conductance Matrix.
7. Subroutine ECA25—Prints calculations of DC nominal solution.
8. Subroutine ECA27—Controls Worst Case Analysis of DC nominal solution calculations.
9. Subroutine ECA28—Calculates Sensitivities, and Standard Deviation; and prints Sensitivities.
10. Subroutine ECA29—Prints Standard Deviation.
11. Subroutine ECA30—Modifies component parameter values.
12. Subroutine ECA31—Prints type of modified component, and modified value during a particular interaction cycle.

It is noted that Subroutine ECA19 is an interface subprogram between the DC Analysis subprograms and a set of subprograms known as the ECAP Language

Processor. This processor program performs the same function as the ICES data translation programs (ICETRAN subprograms, Command Definition Language, and Problem Oriented Language), and will be partially replaced by a newly created set of ICETRAN subprograms.

COMPATIBILITY OF ICETRAN AND FORTRAN

The conversion process was divided into two jobs:

1. To insure compatibility between ICETRAN statement requirements and existing ECAP FORTRAN IV statements, and
2. To create the necessary ICETRAN subprograms for the ICES data translation function.

Both tasks require a thorough knowledge of FORTRAN, and ICETRAN before any program modifications can be undertaken.

The first job consisted of a complete inspection of all FORTRAN language statements in the DC Analysis program. This resulted in the modification to ICETRAN of two types of FORTRAN statements: The Subroutine Statement, and Literal Constant Statement.

The Subroutine Statement appears as the first statement in any subroutine program. It has the following general ICETRAN form:

SUBROUTINE name $(a_1, a_2, \ldots, a_n)$

Where name is the subprogram name consisting of one to six letters or digits. The first must be a letter, but cannot start with the letters QQ.

$a_1, a_2, \ldots, a_n$ are the subroutine arguments (variable names) of data values used in the subprogram for calculations.

This statement is the same as its FORTRAN equivalent, but, in ECAP, subroutines with arguments were written without a space between the subprogram name and subroutine arguments; i.e., SUBROUTINE ECA20(ZPRL). This is critical in ICETRAN, especially in the Load Module Generation task in Appendix A of this paper. It was found that five subroutines (ECA20, ECA22, ECA24, ECA26, and ECA28) required the appropriate spacing modification to conform with the ICETRAN statement format.

The Literal Constant Statement is unique to ICETRAN and facilitates the programming of constants containing alphameric information. It has the following general ICETRAN form:

$$v='e'$$

Where v is a double precision scalar or variable name less than six letters.

e is a set of eight or less alphameric characters set off by apostrophes.

In ordinary FORTRAN, literal constants are defined as decimal equivalents which are translated into a hexidecimal representation of the literal word. For example, in Subroutine ECA28, the integer variable NJ is used to output the letter R signifying a particular circuit component, the Resistor. The following tabulation compares FORTRAN and ICETRAN statement:

FORTRAN $NJ=-0650100672$

ICETRAN $NJ='R'$

After ICETRAN Precompile (System Task 1)

$NJ=QQHCNV(-0650100672, +1077952576)$

Before ICETRAN, the appropriate decimal equivalent number for $'R'$ was determined by a special computer program. Now, this program is built into ICES as a load module program, *QQHCNV*, and eliminates a previous FORTRAN programming limitation.

ICETRAN CONVERSION OF ECAP

The second conversion job had to be performed in order to satisfy the following ECAP and ICES design limitations:

1. All circuit data in ECAP is stored in subscripted variable arrays.
2. The ICES Command Definition Language (CDL) rules state that data cannot be directly translated into subscripted variable arrays.

In order to circumvent these design limitations with a minimum of ECAP programming changes, a solution was devised that would utilize a set of ICETRAN subprograms in conjunction with the Command Definition Language to transfer circuit data from temporary variables into the required subscripted variables. In particular, the CDL could translate each type of circuit data into a standard, predefined set of temporary scalar variables. The CDL can then designate an ICETRAN subprogram to perform the final translation of data into a unique subscripted variable within an array. It is important to note that each subprogram is individually executed as many times as necessary to complete an array. The first step in this proposed method is to tabulate all ECAP data arrays and establish a set of temporary scalar variables for the subprogram data translation. The following tabula-

tion describes the ECAP data arrays, the corresponding temporary data variables, and overall data function:

ECAP Variable	Temporary	Function
I	NUMB	Circuit Branch Number
Y (I)	YT	Impedance (R), Admittance (G)
YMIN (I)	YMINT	Minimum R, G
YMAX (I)	YMAXT	Maximum R, G
E (I)	ET	Voltage Source (E)
EMIN (I)	ET	Minimum E
EMAX (I)	EMAXT	Maximum E
AMP (I)	AMPT	Current Source (I)
AMPMIN (I)	AMPMIT	Minimum I
AMPMAX (I)	AMPMAT	Maximum I
NINIT (I)	NINITT	Initial Node of Circuit Element
NFIN (I)	NFINT	Final Node of Circuit Element
I	NUMT	Transconductance Number
YTERM (I)	YTERMT	Transconductance (BETA, or GM)
YTERML (I)	YTERLT	Minimum BETA, or GM
YTERMH (I)	YTERHT	Maximum BETA, or GM
ICOLT (I)	ICOLTT	Transconductance "FROM" Branch
IROWT (I)	IROWTT	Transconductance "TO" Branch

To modify data within these previous arrays, the following variables are utilized:

VFIRST (I)	VFIRTT	Nominal or First Iteration value
VSECND (I)	VSECNT	Minimum or Number of Increments
VLAST (I)	VLASTT	Maximum of Last Iteration value
MOPARM (I)	MOPART	Type of Component (R, G, E, I, or GM)
MOSTEP (I)	MOSTET	Increment Designation
MOBRN (I)	NUMT	Branch or Transconductance No.

To obtain the engineering results of the DC Analysis, the following output variables are utilized:

NPRINT (1)	NPR1	Prints Node Voltages
NPRINT (2)	NPR2	Prints Element Currents
NPRINT (3)	NPR3	Prints Element Voltages
NPRINT (4)	NPR4	Prints Branch Currents
NPRINT (5)	NPR5	Prints Branch Voltages
NPRINT (6)	NPR6	Prints Branch Power Losses
NPRINT (7)	NPR7	Prints Sensitivity Analysis
NRPINT (10)	NPR10	Prints Nodal Impedance and Nodal Admittance Matrices.

In summary, the ECAP DC Analysis program was converted to ICETRAN in two steps. The first step required that a compatibility study be performed to check the language interface between ECAP FORTRAN and ICES FORTRAN (ICETRAN). It resulted in the modification of several ECAP FORTRAN statements which would have created ICES system errors. The second step was required because of a design limitation concerning the treatment of subscripted variables in the ICES Command Definition Language. A set of ICETRAN subprograms was developed to accomplish the final translation of circuit data into ECAP subscripted variables. The completion of both steps produced a permanently stored set of DC Circuit Analysis subprograms.

Interactive graphic consoles—Environment and software

by ROBERT L. BECKERMEYER

Research Laboratories, General Motors Corporation
Warren, Michigan

INTRODUCTION

The usual software support for graphic consoles does not provide system services designed for the console user who is a production-oriented application expert. This paper describes a console environment and high-level language with a supporting operating system designed for the application expert.

The aim has been to improve four troublesome areas: programming a new application, defining input data for functional routines, screen management and dynamic error recovery.

The importance and acute need for improved software of this type is substantiated by the large number of such systems being designed and languages proposed. RAND Corporation is developing a programmer-oriented graphics operation.[1] At the University of Utah, W. M. Newman is working on a high-level programming system for remote graphic terminals,[2] and A. van Dam at Brown University is working on a hypertext editing system.[3] The list is very long.

At the G.M. Research Laboratories, experience with the DAC system[4] has enabled us to evolve a console environment incorporating those human factors in man-machine communication which we have found to be best. This paper describes the principal parts of that environment with emphasis on external appearance.

The material is divided into four main sections: graphic console environment, programming language, procedure execution, and operating system. The principal features of the DAC programming services and console man-machine communication techniques will be outlined for background information. The system facility and console appearance for selection of values for input data will be described and illustrated. Control of displays and interaction with procedure* execution will also be discussed. Procedures use console displays for a two-way communication instrument between user and machine.

This environment is based on the belief that people solving a problem using a graphic console should determine as much of the display and execution logic as possible. By making the programming language for display generation and procedure logic use exactly the communication techniques used during problem-solving sessions, the application expert feels more at ease in a programming situation. An additional strong aid to the programming task is the automatic retrospective programming provided by the operating system. This allows a console user to "capture" the solution procedure as it evolves during his problem-solving session.

Program execution is done interpretively on an internal structure that is, by design, amenable to change. Most graphic console applications are "cut and try" and highly dependent on the console user's background and experience. The approach and problem solution has to be individualistic. With a flexible internal structure the solutions can be altered many times so the final procedure is tailored by the console user.

An operating system to manage screen displays, accept all user interrupts and supervise execution will be described in terms of the fundamental modules within it. This system allows data to be defined by a new technique called "association." Association enforces a uniformity in communication that helps to insure usability of new programs without much experience by the user. And yet the system is in no way dependent on the actual appearance of the displays or assignments of function buttons.* The operating system** is table driven, allowing it to service different applications and many consoles simultaneously without restricting the

* The term procedure in this paper is a collection of computer programs or subroutines linked with a logic structure designed for several variations of a problem solution.

* A function button is a key or button that may be programmed to link to a particular computer code when pushed by the console user. These buttons may be special keys on the alphanumeric keyboard or collected together in a separate box on the console.

** Operating system in this paper refers not to the basic system support like OS/360, but to a collection of routines designed to interpret procedure internal structure, handle all data associated with this structure and interpret all the console user interactions with functional parts of the console (keyboard, function button or light pen).

flexibility required by the user and at the same time keeping the operations in the different consoles separated.

Included is a complete example illustrating the principal parts of this environment. Although only a small portion of the language symbols are used in the example, those used are among the most important.

GRAPHIC CONSOLE ENVIRONMENT

Long-range objective

A seven-year study[4,5] on computer graphics with emphasis on programming efficiency has led to the long-range objective of a more direct path for console users to design solutions for their problems. Many applications have been considered including body design, statistical analysis, information retrieval and project planning and control. A characteristic of these applications is that the procedures continuously change to incorporate improvements and new approaches. For programming efficiency, a high-level language allowing the application expert to program in an environment very close to that in which they solve application problems is needed.

DAC background

For DAC[4] system programmers designed what is called the Descriptive Geometry Language (DGL). The DGL statement

```
LN7, PT8, PT9, UF2 = INLIN(SU3, Y = 20.), RG2
```

makes symbolic references to geometric data. It also uses a package of functional evaluations and subroutines to do transformations, generate new data and other geometric operations on the data. In DAC these subroutines are called operators. In the DGL statement above, the INLIN operator is used to find intersections of surfaces and lines. We take the surface SU3 and the plane Y = 20 and form an intersection over the range RG2. The intersection line is LN7 with end points PT8 and PT9 and the surface normal unit vector function UF2. A large number of such operators are available to designers. All were written by G.M. Research mathematicians.

Designers and engineers use these operators in statements to write a computer program or procedure. The system then interprets these statements one by one in an interpretive mode. The designer or engineer, when at the console, can start and stop at any statement in his procedure. If need be he can back up and reexecute any number of the statements to correct or modify the design. Graphic Displays on the DAC console screen provide for review and evaluation of the results. We found, however, that application specialists were not amenable to the discipline of a written programming language and a search for a method of defining procedures in a manner more natural to the users continued.

In the next system, groups of operators were made available simultaneously via DAC console function buttons (these are similar to those on the IBM 2250 function keyboard). The groups were called "overlays"[5] because as each group became available, a celluloid sheet with identifying labels for that group was placed over the keyboard. In general, operators were grouped by function; for example, the SURFACE overlay operators dealt with surface definition problems. There were eight overlays in all. The coding for these operators was done by system programmers and required 13 to 30 man-months per overlay. Using a combination of Descriptive Geometry Language procedures and overlays, the DAC system is now being used heavily for production work.

Based on our experience, the next development was to communicate with the user through pictogram[6] representations of the operators. Through considerable experimentation with DAC-experienced users we were led to the amalgam of proven DAC techniques and pictogram communication which constitutes our present console environment.

Improved console environment

The new environment provides the same operators as were available through the function buttons of an overlay in the previous system. The improvement is obtained by pictorial communication between the mathematical function routines stored internally to the computer and the man sitting at the console who wants to use them in a problem solution.

Screen management is carried out by dividing the console display area into three function areas. A typical screen format is shown in Figure 1. The upper area is called the work area. The lower right-hand area is known as the operator area and in some ways corresponds to the celluloid overlay of DAC. The lower left-hand area is known as the control area. The operating system distinguishes between console user actions in these areas and responds accordingly. The display in the work area shows three parts of a design problem: a windshield surface, some lines for the windshield wiper pattern and

the rear view mirror subassembly.* As the design is evolved, the modifications and additions are shown in this area of the screen.

The operator area in Figure 1 contains only text describing intersection operations and thus wastes communication potential of a graphic console. Contrast this with the operator area shown in Figure 2 where a pictorial representation, a pictogram (6, page 384), of the functional capabilities of the operator realizes more of this power.

The pictograms in the operator area show three types of geometric intersections to determining an intersection point. These illustrations, if well done, have a "universality" that will transcend the limitations of the prose equivalent shown in Figure 1. The differences between these types of intersections and the geometric elements involved is readily understood by those trained in elementary geometry regardless of the way in which they express the concept, as a mathematician or designer. Consider the problem of designing a tool used for automobile body panels called a die shoe and having people from different departments working on problems related to making this die shoe. Different jargon may be used by the departments, but all persons will recognize a blueprint or stylized drawing (or pictogram) of this part. A simplification of the drawing of this part with emphasis on the possible design operations can be shown

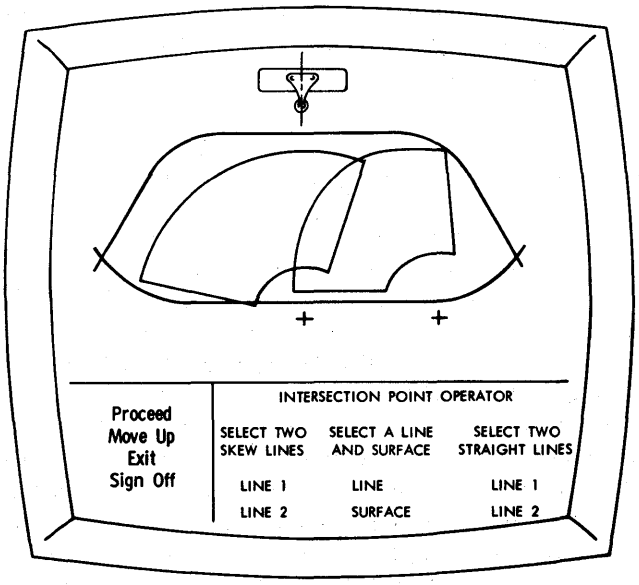


Figure 1—Display area format

* A subassembly is the automobile body designer's term for a detail drawing of a subpart of a major assembly. For example, a door lock subassembly of the door panel.

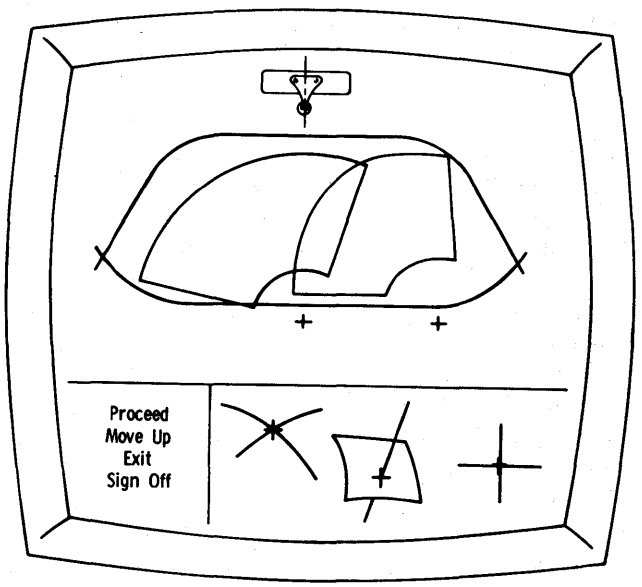


Figure 2—External appearance of a graphic operator

in the operator area and the operator will be a useful tool for people in any of the involved departments.

In the control area there are four possible controls. The console user transmits his wishes for display control and execution interruption to the operating system through this area. Use of the three functional areas is described in more detail below.

Suppose the user wished to find the intersection of a surface and a line. These two parameters are indicated in the center pictogram which represents this operation. The user must supply the values for the parameters from the work area display, however. The order in which the values are defined is arbitrary. The user starts by selecting with the light pen either the surface or the line in the middle pictogram. This first selection tells the system two things, which type of intersection is to be calculated and which parameter is to be supplied first.

The choice of parameters is done by associating each stylized parameter in the pictogram with an actual counterpart in the work area, using the light pen to indicate the two members of the correspondence. The first part of the association is made in the pictogram allowing the system to limit the user's choice in the work area to elements of compatible type. This prevents such erroneous associations as a line parameter with a surface element. This error prevention is done using the technique of selective enabling described below. As soon as all the parameters for the computation have been defined, the system calls the appropriate subroutine passing the work area line and surface chosen by the

user. If mathematically possible, the subroutine computes the intersection and passes the intersection point back to the operator for display in the work area of the screen.

At any time prior to the completion of parameter definition and the subroutine call, the user may redefine any of the data values by a simple reassociation. He first selects the parameter in the pictogram that is to be redefined and the system prepares to override the work area data choice made earlier. The user's second selection of a work area item completes the parameter redefinition.

In the windshield example, the user might associate the line parameter in the pictogram with the center line of the rear view mirror and the surface parameter with the windshield surface. The windshield surface is selected by placing the light pen over any part of the boundary lines of the windshield.

The light pen selections make use of both selective enabling and reactive displays.[7] On display in the work area of the example are lines, surfaces and points. The operator can cause the system to enable for selection lines only, points only, surfaces only or any combination. This selective enabling is controlled by information stored with each parameter in the operator and passed to the system when the user selects a parameter in a pictogram. All attempts to select items which are not enabled are ignored. Selective enabling is not used in the operator area display; all parameter representations in this area are always enabled, allowing the user to respecify parameters should he make an error.

The reactive display technique allows the user to ensure that the intended element is being selected. This is particularly important in crowded parts of the work area where elements are close together. When a light pen is over an enabled item, it is intensified and will remain so until it is selected by removal of the pen from the screen or until the pen is moved away from the item. The user may thus adjust the position of the pen on the screen until the correct element is intensified and then make his selection. As a further aid to the user, only enabled items are intensified.

The words on display in the control area represent functions which can be executed independently of the current operator and of the function being performed by the operator. For instance, if the user selects "MOVE UP," the operator is interrupted and the items on display in the work area are shifted or translated up some predetermined distance.

If an error occurs while using any operator, an appropriate message can be put on display in the operator area and the entire interpretation of the operator will stop until the user acknowledges the error with the light pen. Thus he has an opportunity to take some immediate corrective action and continue. Response to an error might involve a change to a parameter value and reexecution of that part of the operator or the current operator may be pushed down and another operator called up to generate new data. This new data may then be used to replace some parameter value that caused the error condition. The operator may also be programmed to open an auxiliary path allowing the user to supply extra data needed to overcome the error condition. The system can provide some standard error escapes, but the operator can be programmed to have almost an unlimited variety of error recovery paths dynamically selectable by the user.

With the same operating system supporting all applications for screen management and interrupt processing, data selection by association becomes paramount and can be fine-tuned giving the console user good response to his requests. All operators have a similar internal structure: an APL[8] structure generated by APL statements within a system module from the graphic operator source program. This structure is illustrated for the example in Figure 2 later in the paper. With similar internal structures a more uniform approach to the console environment can be assured. Uniformity is important because any departure from a conventional course of action is a distraction for the user. This became evident while observing DAC users working in the overlay environment. Each overlay was considered an experiment in human factors so the standard course of action varied from overlay to overlay and resulted in confusion for the user when switching overlays.

OPERATOR PROGRAMMING LANGUAGE

Operator programming

The operator in the previous example has shown that an operator must be able to present displays in the operator area and associate each parameter of the operator with the data selected by the console user. An operator must also be able to invoke subroutines and other operators, display the output from a subroutine call, and give the console user the ability to select alternative ways to solve his problem. The flow of logic in an operator is such that a subroutine will not be called until all its parameters have been given values. As soon as the last parameter receives its value the subroutine is called. The user has freedom of choice of the order in which he specifies parameters, thus he may define values for the parameters of several subroutines in a mingled sequence. For this reason the order

in which the statements of an operator will be executed is indeterminate at the time of programming. This contrasts with the usual type of program where statements are executed in strict sequence except for branching. The language used to program operators must not only provide for simultaneous flow of data and logic, but must allow an operator to begin execution at almost any statement. Once the initial display for an operator is made in the operator area, the user's selection of the available parameters determines where the operating system starts operator interpretation.

Graphical operator language and representation

The illustration of Figure 3 shows both the graphic operator interface (operator picture) as before and a graphic representation of the operator program, the programmer's view of the operator. This program representation is called a logic diagram since it describes the logic flow of the operator. It is the programmer's source program and is never shown to the console user in an application problem-solving situation; however, since operators are programmed at the screen in the same way in which the application user solves his problem, this diagram forms the work area display for the operator programmer. The programming language for operators is really a set of operators which add symbols to the logic diagram and build the structure to be used by the interpreter during execution. In Figure 3 dashed lines show the connection between symbols in the diagram that represent data parameters and their graphic display representation in the operator area.

This operator consists of three alternative ways of generating an intersection point. Each alternative is shown as a horizontal line of symbols and ends with an X symbol. Each line of symbols is referred to as an alternative path. The upper path, represented in the operator picture by the left illustration, covers the case of intersecting two skew lines, and is made up of two AS symbols, a CL and a DS symbol. The middle path represents an intersection of a line and surface and the lower path the intersection of two straight lines. The three alternative paths are joined by a vertical line to the AL or alternative symbol. With the AL symbol the console user is free to select the intersection construction that fits his needs for the problem at hand. Any one of the parameters may be selected to start the operator and this selection will determine which alternative is to be used.

In the first alternative path, the AS symbol representing a data parameter denotes a data association as described above. Part of the information stored with

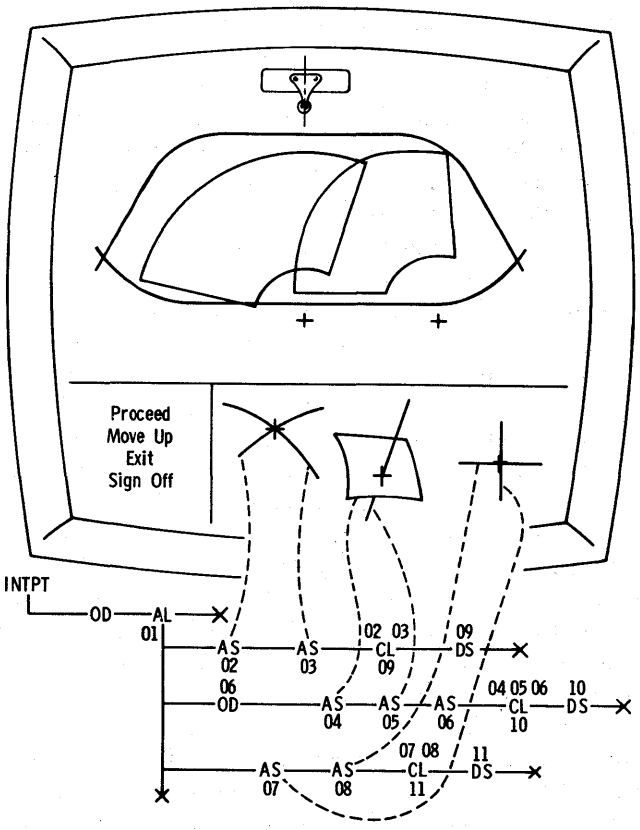


Figure 3—Example operator

this symbol is a data-type used for selective enabling. The number 02 below the AS is called a data code. The data code is no different from a conventional parameter name except it is a more compact way to display the parameter. If the parameter name is desired, it can be displayed along with data declarations and all other information about the symbol. The data code also represents an index into the data table (Figure 5) used at execution time to store the data values assigned to each parameter in the operator. If a complete parameter cross reference is desired that data code may be selected and all references to that parameter will be shown by intensifying all diagram symbols using that parameter. The data code number is a sequence of numbers generated by the system and assigned to symbols requiring a data code as they are added to the diagram by the programmer.

The third symbol, a CL symbol, represents a subroutine call. The subroutine name is stored internally with the symbol and may be displayed if requested by the user. The data code 02 from the first symbol and data code 03 from the second symbol are shown above the CL. This shows 02 and 03 to be input items for the

CL. Below the CL in the first path is data code 09. This is the output data, the intersection point, from the subroutine.

The DS symbol, next on the first alternative path, means that a display of a computed data item is to be made in the work area. Data code 09 above the DS symbol indicates that this parameter is to be displayed.

All symbols in the logic diagram are interconnected by a solid line. To follow this line from symbol to symbol will show a logic flow within the operator. A sequence of interconnected symbols will end with an X symbol representing a path end. The two AS symbols, the CL and DS symbol of the first path are interconnected and terminated with the X signifying a complete path for one choice of the alternative. When the X is reached the logic flow returns to the next higher level and continues. An X symbol on the highest level path terminates the operator. The logic diagram differs from the standard flow chart in that a path of symbols has level significance and the logic flow automatically moves to the next higher level when the path end is reached. An operator logic diagram also has a larger set of symbols than the average flow chart has kinds of polygons.

Further explanation of the alternative (AL) symbol can now be given. It is possible to make displayed items a part of more than one illustration, but regrettably, this is not demonstrated in the example. There must be at least one unique item to each AL path to allow the system to determine which alternative the user intends to use from the selection of a unique combination of items. The unwanted alternatives are removed from the operator picture area, further aiding the user by deleting possible erroneous choices. The remaining pictogram can then be enriched by additional required parameters of lesser importance. This is shown in the example by the OD, operator display, in the middle AL path.

The OD symbol allows the programmer to determine the composition of the operator area. Operators may have one or more operator pictures and additions may be made to the current picture. The OD symbol serves both of these functions. Any OD on the principal or highest level path means that the current picture is to be wiped out and a new composition begins. An OD serving this function is shown in the example immediately after the name INTPT. The second OD in the example on the middle alternative path shows the second function of an OD. This OD is a tolerance value, unimportant to the user of the operator until that path is chosen so the display for that parameter is held back until needed. The name symbol, INTPT in the example, is the initializer for the operator containing required limits and constants.

The Operator Programming Language Compiler allows the programmer to start building up the logic diagram in the work area symbol by symbol. At any desired time, the logic diagram can be shifted to the operator area and an operator picture built up in the work area. Light pen associations between diagram symbols and items on display in the work area build up the operator graphic interface. There are many sources for displays that may become parts of operator pictures including a picture library, digitizing physical models, parts of work area displays resulting from a design session and a sketchpad facility.

The operator programmer can make rapid and frequent changes to his operators. The display for the operator picture can be altered in any way and any change desired can be made to the logic diagrams. The language compiler will automatically update the internal structure to reflect the changes.

The operators can be executed with a minimum of the logic diagram programmed so it can be built "cut and try." Our DAC experience shows that this is much faster for this type of work than a conventional procedural language that requires careful preplanning of all logic and all too often complete rewriting to incorporate modifications.

PROCEDURE EXECUTION

Interpretive execution

A module of the operating system called the Executor proceeds from symbol to symbol performing the execution as in conventional interpreters. Major symbols like NAME, OD, AL, AS and CL each have subroutines to perform all the functions required. In this example, the NAME processor sets up tables that drive the operating system. These are illustrated in Figure 5 and discussed briefly in the operating system section.

After initialization the OD module is called. This clears the operator area display and searches the operator structure for symbols with displays, e.g., an AS symbol. The display information is built up until another OD symbol is found or the lowest path end comes up. The picture is then painted on the screen. In the example the operator picture would be painted as shown in Figure 2.

Suggested input sequence

In our example the next symbol is the AL and the suggested input sequence scheme comes into play. This is a prompting mechanism to aid the user with an unfamiliar operator and also to reduce the number of

light pen selections that must be made. Its real function is to make the first selection of each association for the user. The first path of the AL and the first AS (with data code 02 below) is suggested. Externally, the line in the pictogram corresponding to that AS is intensified, selective enabling is set up based on the data type stored with the AS symbol, and the system waits for the user's work area light pen selection to complete the association. However, all other parameters in the work area remain enabled and the user may override the suggestion. The suggested input sequence is then interrupted by selecting the parameter he requires. If this changes AL paths the selected path is used for suggestion, otherwise the system reverts to where the sequence was interrupted. In the windshield design example above, where we suggested the operator was used to intersect the rear view mirror center line with the windshield surface, the first selection of the line in the middle pictogram of the operator picture would be an interruption to the suggested input sequence. The sequence would be shifted from the upper path of the AL to the middle path. The suggested input sequence is determined entirely by programming, according to the order of symbols in the logic structure. The CL, call, symbol causes the Executor to check whether all parameters required for the call have been given values so that the subroutine may be called.

The CL symbol has a feature not available for standard subroutine calls. It may be designated as "once only" or "continuous" execution. The "once only" form is the standard subroutine call but "continuous" means that as long as any one input parameter is continually being given a new data value the subroutine will continue to be reexecuted allowing a dynamic change to an affected part of the design in the work picture. This allows the user to make many small adjustments to his work picture and monitor the result.

Operator internal structure

A portion of the internal APL structure and corresponding logic diagram is shown in Figure 4. The language compiler, as directed by the logic diagram built by the programmer, generates the internal structure by executing APL[9] statements.

The structure is composed of blocks of information or entities and sets of entities with at least one kind of entity for each symbol in the language. The entities can be connected together into a set by internal pointers to the next entity and to the previous entity in the set. Entities may be members of one or more sets. There is an entity type corresponding to each symbol type in the operator logic diagrams.

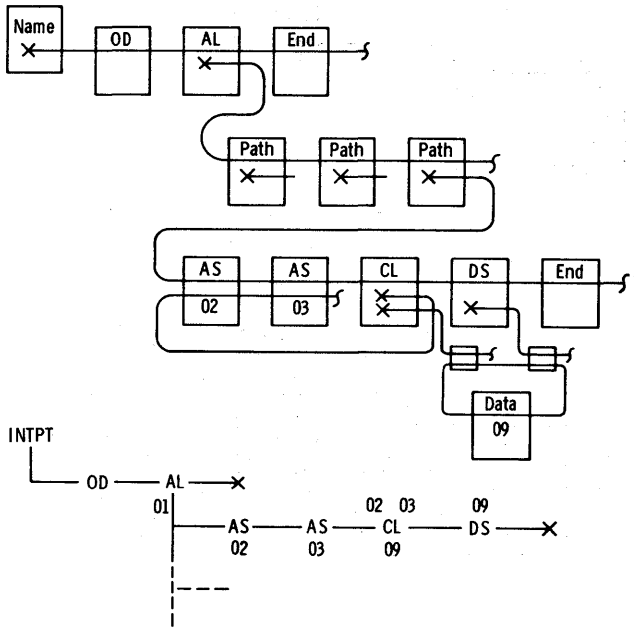


Figure 4—Internal APL structure

There are many ways an operator structure may be modified. The entities can be changed, deleted or hooked up in sets in various ways. These are local structure changes with the advantage that they do not require regeneration of the rest of the structure.

In the example of Figure 4, the logic diagram symbols are each translated to an entity type in the internal structure while logic flow and data flow determine the set connections. Connected to the AL entity are its three alternative paths and most of the entities for the intersection of skew lines path are shown. Also shown in Figure 4 are two node blocks which serve to connect the DATA entity (data code 09) to a set in the CL entity and another set, but with the same set name, in the DS entity.[9]

Hierarchical program structures

A very important symbol not shown in the example allows an existing operator to become a substructure to a higher level new structure being programmed. It is the primitive operator or PO symbol. An existing structure, when included in another structure, is a lower hierarchical structure and therefore primitive relative to the higher level structure. The PO is like an external subroutine with procedural language programs. This means any operator structure can become a substructure to another operator's structure. Large complicated

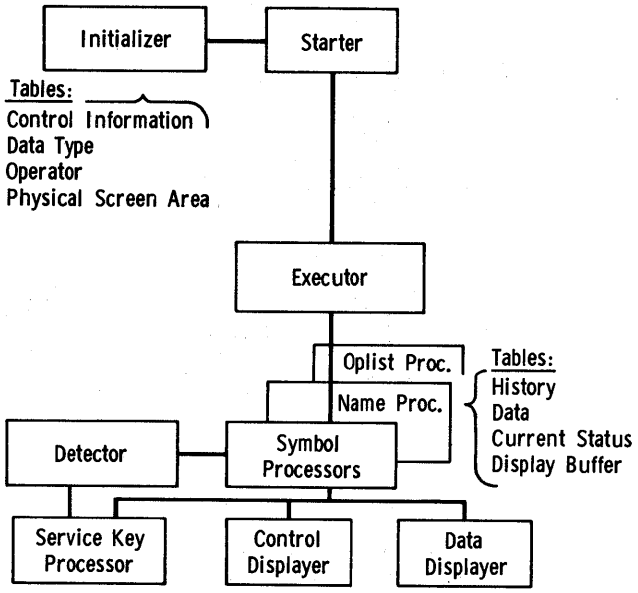


Figure 5—Console operating system modules

structures can be built up in modular fashion from small simple structures.

Automatic retrospective programming

As a problem is being solved using operator after operator, the operating system can be told to keep a history trace, i.e., to "capture" the procedure. The system builds a simple structure and generates a new PO symbol in that structure for each new operator as it is called. The system tables generated for execution[10] of that operator are saved with the PO. When this simple structure is complete, just a string of PO symbols, it is modifiable in all the standard ways for operator logic diagram generation and modification. Thus a new operator is generated without programming a logic structure, symbol by symbol, in the standard way. The new operator as a simple linking of existing structures was automatically generated by the system from the history trace that was kept. It is retrospective in showing all steps to the new procedure including all data used as input and even the false starts that were abandoned.

CONSOLE OPERATING SYSTEM

System modules and driver tables

The interpretive processing of the operator internal structures is done by system modules driven by two groups of tables as shown in Figure 5. One group of tables contains information concerning a particular application, screen formatting, data types, graphic orders for control area display, function button assignments, etc. This information remains constant during execution and thus these tables augmenting all the individual operator structures may be shared between a number of consoles working on the same application. The second group contains information about the individual console user and the current operator being executed. The use of a standard set of system modules which are controlled by tables allows an application to be tailored to its particular needs. At the same time a uniform approach to the environment is enforced. People working in a particular application are never "surprised" by the kinds of reactions that are expected from them in using their operators just because some operator programmer wished for notoriety by giving his operators a unique set of conventions.

The initializer allocates and initializes the tables universal to an application. This initialization information is supplied by a programmer using one of the programming operators. Different initializing information is called in if a switch is made from one application to another. The control information table determines the control area display. The data-type table indicates types of data to be permitted when setting up for selective enabling for an association. The physical screen area table formats the screen into work area, operator area and control area. The operator table builds up a list of operators used when the user requests automatic retrospective programming.

The name processor starts interpretation of a particular operator. The tables it keeps are for the current operator only. They are saved if a history is being kept, but otherwise freed when the operator is completed.

The executor module carries the structure interpreting along from symbol to symbol. It determines from the entity encountered what processor to call. If a user steps out of the suggested input sequence, the executor must make the change in logic flow and then, when the association is completed, revert to the suggested input sequence.

PROJECT DEVELOPMENT STATUS AND PLANS

An experimental implementation of the key system modules and major symbol processors has been made for the IBM 360/67, a paged time-shared machine, with several 2250 III graphic consoles attached. Positive results were shown for the proposed internal structure

for operators and system management of the three functional areas of the screen.

Further development for the operator programming language is in progress. The power of the language needs to be increased to include dynamic expressions for the console user while in a problem-solving mode. These are basic arithmetic expressions on numerical data. A higher level expression permitting application defined expressions on the application dependent variables is also under consideration. An example of the higher level expression is the automatic calling up by the system of the two skew line intersection point operator when a point item is needed as data and the console user has just selected two skew lines. More development for the automatic retrospective programming facility is also planned.

SUMMARY

The treatment of design problems at a graphic console demands communication between man and machine in a manner which is natural to the user so that he may concentrate on finding application problem solutions. This type of interaction can be achieved by presenting the user with pictograms symbolizing concepts and allowing him to select data by making an association between elements of the pictogram and items of the work picture.

Through selective enabling and reactive displays the number of erroneous selections can be greatly reduced. Since data selections may take place in any order, the operators which manipulate the data differ from normal programs in that the statements are not executed in a predetermined order.

The programming of these operators is done in the same graphic environment as application problem solving using the same operator techniques. Through a recording technique the sequence of operations at a console may be captured and used as new operators.

The operating system will manage three functional areas of the screen and permit extensive user interaction with the system to control problem execution and the screen displays. A uniformity of environment can be achieved throughout all the programs of an application that will make the user more at ease and minimize learning to use new programs.

ACKNOWLEDGMENTS

The techniques and ideas presented in this paper have been developed over a period of time at General Motors Research Laboratories as a cooperative effort of many people working on DAC I and successor projects. I thank Michael Marcotty for his assistance in the preparation of the material for this paper.

REFERENCES

1 B W BOEM et al
POGO: Programmer-Oriented Graphics Operation
Proc of the Fall Joint Computer Conference Vol 34 pp 321-330 AFIPS Press Montvale New Jersey 1969

2 W M NEWMAN
A high-level programming system for a remote time-shared graphics terminal
Conference on Computer Graphics University of Illinois Urbana Illinois April 1969

3 A VAN DAM
Hypertex editing system for the /360
Conference on Computer Graphics University of Illinois Urbana Illinois April 1969

4 E L JACKS
A laboratory for the study of graphical man-machine communication
Proc of the Fall Joint Computer Conference Vol 26 pp 343-350 Spartan Books Baltimore Maryland 1964

5 B HARGREAVES et al
Image processing hardware for a man-machine graphical communication system
Proc of the Fall Joint Computer Conference Vol 26 pp 363-386 Spartan Books Baltimore Maryland 1964

6 P A KOLERS
Some formal characteristics of pictograms
American Scientist Vol 57 No 3 pp 348-363 1969

7 J D JOYCE M J CIANCIOLO
Reactive displays: improving man-machine communication
Proc of the Fall Joint Computer Conference Vol 31 pp 713-721 Thompson Book Company Washington D C 1967

8 G G DODD
APL—A language for associative data handling in PL/I
Proc of the Fall Joint Computer Conference Vol 29 pp 677-684 Spartan Books Washington D C 1966

9 G G DODD
APL—Associative programming language user's manual
Research Publication GMR 622 General Motors Corporation Warren Michigan

10 G G DODD
Associative information techniques
Associative Aspects of Problem Solving p 51
American Elsevier Publishing Company Inc New York New York 1970

MDS—A unique project in computer-assisted mathematics

by ROLFE H. NEWTON and PAUL W. VONHOF

Rochester Institute of Technology
Rochester, New York

INTRODUCTION—A MATHEMATICS DIAGNOSTIC SYSTEM?!!!

In the education of the deaf, are there problems so special that their solution justifies the use of a computer? And if there are such problems, how can the computer best be used to solve them? Early in the history of the National Technical Institute for the Deaf, its administrators voted "Yes" on the first question and "Let's find out" on the second. As a result of these decisions, an IBM 1500 Instructional System was included in the Institute's original home on the campus of the Rochester Institute of Technology.

The National Technical Institute for the Deaf (NTID), is the only institution devoted exclusively to education of the deaf in technical and scientific studies beyond the secondary level. The NTID is a division of the Rochester Institute of Technology, and NTID students studying at the degree-level attend classes that are predominately populated by "hearing" students, students without an aural impairment. RIT is justly proud of its status as the only conventional "hearing" campus in the nation to educate large numbers of deaf students.

There are, of course, special problems associated with providing the means by which the deaf student can adapt to the environment of the conventional classroom. The instructor's lectures, inaudible to the deaf student, must be made meaningful by the use of interpreters and tutoring. The instructor must receive training in the consideration of the special needs of the deaf. Hearing students volunteer to make their lecture-notes available to the deaf by means of special self-duplicating notebooks. These are some of the devices that work toward placing the deaf degree-candidate on an equal basis with his hearing colleague.

But it is in *preparing* the NTID student for participation in regular RIT classes that the most interesting problems occur. The problem most directly related to pedagogy is inadequacy of educational background. The educational backgrounds of incoming NTID students generally contain serious deficits in the information, skills, and attitudes necessary to success in degree programs at RIT. It is the responsibility of NTID to rectify such instructional shortcomings before the deaf student is exposed to the RIT classroom. The responsibility for filling gaps in the student's secondary education falls to the NTID Vestibule Program. The project described in this article was done by the Computer Assisted Instruction Section of NTID in support of the Vestibule Program.

But how did the remediation of instructional deficiencies become a project in computer-assisted instruction? What special characteristics of the needs of NTID students make them susceptive to fulfillment by computer-assisted instruction? What special attributes does the computer have that justifies its use in the diagnosis and treatment of ailing educational backgrounds?

Initially, the answers to these questions will be given in general terms. First let us say that the remediation of instructional deficiencies became a CAI project because (1) the nature of the problem seemed appropriate to solution by computer, and (2) money had been included in the budget for such an activity.

If the solution of an educational problem is to be appropriate to the computer, the problem should be complex enough that its solution requires the computer's capacity for easily manipulating large blocks of data. In particular, the problem should need to apply the computer's almost infinite capability for conditional branching—the ability to sift, winnow, and select output data—that depend on the nature of the input. (For example, the computer can pre-

scribe appropriate instruction if the student's answer to a question indicated the need for such instruction. A different response to the same question might "branch" the student to a different remedial sequence on the same topic or to a new topic, depending on the correctness of the response.) It is, however, inappropriate to use the computer unless the computer is needed.

The implication of this stipulation that the computer's data-handling capability be *needed* is that it is usually not appropriate to use CAI unless there is great need for the program to adapt itself to a highly disparate set of individual needs. A situation appropriate to use of CAI might be one in which the subject area is inordinately broad and/or the educational backgrounds of students are extremely diverse.

Such a set of conditions exists among students entering the NTID. Probably the most prevalent instructional deficiencies are within the subject-area of pre-college mathematics. Many students display serious shortcomings in this area of study so vitally important to success in scientific and technical subjects. But, in any large group of students, the defects generally appear over the entire range of the subject from arithmetic to analytic geometry, with no means of predicting what deficiencies any individual will display. When the range of potential deficiency encompasses the entire mathematics curriculum from grades 8 through 12, the problem of ferreting out specific lacks in the mathematical background of any individual student is formidable. General review of topic-areas over a range as broad as the secondary mathematics curriculum is too time-consuming and inefficient to be practical. Therefore, the diagnosis of problem-areas must pin-point all the specific bottlenecks in the mathematics behavior of each individual student. The objective of such a precise diagnosis is that the resulting remedial instruction will encourage the student to acquire all the missing learning behaviors and only those behaviors that are truly missing from his behavioral repertoire. The result of such a program is that the student works full-time in areas of deficiency and is never exposed to material in which he has demonstrated proficiency.

Achieving this goal under the conditions described in preceding paragraphs is a task that is peculiarly appropriate to the unique capabilities of a computer-based education system. So the first task assigned to the CAI Section of NTID was to design an instructional system that would diagnose and remediate defects in the mathematical backgrounds of incoming and prospective NTID students. The instructional program by which we are pinpointing defects in the skills that comprise secondary mathematics has been named a Mathematics Diagnostic System (MDS).

This article not only describes the rationale, design, and structure of the MDS, but also provides some insight into its historical and philosophical aspects. The article also contains evaluative information and conclusions resulting from the first field tests.

The next few paragraphs very briefly describe the events that led to the creation and utilization of our Mathematics Diagnostic System (MDS).

HISTORY

With the passage of Public Law 89–36 in 1965, the National Technical Institute for the Deaf came into being. By 1968 the Rochester Institute of Technology had been chosen as its home, and a small cadre of key faculty-staff personnel became the organizational nucleus. The Applied Science Building was designated the location for NTID until permanent housing could be completed. June, 1968, brought the appointment of a director for the as yet nonexistent CAI Section of the NTID. In July, 1968, IBM representatives installed the IBM 1500 Instructional System. Between July and September of 1968, technically-oriented people joined the section, and by October, 1968, there were, in addition to the director, an operations manager, a systems programmer, a computer operator, and a keypunch operator.

By November of 1968, RIT and NTID faculty members were trying their hands at writing instructional programs for CAI, and student-assistants were translating these programs into a computer language called Coursewriter II. Lacking mediation by trained CAI course development personnel, this method of producing CAI materials proved to be not entirely satisfactory. As yet there were no CAI course development people on board.

Meanwhile, in July, at about the same time the 1500 System was being installed, steps were being taken that would eventuate in the Mathematics Diagnostic System (MDS). Once secondary mathematics was identified as an area of study appropriate to the computer, Dr. Robert L. Gering, then director of the CAI project, was apprised of the need to upgrade the mathematical backgrounds of NTID students ultimately to be registered in RIT's Calculus 75–101. Though he did not at that time know what the exact solution to such a monumental problem would be, Dr. Gering exhibited the educational acumen for which he is justly respected; he decided that, before a solution was attempted, the problem should be defined. He decided that before trying to provide

students with skills prerequisite to learning the calculus, these prerequisite skills had to be defined in detail and in behavioral terms.

This approach to producing a diagnostic-remedial instrument resulted in the Summer Mathematics Workship in July, 1968. The Workshop was charged with defining the skills necessary to the study of Calculus 75–101 and therefore consisted of members of the RIT mathematics faculty as well as members of the NTID faculty and staff. The efforts of this group were reasonably successful; it identified 23 subject-matter areas required for success in studying the calculus, pointed to some of the skills required, and indicated the emphasis to be placed on the various topics. As might have been expected, this activity did not produce a detailed statement of terminal behaviors; what resulted was a fairly comprehensive statement of projected course content.

Until December, 1968, the Summer Mathematics Workshop was the last activity that led directly to the development of the MDS.

In December, 1968, I was fortunate (I think), to be appointed CAI Course Development Leader at the NTID. Before coming to the NTID, I had spent six years at Friden, Inc. (a division of the Singer Company), applying educational technology to the problems of training service technicians. Although the position at Friden had required the creation and implementation of multi-media instructional systems, use of CAI had not yet been made available to us.When the Eastman Kodak Company offered the opportunity to extend the scope of the job to include using the computer in education, I joined that company as an education communications specialist. While at Kodak, I developed some of the CAI techniques that later were to facilitate the creation of the MDS.

Although no signs of the existence of an MDS were to become visible for several months, during the next eight months every working hour was devoted to some aspect of the MDS. There were questions of rationale and instructional strategy to be decided. Objectives (terminal behaviors) and subobjectives (enabling behaviors) had to be decided upon. And above all, when problems of strategy, rationale, and purpose had been resolved, there remained the overriding urgent question, "Who will do the enormous amount of rather specialized *work* that is required by this project?"

For, contrary to the somewhat naive expectations of the NTID administrators, this was not a job to be done by one lone CAI course development specialist. Even with the cooperation of a few subject-matter experts, if this task were to be completed in time to have maximum effect, it would require a substantial number of trained instructional programmers working overtime. Getting additional people was a sticky problem, for the table of organization provided for but a limited number of full-time CAI personnel; and most of these were allocated to technical support of course development activities. And if authorization to expand the course development staff could be obtained, where would we find competent CAI instructional programmers who were available and who would fit into a limited budget? People capable of this kind of work are rare, usually they are already employed, and they are expensive. It was obvious that if instructional programmers were to be added to the staff, they would have to be "home grown"—trained at the NTID. A time-consuming process and not conducive to producing a Mathematics Diagnostic System.

But what if development of the MDS and the training of personnel could be done simultaneously? Why not a sort of "Learn as you earn" operation in which the steps in developing the MDS coincided with the acquisition of expertise in educational technology? In fact, could the production of the MDS be organized in such a way that it could be fragmentized; so that it could be done by a large group of non-specialists having backgrounds appropriate to learning the rudiments of instructional programming? We decided that this approach was possible, and acted accordingly. By March, 1969, we had employed a group of ten people having scientific and/or mathematical backgrounds and had them working under the supervision of Mr. Alex Andres, an educational consultant from the University of Pittsburgh. The term of their employment was to end June 30, 1969. Because all but one member of the group were female, the group became known as the "Distaff Practicum".

By July 1, the Distaff Practicum had, within the constraint of a predetermined logic, produced the first draft of the diagnostic portion of the MDS. In addition, they had produced first drafts of objectives and subobjectives.Enough training had occurred that, by excellence of performance and by expressed inclination, three members of the Practicum were considered to be usable on a full-time basis. So when the term of the Distaff Practicum ended, it had not only achieved its goals but had also provided the CAI section of NTID with an organizational nucleus of three excellent people. In addition, one member of the Practicum, a chemist, was retained by NTID's Division of Instructional Affairs.

Previous paragraphs have presented the Distaff Practicum's work as being not only design and production but also as a sort of boot-strap training activity; the members were on a "learn as you earn" basis.

Most of this training was of informal nature. It is, however, worth mentioning at this point that some formal training occurred when 8 to 10 days of the group's time was spent in going through the Instructional Technology Workshop (ITW) produced by General Programmed Teaching. Bill Deterline, president of GPT, was the guiding spirit behind producing the ITW workshop.

Besides ITW, the only other formal training instruments made available to CAI instructional programmers at NTID have been standard texts by Skinner, Lysaught and Williams, Mager, Bloom, Markle, and others. One text, not so well known as the others, is this manual *The Creation of Remedial Courses for CAI*, written at NTID between January and June of 1969. While written primarily as a training instrument, it functions equally well as documentation of the principles, policies, and procedures used by CAI-NTID to ensure effective instruction by means of the computer.

We don't make any extravagant claims for this little manual—it is just a "how to" book on causing the computer to be a tutor. Perhaps this much should be said on the subject of manuals that deal with the "nuts and bolts" of CAI instructional programming: They are scarce. The scarcity of such materials probably accounts for most of the interest shown on our manual. For example, I was surprised and flattered when Dr. Gabriel Ofiesh, Director of the Center for Educational Technology of Catholic University, requested multiple copies for use in graduate courses in educational technology. IBM has requested permission to use the manual in some of its training activities. If you have an interest in, or a use for, such a handbook, it will be sent to you on request.

Using the manual to set the ground-rules for completing the first rough approximation of the MDS, the months of July through September, 1969, were spent in revising and polishing existing diagnostic test items, formalizing remedial prescriptions, and translating the diagnostic portion of the MDS into computer language, Coursewriter II. (In the first version of MDS, remedial prescriptions were administered by an instructor-proctor.)

By the last week in September, the writing and programming of the MDS was sufficiently advanced that we were ready to work with our first group of test subjects; we were ready to begin the initial field test. On September 22, 1969, eight deaf students and one hearing student sat down at the computer terminals and began item 1 of segment 1 of the Mathematics Diagnostic System (MDS). Three months later, six of the nine were considered well-prepared for entry into Calculus 75–101. Because they have been in calculus for a short time, it is too early to report positive results, but the prognosis seems favorable.

While the students were working through the MDS during the fall quarter of the school year, they were producing performance records that would be the basis for revising, improving, and extending the MDS. The winter quarter of 1970 was spent in updating the MDS from the original version, now known as MDS-V1, to the more effective, more reliable, more completely computerized version designated MDS-V2. No new students were accepted during the winter quarter so that this major operation, the production of MDS-V2, could be as complete as possible by the time the next group of students arrived in March of 1970.

When RIT's spring quarter began on March 23, there were 20 NTID students on-line with the first segments of the MDS-V2. For a number of reasons, among the best of which was that our computerated classroom has but 10 student-positions, the group was divided into two classes, each containing 10 students. Because MDS-V2 eliminates much of the administrative and clerical activity that had wasted the instructor's time in MDS-V1, it was thought that the ratio of students to instructor could be raised from 4 to 1 in MDS-V1 to 10 to 1 in MDS-V2. Another significant difference in the administration of MDS-V2 was that, while remediation in MDS-V1 had been tutored by the course authors, the instructors for MDS-V2 would be members of the RIT and NTID mathematics faculties.

At this point in our narrative perhaps some mention must be made of the role of the instructor-proctor in the effective presentation of MDS-V2. As "proctor," he keeps the mechanical and technical aspect of the course going smoothly; he sees that each student is having no difficulty with operating the terminals; he cooperates with the technical staff in operating the equipment; he evaluates records of student progress. These are the more mundane and mechanical features of the job that require some technical ability but do not evoke much creative effort.

But as "instructor" his knowledge of mathematics and his tutorial effectiveness may well be strained to the limit. This demand on the tutorial skill of the instructor-proctor results from the fact that the nature of the MDS produces a mode of instruction demanding that the full-time job of the teacher is to teach—not to present instructional materials. The computer presents diagnostic materials, prescribes remediation, and tests to see whether or not the desired learning behaviors have occurred. Because there is no such thing as a perfect program, some of the program must fail with some of the students. When

the program fails to produce the desired results, the student is directed by the program to consult the instructor. There is another, happier way that can produce tutorial activity; the program may have so stimulated a student's imagination and enthusiasm for the topic that he requires more information than has been provided. When this happens, the student may voluntarily seek the guidance of his instructor in going beyond the limits of his program. Whatever the reasons for the demands placed on his teaching skills, the instructor who operates in this setting is faced with an intense, varied, and—I should think—rewarding experience.

While the combined activities of human and machine are working to produce the most effective preparation for calculus that is within their power, records are being produced that will permit the effectiveness of the course to be evaluated. The results of MDS-V1, considered by its authors to be a first rough approximation, has been empirically evaluated and revised on the basis of student performance records. Further evaluation of MDS-V1 will be possible when it becomes clear how many of the difficulties experienced in calculus by MDS-V1 students may be related to lack of preparation—to failure of the MDS-V1. Such crudity of method was thought to be not only adequate to the circumstances of MDS-V1 but necessary if MDS-V2 were to appear on schedule.

Because MDS-V2 is considered to be reasonably close to a finished product, evaluation of this instrument will be carried out by the NTID's Research and Training staff. It is hoped that applying the talents of this group to determine factors of reliability and validity will produce data that are useful in improving subsequent versions of the MDS. Data for this project being gathered now during the spring quarter should produce usable results before the opening of the fall quarter in September, 1970. By the winter quarter (1970–1971), it is expected that the MDS will have reached a point of stabilization; that subsequent versions will be extensions and additions to MDS rather than revisions.

While determining the effectiveness of the MDS in educating deaf students is of first priority, there is reason to believe that it could be equally effective in performing its function for hearing students who anticipate entering first year calculus at RIT. To test this belief, during the 1970 summer quarter the MDS will be given to two groups of regular RIT students who would otherwise have been enrolled in pre-calculus math courses. If this experiment proves to be successful, it will lend a universality to the application of the MDS that will greatly enhance its economic feasibility—always an important factor in determining the practicality of any project in computerated education.

Practicality? Well, we here at the NTID think that the MDS, if not immediately justifiable on purely economic grounds, points the way to innovations in education of the deaf that will someday make CAI projects feasible from all points of view, educational as well as economical. Although it is much too early to label the MDS an unqualified success, its existence and apparent effectiveness are stimulating RIT and NTID faculty members to cooperate in similar projects. Several such projects are already under way; courses in thermodynamics and circuit design are being produced by the combined efforts of RIT faculty and the CAI Course Development that could herald RIT's emergence as a front-runner in the field of educational technology as well as in the education of the deaf. That is what we hope, and that is the goal toward which we shall continue to work.

What I have told you thus far has been a history and description of the first major project undertaken by the CAI Section of the National Technical Institute for the Deaf. The information you have is but part of a complete report on the MDS that includes a description of the underlying philosophy of course-design, the rationale and design-principle, and a report on initial field-tests. If any of those present are interested in these details, please leave your name and address and you will be sent a copy of the complete report.

Also included in the final report are details of the computing system by which the MDS is administered. This part of the report was prepared by Mr. Paul Vonhof, Technical Support Leader for NTID's CAI Section.

THE IBM 1500 INSTRUCTIONAL SYSTEM

The IBM 1500 Instructional System consists of two major elements: a "hardware" element and a "software" element. All the items of equipment—mechanical, electro-mechanical, electrical, and electronic—make up the hardware element of the 1500 System. The software element consists of the various computer programs that instruct the hardware in what it must do to achieve the results desired from the 1500 System. Both elements must be present if the system is to work; the hardware is just expensive junk without the software, and the software in meaningless symbology without the hardware. It takes both elements working together as a complete system to produce the instructional magic of which the System is capable.

For there is an element of magic about a machine-based instructional system that adapts itself to the specific needs of individual students; that makes allowances for individual differences in learning rates, entrance behaviors, and intellectual capacities; that can perform this service for as many as thirty-two students, not all of whom are working on the same course; that records and reproduces data on course-performance and student performance; that, in short, provides educational services the quality of which is limited only by the capability and imagination of the course-designer.

A machine such as we have been describing is, however, only a machine. In the final analysis, it is no better than the people who use it. Its effectiveness depends very much on the interaction between the people who consume its output, the people who control its operation, and the people who design and implement its input. These are the three general categories of users served by the System: the student, the instructor-proctor, and the course-author.

Of these three users, the student comes first, for he is the ultimate consumer, the reason for the existence of the System. The System prescribes or presents the instructional materials that uniquely fulfill the needs of the individual student. Instructions may reach the student as text on the CRT (cathode-ray tube) display. Or, the CRT may contain graphic information with which the student must interact. Student interaction with the CRT may require the use of the keyboard or it may require that responses be made by pointing at the display with a "light-pen." (The light-pen communicates to the computer the exact location of the spot at which the pen touches the screen.) The student also may receive textual or graphic information from the image-projector. (The image-projector is a random-access film-strip projector with full-color capability.) The IBM 1500 Instructional System also includes an audio capability that is not used at the NTID.

The System, of course, accepts and processes all student responses, whether they are entered by means of the keyboard or the light-pen. The material presented to the student usually is contingent upon the nature of the preceding student response. It should be apparent, therefore, that input and output are each a part of a total continuum in which each determines the other. This characteristic of the System allows us to design courses having a high degree of adaptiveness to individual student-needs; it is the characteristic of the System that ensures the ability to provide instruction that is truly individualized.

To enhance the interactive nature of the CAI program, the System provides for student comments at any time. It also permits the student to request help from the instructor-proctor at any time the student thinks he needs it.

While the student interacts with the System, the instructor-proctor controls it. The duties of the instructor-proctor are covered in a fair amount of detail within the body of the text of this article, All that needs to be added is that, because students and course-authors may simultaneously be using the System, the proctor exerts control over the activities of both students and course-authors. The term "course-author" includes both the instructional programmer and the Coursewriter programmer. The System provides the author with services that are essential to him. It "assembles" his courses in a storage device—assembles Coursewriter II into a format that can be interpreted by the machine. During the execution, it "interprets" his courses—implements machine-language instructions. It presents his courses as specified by the logic of his program. It analyzes progress. The author can make changes, additions, and deletions that are based on information received from the System itself.

All the activities and services to author, proctor, and student depend on human interaction with the hardware/software complex that is the IBM 1500 Instructional System. Having given some indication of what the System does, the remainder of this sup-

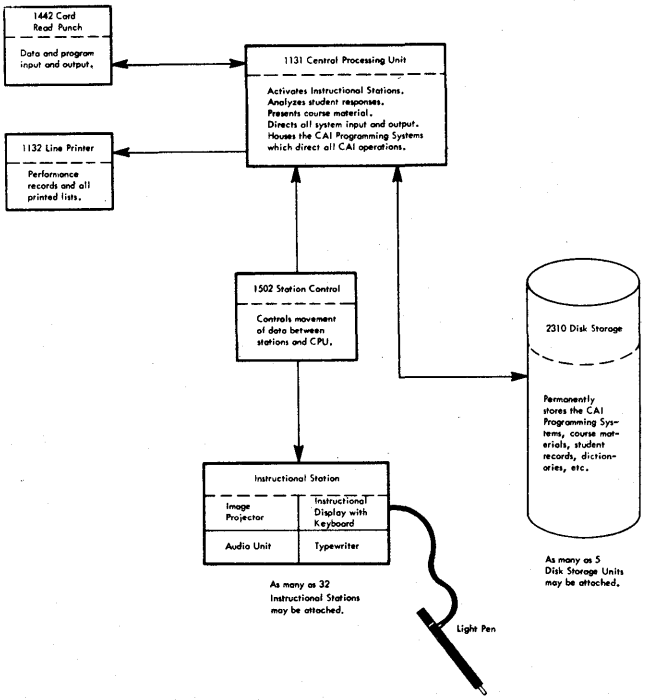


Figure 1—Interrelationship of hardware components

plement describes its hardware and software components.

Hardware

It is the hardware of the System that is visible to the consumer or observer. He becomes familiar with the keyboards, the light-pen, the CRT display, and the film-strip projector—all of the facilities to be found in the computerized classroom. If he is a special friend of the operations manager, the observer may be allowed into the machine room, where he sees the devices by which course materials are stored—the magnetic disk drives and the magnetic tape units. He sees the card reader/punch by which course materials and computer software are created, assembled, and entered into the system. Also located in the machine room are the central processor and the station control unit, which directs the flow of course materials from the central processor to the appropriate terminals. A printer prepares performance recordings and other lists required in operating the System. (See Figure 1—a pictorial representation of how the various units relate to one another.)

Software

The 1500 software, expressed more formally as the 1500 Programming Systems, supports the 1500 hardware. It supervises and expedites all operations performed by the IBM 1500 Instructional System. In addition to the Coursewriter II System described in Appendix B, it includes six major programs that cooperate to keep the 1500 System operating efficiently. These programs are:

1. Main Control Programs.
2. Station Command Processing Programs.
3. CAI Processing Programs.
4. CAI Support Programs.
5. CAI Utility Programs.
6. System Utility Programs.

The Main Control Programs are supervisory programs and form the basic operating system. These programs provide scheduling service to each instructional station that demands individualized service in handling inquiries and responses. They accumulate performance records and, when necessary, provide the user with diagnostic information about the operation. The operating system routinely stores and maintains all data needed by the programs executed under its control.

Station Command Processing Programs provide the language link between the system user and the computer. It facilitates all communications between the student terminal and the computer.

The CAI Processing Program is the major CAI application program, Coursewriter II. This program contains the Coursewriter II interpreter, which executes the user's assembled course and interacts with the students at the terminals. The CAI Processing Program can present textual material on the typewriter or display screen, present problems, process student responses, and operate the image projector. It performs arithmetic and logical operations. The CAI Processing Program also can be called upon to set and to interrogate a response timer, a device for recording the speed with which students respond to questions.

The Coursewriter II assembler translates Coursewriter II language statements into a form acceptable to the interpreter. The assembler is an important program within the CAI Support Programs section of the programming systems. Material can be inputted by means of either keyboard entry or card input. CAI Support Programs also allow for modifications to courses.

The CAI Utility Programs allow certain special background jobs to be done; these jobs support the operations for organizing courses on magnetic disk.

The System Utility Programs provide the functions necessary to preparing and maintaining systems package.

All of the software briefly described in the preceding paragraphs are necessary to control the multitude of operations demanded of the hardware. Students taking courses, authors entering Coursewriter statements, proctors sending supervisory commands, and operations people scheduling background jobs—each makes demands on the programs included in the 1500 Programming Systems.

SUMMARY

The combined facilities of the 1500 hardware and software present a versatile tool for instructional techniques. Thirty-two students, each working independently on a different problem or program, can time-share the system. Textual material, full-color film, and audio messages can be presented to the instructional stations under computer control. The computer automatically provides file maintenance to course and user's records. Course and student information is

stored and retrieved as required by each station when it is serviced by the computer.

The operating system controls all interaction between the students and the course material being presented. The answer analysis of each problem and the infinite branching through the course is automatically performed by the CAI processing programs. The system allows for a standard dictionary of 128 characters, three special 128-character dictionaries, and three graphic sets of 64 symbols each. Coursewriter allows the system to display alternate dictionaries or graphics during course presentation. The interactive graphic capability allows the student to point at a position on the CRT and have the system determine the co-ordinates of the pen response. Finally, the system is flexible in its software capabilities and allows the user to make additions or extensions to it.

But, above all, the IBM 1500 Instructional System provides the course development specialist with a means of presenting the student with a course the effectiveness of which is limited only by human ingenuity. The potential of the System has only just been scratched; CAI itself is but an infant-giant.

Teaching digital system design with a minicomputer

by MARVIN C. WOODFILL

Arizona State University
Tempe, Arizona

INTRODUCTION

Design education requires a study of the "art" involved in a given discipline in addition to the study of its "science." Textbooks tend to concentrate on the science and leave the art as 'an exercise for the student.' One solution to this problem is the use of a "real" example of the discipline involved as a teaching aid and laboratory tool. With this approach the art and science can be taught in integrated form and with the given system as a base, these principles can be readily extrapolated to other systems. The "broad brush" approach to design education without a base of knowledge tends to leave the student little net gain.

In digital system design education a minicomputer can provide a very good example of the discipline. Minicomputers are superior to "larger" systems for this use because they are small enough in size and complexity to be easily understood, but with the addition of suitable software can provide a very useful computational tool. The Digital Systems Laboratory (DSL) at Arizona State University was originated in 1966 with the aid of an NSF Educational Equipment Grant which provided the funds for the computer which forms the nucleus of this laboratory. The DSL was conceived to provide: A very flexible teaching aid; a prototype digital system which students could study in detail; and a vehicle for student hardware, software, and application experiments, studies, and projects.

THE COMPUTER

A DATA 620 manufactured by Data Machines Incorporated (now Varian Data Machines) was the computer purchased for this laboratory. The DATA 620 (currently available in integrated form as the DATA 620/i) is a binary, parallel, single address, bus organized, 16 bit, general purpose digital computer with an extensive command repertoire.[1] The computer was purchased with a 4,096 word, 16 bit memory with memory cycle time of 1.8 microseconds. The machine design provides a bus oriented party-line I/O and a readily expandable interrupt capability. A block diagram for this computer is shown in Figure 1. The architecture of this computer illustrates many of the "pin limited" design constraints of large scale integration system design which provides a "state of the art" system model constructed with transistors.

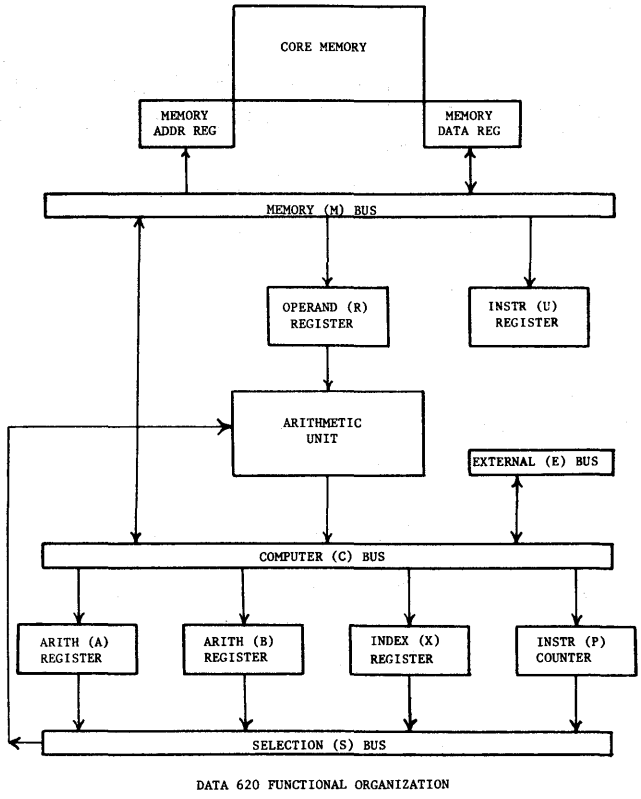


Figure 1—Data 620 functional organization

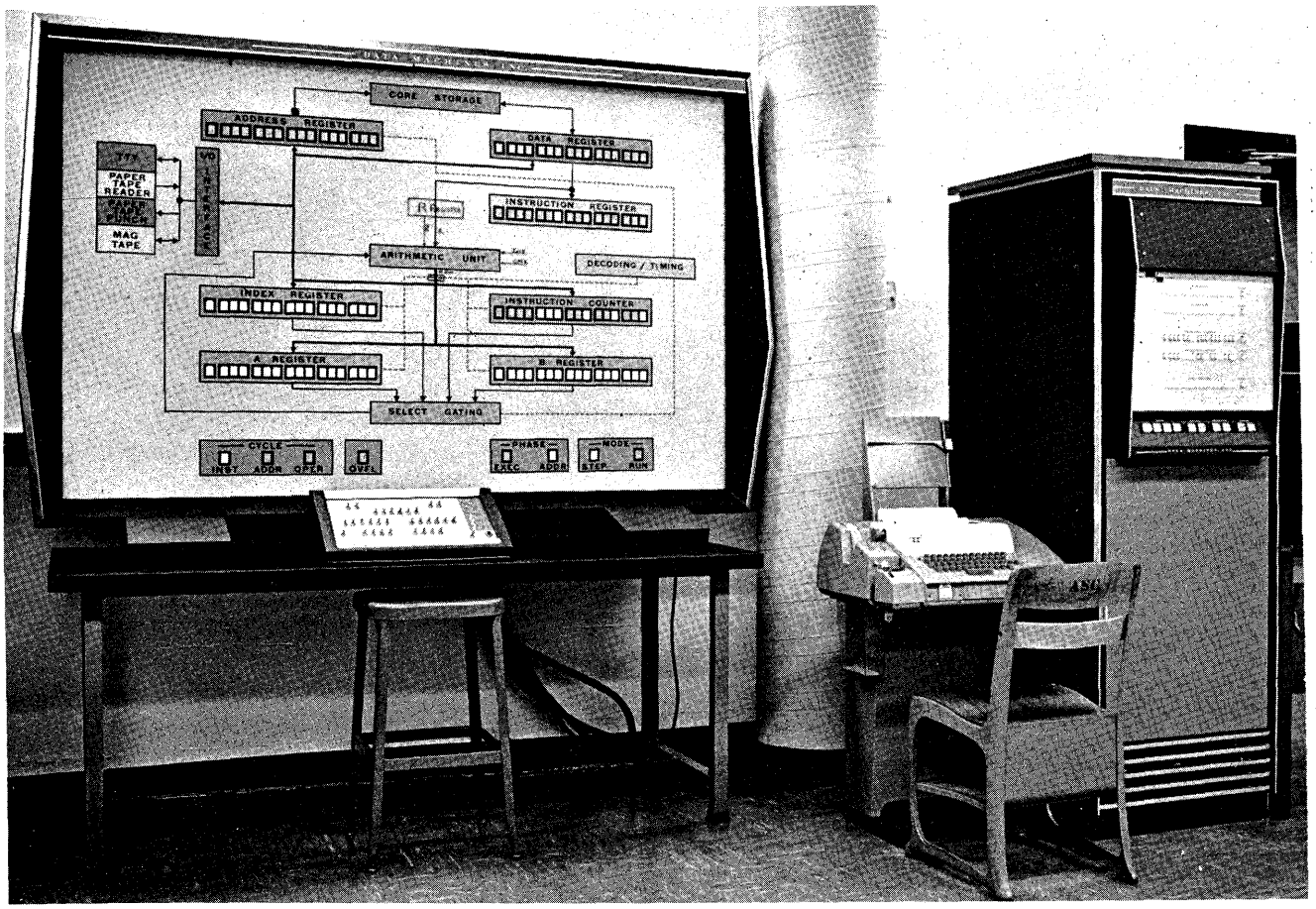


Figure 2—The initial systom

This computer was purchased with several options which greatly enhanced its value as an instructional tool. (A picture of the original system is shown in Figure 2.) These options include: (1) A large classroom display, (2) A micro-exec bus providing external processor control; (3) the pulse instruction execution option; and (4) A real-time clock. The classroom display shows the contents of the registers in the processor and memory and the state of the system's clocks. The communication paths and interconnections are also illustrated on this board. This board is used extensively in the lecture as a training aid and demonstration media. The micro-exec bus provides a means of external control of the processor micro-signal execution. During demonstrations of the operation of the processor logic, a switch box is used to selectively enable the processor micro-signals to illustrate the execution of various instructions and operations. This facility is also an invaluable maintenance tool. The pulse execution option provides the user with the ability to supply manually, by a switch closure, the pulses normally supplied by the 2.2 MHz master clock. This allows the observation of the execution of machine instructions at the microstep level which provides a very graphic display of the operation of the processor logic. The real-time clock which provides a time base interrupt capability is used to illustrate real-time operation and provide "clever" displays, i.e., the functioning of a multiplication algorithm in slow motion. The system also contains a maintenance panel which facilitates oscilloscope observation of the major processor microsignals for display or maintenance use.

These options are very convenient but should not be considered an absolute necessity. Most minicomputers (including the DATA 620/i) have, for example, a much more limited console than the DSL system.

However, the most meaningful interaction a student has with the computer is through the on-line teletype and other peripherials so a limited register display is inconvenient but not disabling. It should also be noted that the DATA 620 is only one example of many minicomputers that could be used in a laboratory of this type.

The original system functioned well as a teaching aid and display device but was not a satisfactory "hands-on" tool for student laboratory use, because of its slow and inefficient I/O facilities and its limited software.

THE SYSTEM HARDWARE

The first problem to be solved in the realization of a useful student tool was alleviation of the I/O bottleneck. An off-line ASR-35 teletype was purchased to allow off-line preparation of student tapes but the slow input speed (25 char/second) of the on-line ASR-33 tape reader still limited system through-put. The graduate college at Arizona State University requires all Master of Science students to write an Engineering Report or Thesis as a final step in fulfillment of graduation requirements. The expansion of the hardware of this computer system has provided and will continue to provide numerous Engineering Report and Thesis project topics. To date, the following peripheral hardware has been interfaced to the system in this fashion: (1) A paper-tape reader and punch with 50 character/second capability (the reader and punch was an industrial gift); (2) A Holly-Line Printer (also an industrial gift) with 300 character/second capability; (3) A 1000 character/second paper-tape reader; and (4) A 150 character/second paper-tape punch (the latter two were purchased with funds remaining in the NSF Grant); (5) A patch panel facility for the micro-exec bus to allow the implementation of patched machine instructions (complex algorithms accomplished as one machine instruction). Many other projects have involved the modification of the basic command structure of the computer. The conditional jump structure of the machine was augmented to essentially double the number of useful jump conditions. The mechanical sense switches of the original computer were replaced with logical switches (flip-flops) under program and/or mechanical control. Extended addressing capability was added which provides direct, indirect, and indexed addressing to 32K words of memory was added. Fixed point multiply and divide hardware was added to the central processor. A hardware parity testing instruction and hardware instruction to save the volatile state indicators (overflow and sense switches) was added. Logic was added which causes the execution of a halt instruction to control return to the Operating System. These changes and others have resulted in a dynamic system hardware configuration tailored to the needs of the system users. See Appendix A for the DSL instruction format. Although the original computer was composed of "discrete parts" most of the hardware modification projects have been accomplished with integrated logic. The hardware of the system is continually being improved and expanded.

The following hardware projects are currently in progress: (1) The addition of an auxiliary 64K words of primary memory to the CPU (the memory was an industrial gift); (2) The interface of a magnetic tape system of four tape drives (also an industrial gift); (3) The interface of a magnetic drum memory (also an industrial gift); (4) The addition of a GE 115 computer with card reader and line-printer as a secondary processor (also a gift); (5) The construction of a priority interrupt facility and analog to digital and digital to analog conversion facilities. (6) The interface of three additional ASR-33 teletypes which will provide the basis for a time-sharing facility.

THE SYSTEM SOFTWARE

The development of adequate software to interface the user with the physical capabilities of the system is a never ending task. As each hardware change is realized the system software must be redesigned to take fullest advantage of the hardware. In a system with 4K words of main memory and no secondary storage, the primary system software design goal was to provide maximum capability in minimum memory. All of the software used in this system was designed expressly for this system. Figure 3 shows the normal configuration of memory in the laboratory system.

The primary operating system is called the **M**onitor **U**tility **D**river **PAC**kage (MUD PAC) which provides: (1) Subroutines used by students (and the other system software) to accomplish I/O (monitor); (2) A comprehensive set of on-line debugging utility routines; and (3) A driver system which allows on-line teletype control of the entire system. Teletype control is essential to the efficient operation and maximum through-put of the system. The MUD PAC was designed to provide the user with all the computational tools necessary for the operation of the system with minimum interference to the user. The MUD PAC requires 1400_8 or 768_{10} memory locations.

The monitor is organized into 3 subroutines: CHAR,

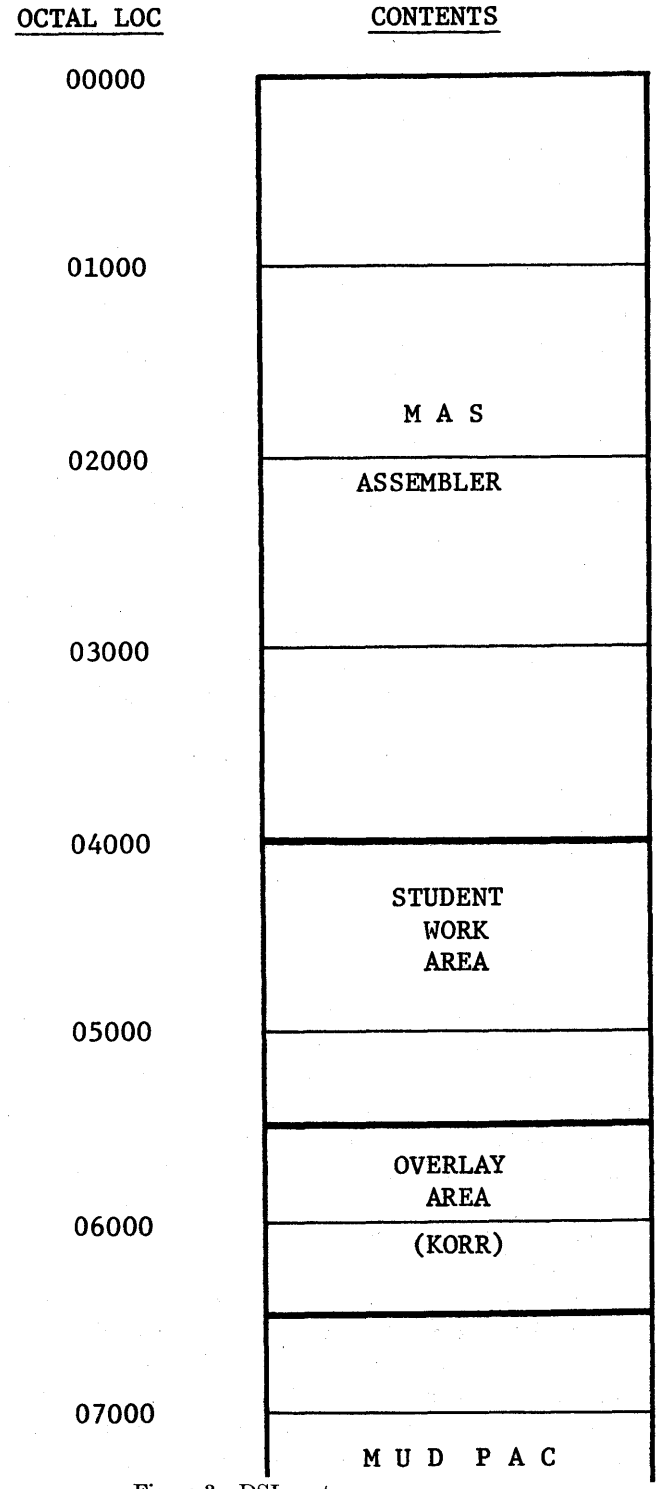


Figure 3—DSL system memory map

for character (8-bit) I/O; NUMB, for octal number (6 digit) I/O; and MESG, for message (strings of ASCII characters) I/O. In all cases, the setting of the sense switches, which is under program control, designates the I/O device involved and the direction of data transfer. See Appendix B for further subroutine description and the coding used for the accomplishment of I/O operations under monitor control.

The utilities provided include the usual types of utilities: Pseudo register manipulation, memory change, memory initialization, memory dump, selective memory search, location flagging, and program trace capabilities. In addition, the routines to load and punch object information in the object formats of this system are provided. The more nonconventional of these formats is known as monitor format and is an inefficient format using octal numbers represented as ASCII characters on paper tape to fill sequential memory locations. This format is used by the students in the initial phases of their laboratory course to enter their machine language programs. This object format can be generated on an off-line teletype and thus provides much better turn around time for machine language programs than would manual entry.

The driver routine accepts keyboard inputs of single letter mnemonics, which correspond to one of the defined MUD PAC operations, then the driver accepts the required number of parameters (octal numbers from the keyboard) and transfers computer control to the routine which performs the function of the particular mnemonic. MUD PAC mnemonics are listed in Appendix B. In addition to the utilities, driver mnemonics are also provided to accomplish linkage to the other system software, namely the Monitor Assembly System (MAS) assembler and the overlay area programs, usually the source tape correction program (KORR).

The MAS assembler was expressly made to take advantage of the hardware of this particular system. MAS possesses all the capabilities available in most non-macro assemblers including page titling and source line numbering. Current MAS operation codes are listed in Appendix C. MAS contains predefined values for the monitor subroutines and its syntax has been chosen to complement the machine language structure of the computer. The fixed portion of the MAS assembler occupies less than 4000_8 or 2048_{10} memory locations and uses the student work area for temporary storage.

The overlay area of the lab system normally contains the KORR source tape correction which provides the efficient source correction capability available automatically only in a punched card environment. Whenever a source tape is assembled or pre-listed (which can be done with KORR) each source line is assigned a decimal identification number determined by its sequential position on the source tape. Using these numbers and the KORR mnemonics provided, during the duplication of a source tape any combination of source lines can be deleted, replaced, or added. This process

can be accomplished under on-line teletype control or by preparing the corrections off-line and using the auxiliary paper-tape reader as the on-line correction device. The latter approach is used in the laboratory to increase system through-put.

The efficiency of expressly designed software is illustrated by the MAS assembler. In comparison with the DAS assembler (supplied by the manufacturer) for this system: (1) Both assemblers are self-contained (perform their own I/O), however for DAS all I/O was through the ASR-33 teletype (since that was the only I/O device in the original system), whereas MAS uses the Remex Reader, Holly Line Printer and Tally Punch; (2) The fixed portion of DAS used 5776_8 or 3070_{10} memory locations, MAS uses 04000_8 or 2048_{10}; (3) DAS had 134_{10} defined op-codes, MAS has 163_{10}; (4) DAS allowed only four character labels, MAS allows 6 character labels; (5) Within the 4K memory DAS would allow the assembly of a program with 224_{10}-four letter labels, MAS will allow 640_{10}-six letter labels. (6) MAS provides source line numbering, parity checking of source input, page titling, and automatic symbolic formatting. These were not provided in DAS. The MAS assembler then is clearly superior to the DAS assembler in every respect. In defense of the manufacturer, the DAS assembler is conditionally assembled from an assembler which can be made to match the hardware configuration of any DATA 620 system and thus is necessarily less efficient in any given system. This assembler was designed and coded as part of the graduate software design course.

SYSTEM USE

This laboratory system is used extensively in a senior elective/graduate course in Logical Systems Design (enrollment about 80 per year) and in a graduate Integrated Systems Engineering Program (enrollment about 20). Both courses are preceded by a course in Logical Component Design using Chu,[2] and followed by graduate courses in Digital System Hardware Design and Digital System Software Design.

The logical systems design courses have no programming prerequisite and the students begin by learning to program the DSL system in assembly language and machine language. This process takes about half of the course and results in the students acquiring programming ability, knowledge of the use of the arithmetic hardware, and knowledge of the applications of a small computer. During the second half the students become familiar with the micro-level processor signals, system timing signals, system logic, instruction execution logic and the significance and handling of interrupts. The class is conducted in the classroom-laboratory and frequent use is made of the available display tools. In the laboratory portion of the course, the students practice their skills in the writing, testing, and debugging of programs of increasing complexity, employing an increasing amount of the system hardware and software. In a typical laboratory sequence the students first lab (about the second week of the course) will include a simple program in machine language entered and executed manually by the students through the console (with no software). Each subsequent lab period the students are given more complex problems and more software to work with. By about mid-term the students are using the total system and by the end of the semester the students are solving problems requiring several weeks of coding and debugging. The students also spend some laboratory periods involved with the hardware signals and the micro-exec bus. When the students finish the design course they have a common familiarity with the machine and assembly language, use, logic, logical language, and micro-signals of this computer system and limited exposure to alternate techniques. Primarily, they have a solid base of knowledge about the processor of this particular computer and an intuitive understanding of what a computer can do and how the internal functions are accomplished.

The emphasis in the subsequent graduate hardware and software design courses is system oriented but they are not restricted to the laboratory system, although the base of knowledge or common language developed in the DSL system is exploited. Even though these courses are engineering based, they have no engineering prerequisites and attract computer science students from many other disciplines.

FUTURE PLANS

The most urgent problem presently facing the Digital Systems Lab is the student load. With 3 hour laboratory periods of 10 students each and one on-line teletype, each student has about 18 minutes of computer time per week to accomplish his assigned tasks. To provide some relief a DSL DATA 620 simulator was built and is available on time sharing service from a GE 600 system. This system is not the ultimate answer since it tends to remove the student from the hardware completely and it suffers from the "finite" lag problem familiar to time share users. For the purpose of the advanced students, however, the simulator provides an excellent algorithm testing media.

A more tractable system will be achieved with the

realization of a local time sharing capability. The system will consist of three additional ASR-33 teletypes, which will provide a total of four "user ports" to the system. The design goal is to provide each user with a 4K memory of his own, a utility capability roughly equal to the present system, and, of course, the feeling that each user has the complete system at his command. Needless to say, successful completion of this project depends upon the realization of additional memory capability. The final system will of necessity separate the user from the hardware to a greater extent than the present system, but the alternative would be to duplicate the hardware of the system and economics forbid that solution. Since each system addition makes the system more complex and difficult for the students to comprehend, the basic 4K machine will continue to provide the basis of instruction for the beginning course, and the more sophisticated capabilities will be exploited primarily in subsequent courses.

The digital systems laboratory has proven to be a very successful and well received addition to a previously pedantic curriculum. The choice of a minicomputer for a digital systems design education laboratory model is considered essential since such a machine is large enough to perform many useful functions, but small enough to be understood by an average student, and small enough so other uses need not be found to justify its cost. The development of this facility has been a painstaking process of evolution and improvement requiring a considerable amount of time and effort. However, the knowledge and experience gained by all concerned has certainly justified the investment.

REFERENCES

1 ———
Varian Data 620/i computer manual
Varian Data Machines Irvine California 1968

2 Y CHU
Digital computer design fundamentals
McGraw-Hill Book Company Inc New York New York 1962

APPENDIX A

DSL DATA 620 INSTRUCTION FORMAT

INSTRUCTION	OP CODE U15-U12	M FIELD U11-U9	U8	A FIELD MICRO INSTRUCTIONS U7	U6	U5	U4	U3	U2	U1	U0
HALT	00	0									
JUMP	00	1	S	S	S	X	B	A	O	A	C
JUMP&MARK	00	2	S	S	S	R	R	R	F	R	M
EXECUTE	00	3	3	2	1	0	0	0		+	P
SHIFT	00	4	(1) A&B	A	RT	LOG	SHIFT COUNT				
			(0) AorB	B	LF	ARTH					
REGISTER CHANGE	00	5	Cond on OF	00 TRAN 01 INCR 10 COMP 11 DECR		SOURCE XR	SOURCE BR	SOURCE AR	DESTINATION XR	DESTINATION BR	DESTINATION AR
IMMEDIATE	00	6	0	0	OP CODE FOR SW INSTR X	X	X	X	0	0	0
EXTENDED ADDR	00	6	0	0	X	X	X	X	M FIELD FOR SW Y	Y	Y
INDICATOR CHANGE	00	7	CHG OF	CHG SS	SOF (1) ROF (0)	RS3	RS2	RS1	SS3	SS2	SS1

EXT CONTROL	10	0	FUNCTION	DEVICE ADDRESS
SENSE	10	1		
DATA INP	10	2	CLEAR	DEVICE ADDRESS
DATA OUT	10	3	REG	00 MEM 01 AR 10 BR 11 A&B

LOAD AR	01	M	ADDRESS (A) FIELD
LOAD BR	02		
LOAD XR	03		
INCR MEM	04		M FIELD CODES
STORE AR	05		0-3 DIRECT
STORE BR	06		4 REL to PR
STORE XR	07		5 REL to XR
INCL OR	11		6 REL to BR
ADD	12		7 INDIRECT
EXCL OR	13		
SUB	14		
AND	15		
MUL	16		
DIV	17		

APPENDIX B

MONITOR UTILITY DRIVER PACKAGE (MUD PAC) DIRECTORY

Monitor Subroutines:

CHAR:	CALL* CHAR Loc. 07000	Inputs one 8 bit char into LS 8 bits of cleared AR or Outputs AR, MS 8 bits first then LS 8 bits.
NUMB:	CALL* NUMB Loc. 07001	Inputs a terminated octal number (in ASCII char) into BR (right justified), or outputs the 6 ASCII char representing the input value of BR plus 2 ASCII spaces if AR=0 or the 2 characters in AR if ≠0. Neither effect AR or XR.
MESG:	CALL* MESG, M Loc. 07002	Inputs a string of ASCII chars delinated by ' and stores them (2 char/word) in sequential locations starting at M and indicates EOM with a cleared loc, or outputs a string of chars starting at M and returns when a cleared loc is found. Neither effect AR, BR, or XR.

Device Coding:

SS1 represents mode: on is output—off is input

SS2 & SS3 determine device according to the following: 1 is on, 0 is off.

SS3	SS2	Device: (Input/Output)
0	0	T-R R/P
0	1	R-M/R/P
1	0	TTY Tape/ Line Pntr
1	1	TTY Keyb/ TTY Type

Utility and Driver Mnemonics:

Parameters are STRT, STOP, KEY, MASK Registers are AR, BR, XR, PR

Code:	Parm:	Description:
A	0	change AR
B	0	change BR
C	1	Change STRT
D	2	Dump loc STRT through STOP in SYST FMT
E	1	Elect (change) pseudo sense switches and overflow.
F	1	Flag location STRT
G	1	Go to loc in STRT
H	0	Halt return, go to PR
I	3	Initialize STRT through STOP with KEY
J	1	Jump to MAS, STRT determines pass
K	0	Korrect, source tape correction routine
L	1	Load or compare object tape SS1 = L/C (0/1). SS2 = Mont/SYST (0/1) format.
M	2	Memory dump of STRT thru STOP on line pntr.
N	0	New—Load new MUD PAC with BOOT loader.
O	1	Overlay linkage STRT is linkage code.
P	2	Punch loc STRT thru STOP in Monitor fmt.
Q	0	Query parameters and reg. (display all).
R	0	display Registers
S	4	Search STRT thru STOP looking for KEY considering only bits with 1 in MASK
T	2	Trace from STRT to STOP

U	0	Untrace
V	0	Vector reestablish interrupt vectors
W	0	Convert flex tape to ASCII.
X	0	change XR
Y	0	list entry to Korrect
Z	0	Zero all pseudo reg & reset pseudo overflow and sense switches.

APPENDIX C

MONITOR ASSEMBLY SYSTEM (MAS) MNEMONICS

SINGLE WORD ADDRESSING

LDA	LOAD AR
LDB	LOAD BR
LDX	LOAD XR
INR	INCREMENT AND REPLACE
STA	STORE AR
STB	STORE BR
STX	STORE XR
ORA	INCL OR TO AR
ADD	ADD TO AR
ERA	EXCL OR TO AR
SUB	SUB FROM AR
ANA	AND TO AR
MUL	MUL BR BY MEM
DIV	DIV A-B BY MEM

DOUBLE WORD NON-ADDRESSING

LDAI	LOAD AR IMMED
LDBI	LOAD BR IMMED
LDXI	LOAD XR IMMED
INRI	INCR AND REP IMMED
STAI	STORE AR IMMED
STBI	STORE BR IMMED
STXI	STORE XR IMMED
ORAI	INCL OR TO AR IMMED
ADDI	ADD TO AR IMMEDIATE
ERAI	ENCL OR TO AR IMMED
SUBI	SUB FROM AR IMMED
ANAI	AND TO AR IMMEDIATE
MULI	MULT IMMEDIATE
DIVI	DIV IMMEDIATE
LDAE	LOAD AR EXTENDED
LDBE	LOAD BR EXTENDED
LDXE	LOAD XR EXTENDED
INRE	INCR AND REPL EXT
STAE	STORE AR EXTENDED
STBE	STORE BR EXTENDED
STXE	STORE XR EXTENDED
ORAE	OR TO AR EXTENDED
ADDE	ADD TO AR EXT
ERAE	EXCL OR TO AR EXT
SUBE	SUB FROM AR EXT
ANAE	ADD TO AR EXT
MULE	MULT EXTENDED
DIVE	DIVIDE EXTENDED

INDICATOR CHANGE

ROF	RESET OF
SOF	SET OF
SS1	SET SS1
SS2	SET SS2
SS3	SET SS3
RS1	RESET SS1
RS2	RESET SS2
RS3	RESET SS3
IDCN	EXECUTE _____

PRE-DEFINED EQU

AR,BR,XR, AP,OF,AZ,BZ
XZ,S1,S2,S3

MONITOR LINKAGE

CHAR FOR CHAR I/O
NUMB FOR NUMB I/O
MESG FOR MESG I/O

DOUBLE WORD ADDRESSING

JMP JUMP UNCOND
JAP JUMP IF A POS
JOF JUMP IF OF SET
JAZ JUMP IF A ZERO
JBZ JUMP IF B ZERO
JXZ JUMP IF X ZERO
JSS1 JUMP IF SS1
JSS2 JUMP IF SS2
JSS3 JUMP IF SS3
JAN JUMP IF A NEG
JNOF JUMP IF NO OF
JANZ JUMP A NOT ZERO
JBNZ JUMP B NOT ZERO
JXNZ JUMP X NOT ZERO
JNS1 JUMP IF NO SS1
JNS2 JUMP IF NO SS2
JNS3 JUMP IF NO SS3
JIF JUMP IF _____
JUL JUMP UNLESS _____
JMPM JUMP & MARK
JOFM JMPM IF OF
JAPM JMPM IF A POS
JANM JMPM IF A NEG
JAZM JMPM IF A ZERO
JBZM JMPM IF B ZERO
JXZM JMPM IF X ZERO
JS1M JMPM IF SS1
JS2M JMPM IF SS2
JS3M JMPM IF SS3
JIFM JUMP & MARK IF _____
JULM JUMP & MARK UNLESS _____
XEC EXECUTE UNCOND
XAP XEC IF A POS
XAN XEC IF A NEG
XOF XEC IF OF
XNOF XEC IF NO OF
XAZ XEC IF AR ZERO
XANZ XEC IF A NOT ZERO
XBZ XEC IF BR ZERO
XBNZ XEC IF BR NOT ZERO
XXZ XEC IF XR ZERO
XXNZ XEC IF XR NOT ZERO
XS1 XEC IF SS1
XS2 XEC IF SS2
XS3 XEC IF SS3
XNS1 XEC IF NO SS1
XNS2 XEC IF NO SS2
XNS3 XEC IF NO SS3
XIF XEC IF _____
XUL XEC UNLESS _____

MISCELLANEOUS

HLT HALT
NOP NO OP
ENTR ENTRY POINT
SYST SYSTEM COMMAND _____
SKIP SKIP NEXT LOC
TINA TRAN IND TO AR
TPAR TEST PARITY OF AR
MCRO ENABLE MICRO EXEC

REGISTER CHANGE

TZA TRAN ZERO TO AR
TZB TRAN ZERO TO BR
TZX TRAN ZERO TO ZR
TAB TRAN AR TO BR
TAX TRAN AR TO XR
TBA TRAN BR TO AR
TBX TRAN BR TO XR
TXA TRAN XR TO AR
TXB TRAN XR TO BR
IAR INCR AR
IBR INCR BR
IXR INCR XR
CPA COMP AR
CPB COMP BR
CPX COMP XR
DAR DECR AR
DBR DECR BR
DXR DECR XR

ZERO	TRAN ZERO TO _____
TRAN	TRAN _____
INCR	INCR _____
COMP	COMP _____
DECR	DECR _____

SHIFT INSTRUCTIONS

LSRA	LOG SHIFT RT AR
LSRB	LOG SHIFT RT BR
LLSR	LONG LOG SHIFT RT
LRLA	LOG ROTATE LF AR
LRLB	LOG ROTATE LF BR
LLRL	LONG LOG ROT
ASLA	ARITH SHF LF AR
ASLB	ARITH SHF LF BR
LASL	LONG ARITH SH LF
ASRA	ARITH SHF RT AR
ASRB	ARITH SHF RT BR
LASR	LONG ARITH SM RT

PSEUDO OF CODES

ORG	ORIGIN
BSS	BLOCK STARTING W/ SYMB
BES	BLOCK ENDING W/ SYMBOL
EQU	SYMBOL EQUALITY
SPAC	SPACE
EJEC	EJEC
DATA	CONSTANT DECLARATION
CALL	SUBR CALLING SEQ
MORE	INTERRUPT INPUT STREAM
END	TERMINATE PROCESS
TITL	PAGE TITLE DEFINITION

I/O INSTRUCTIONS

EXC	EXTERNAL CONT
SEN	SENSE
IME	INPUT TO MEM
INA	INPUT TO AR
INB	INPUT TO BR
CIA	CLEAR INP AR
CIB	CLEAR INP BR
OME	OUTPUT MEM
OAR	OUTPUT AR
OBR	OUTPUT BR

Computer jobs through training— A preliminary project report

by M. GRANGER MORGAN, MARY R. MIRABITO, and NORMAN J. DOWN

The University of California at San Diego
La Jolla, California

INTRODUCTION

Job training directed toward the disadvantaged population in the United States has been under way for many decades. Traditionally this training prepared people for lower entry level skilled and semi-skilled jobs such as plumbers' aides, welders, clerks, and secretarial help. Only recently, with the expanding awareness of the significant social inequalities which continue to characterize U.S. society, have large numbers of people begun to realize that job training—for just any old job—is not enough. If training is to have any appreciable impact upon the social stratification that characterizes the employment structure, efforts must be made to find high entry level jobs which are suitable for such special training projects.

Of course job training is not the ideal solution to the problem. Something that might honestly be called a "solution" will not come until the children of the disadvantaged receive a quality primary and secondary education and an equal opportunity for college level training. There are plenty of people working on reforming the school system so that such educational equality will one day be a reality. But despite some progress this task is proving exceedingly difficult. In the meantime there is a whole generation of young people who have not enjoyed the advantages of an improved school system and who are without significant job skills. The question is, can we devise job training programs which will train these young people for a career other than in low entry level jobs?

A number of workers have looked to semi-technical and business computer programming as a high entry level job area in which disadvantaged students could perhaps be trained.[1] They have viewed programming as attractive because it does not require many of the social prerequisites, such as the ability to speak dialect free English or a working familiarity with business world interpersonal relations, that are necessary for most high entry level jobs. The only real prerequisites to training someone as a semi-technical or business programmer are an ability to organize ideas in a logical way and some basic math skills.

A number of workers around the country have recently developed programmer training projects designed for the disadvantaged. These workers have come to the problem with different backgrounds and perspectives. The various projects which have evolved display a rich range of ideas, many of which might never have been tested had the central planning and coordination that is widely advocated by educationalists been applied to this development from the outset. What is now needed is a literature which describes these several efforts in detail so that future workers will not have to rediscover what has so far been learned, but can build on the basis of the experience of others. This paper will describe one of these projects, the University of California at San Diego's project, Computer Jobs Through Training.

The basic approach

Work at UCSD on programming instruction for the disadvantaged began in the summer of 1968 when we offered a course in digital logic and basic FORTRAN programming to a group of high school students who were working on the campus in summer jobs made available through the Neighborhood Youth Corps program. This initial course had no long term job training intent. It was offered strictly as enrichment, as something we thought would be a "good thing to do."

We were surprised by how well the course went, and especially by how exciting the students found the subject matter. We began quickly to see that programming and other aspects of computer science were potentially very useful topic areas for reaching and turning on students who previously had not gotten very interested in formal education.

But while it is easy to get these students "hooked" on programming, the standard teaching methods, particularly the formal lecture situation, are totally inappropriate. Programming is best taught to these students the way modern languages are now being taught. Rather than listening to lectures on the grammar of the language students learn the language by using it. Beginning on the first day, the instructor gives a bit of basic introduction and then writes a simple program. He explains it, but doesn't really expect his explanation to be fully understood. The students copy this program, punch it onto cards, which is a painful process since many have never typed, and after some brief instructions run the job themselves on a small computer. Inevitably there are errors but sooner or later the job runs and you see the first glimmerings of understanding and excitement. In the weeks that follow you build on this basic understanding, slowly enlarging on the student's repertory until he has a command of most of the language.

The physical facilities

This hands-on approach works, but only if there are adequate computing facilities available for all students to use on a continuing basis. Economically the simplest approach is to take the students to a central training facility. In the compact inner cities of our major urban centers this approach also makes good social sense. But in San Diego, while the Black community is somewhat localized, the Chicano or Mexican American community is spread all across the city in a collection of widely spaced communities. In the early portions of the course, motivation is the single most important consideration—and one good way not to motivate people is to make them sit on a bus for an hour or more every day riding to and from a class.

The prospect of establishing a number of training facilities throughout the community was financially unreasonable. In addition, we were reluctant to choose any one portion of the community for our efforts at the expense of others. The solution we chose was a mobile instructional facility housed in a forty foot trailer truck, which through careful scheduling can simultaneously support up to a half dozen courses at different locations all around the city.

A used forty foot trailer was acquired in the spring of 1969 as a gift from Safeway Foodstores, and with support from the Rosenberg Foundation of San Francisco, Montgomery Ward, and the University, the training facility was constructed in this van during the summer and fall of 1969, Figure 1. Our small project staff was considerably aided in this work by a group of Neighborhood Youth Corps students and UCSD undergraduates, largely from the Black and Chicano communities, who contributed many long hard hours of work at low pay along with much enthusiasm and a number of first rate ideas.

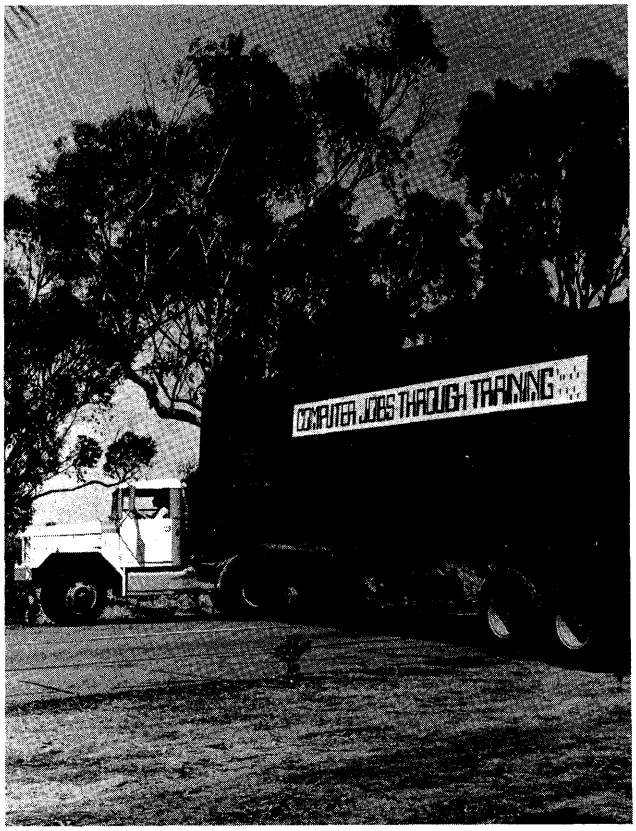


Figure 1—General exterior view of the Computer Jobs Through Training mobile computer classroom facility. Both the tractor and the forty foot trailer are used equipment which have been reconditioned by the project staff

The UCSD Computer Center is in the process of installing a large Burroughs B6500 system. Until sometime in 1971 when that system will be supporting a full complement of remote operations, our hardware in the van consists of a small computer with free standing FORTRAN capability and remote batch COBOL ability. When the B6500 system is in full operation this small machine will probably be replaced by a terminal consisting of a small card reader, a line printer and a teletype.

One important hardware requirement is the ability for students to interact with their program during execution. We stress this kind of programming in the early portion of the course because it helps significantly to motivate students and keep interest high. We find too that a drum plotter is a very useful device. Students

work up cartoons and other line drawings with considerable enthusiasm, and the systematic operations involved in pen control make for good practice in step by step logic.

The floor plan in this facility is completely flexible. This results in part from the admissions policy we have adopted. The problem of identifying potentially successful programmers even among college graduates is substantial. Making this identification for disadvantaged young adults is an almost impossible task. It is widely recognized that aptitude tests display a cultural bias. More importantly, since many of the students we hope to reach "turn off" in a testing situation, we feel that massive pre-testing, which has been the approach of some experimental programs, is not the answer.

Obviously we must require basic math and logic skills, and a level of intellectual development on the part of our students roughly equivalent to that of a high school graduate. We do not specifically require a high school degree, though most of our students have one or are in the process of getting one.

To check for math and logic skills we administer a short entrance test, which like all of our introductory material is bilingual, with English on one side of each page and Spanish on the other. But our basic approach to entrance has been—anyone who seriously claims he wants to be a programmer may enter the course. If he has not done well on the entrance exam we warn him that he will have trouble. But no one who is really serious in claiming that he wants to take the course has been excluded. Actual performance during the first weeks of the course is the real entrance test.

All this is fine, but one must be realistic. Many students will quickly discover that programming is just not "their bag" and will drop out of the course in the early weeks, others will stay with the course for some while, but despite good motivation will just not be able to do the work. These latter students we are trying to direct towards alternative more appropriate forms of training, such as the San Diego Urban League's key punch school, so that their CJTT experience will not represent a failure, but rather a first step toward something else.

Given the diminishing class size which results from this approach to admissions we have designed the van so that we can start out accommodating relatively large

Figure 2—Two interior views of the classroom van. Above, a student prepares to run her program. Below, a general view of students in the first adult evening class

numbers of students and then eventually switch over to a more spacious floor plan once the class size has fallen off. In addition, arrangements have been made to allow the van to be subdivided into smaller areas for group work with teachers aides. A laboratory set-up with normal work benches is also possible. Figure 2 shows two interior views of the van.

Curriculum considerations have dictated a number of other aspects of the physical facilities. In our work with Neighborhood Youth Corps students during the summer of 1968 we looked at a large number of the 16mm films on various aspects of computer science which are available from industry. Almost none of these films are suitable for use with our students. They are either much too technical, or much too simple minded. As a consequence a large set of 35mm slides has been developed for use with the course. These slides come in three types: course slides, which directly support the curriculum with flow charts, diagrams, drill exercises and similar materials; computer science orientation slides, which provide students with an introduction to the physical components of computer science, explain how they work, and introduce the student to a large number of typical system applications such as airline reservation systems, medical diagnostic systems, scientific systems, production control systems, and so on; and social orientation slides which consider things like how to act and what to expect in a job interview. So that these slides can be used as an integral and natural part of the course, without disrupting the flow of thought when slides are introduced, the van has been equipped with a remotely controlled projection system and variable intensity lighting. The instructor is able to use slides easily and at his convenience.

Finally, to support the hardware portion of the course which is described below, the necessary DC logic voltages, signal lines, and 110 volt lines are distributed to convenient plug panels located for student use throughout the van from power supplies and signal generators located in a small shop area in the forward portion of the van.

The instructional program

Before describing the details of the curriculum which has been developed for the CJTT course it is important to explain what kind of person, with what kind of programming skills, we are training in this project and what he will most likely do when he finishes the course. Clearly we will not be producing systems programmers. What we will produce are competent coders and programmer/trainees, who unlike the graduates of many private data processing schools will have a solid foundation in the logical aspects of programming. Our graduates will be able to take a well stated word problem, work up the necessary logic, develop the flowchart, produce the necessary code, debug the program, and make it run.

But despite our earlier observation that the only real prerequisite to success as a programmer is an ability to think logically and a command of fundamental mathematics, it is nevertheless important to realize that while many of today's successful programmers worked their way up with only a high school degree, this is becoming increasingly difficult to do. More and more a two year AA in data processing or, better still, a four year BA is becoming prerequisite to substantial progress up the data processing ladder from the lower coder and programmer positions.

Recognizing this, and understanding that we can reasonably expect to train people who at the outset are employable only in the lowest positions, we have attempted to design both our own project, and the kind of job placements we have arranged, in such a way as to maximize the possibilities of further education for our students. Hence in our course we emphasize a strong foundation in the basic logical techniques of programming rather than the sort of cook-book introduction to existing operating systems that is characteristic of many of the private data processing schools. We treat specific programming languages as secondary in importance to the fundamental ideas of program organization. But, in our choice of languages (FORTRAN and COBOL) we have been careful to select those languages which we think have the best potential for *immediate* employment, consistent with our long term objective of further education.

We begin with FORTRAN. Most of the semi-technical programming jobs in the San Diego area require FORTRAN, as do almost all of the major employers with good programs for continuing employee training and education. FORTRAN has two other important advantages. Unlike COBOL, which requires a substantial knowledge of syntax before even simple programs can be written, it is possible for students to run and understand simple FORTRAN programs on the first day of class. A second advantage is the easy use of subroutines, an aspect which we consider essential in teaching basic programming concepts.

The basic curriculum for the CJTT project was evolved during the summer of 1969 with the support of the Rosenberg Foundation and received preliminary field testing on a second group of 15 Neighborhood Youth Corps students in a 8 week summer course at UCSD, Figure 3. The organization of the course is shown in Figure 4.

The formally developed curriculum material con-

Figure 3—Neighborhood Youth Corps students from the eight week pilot course run during the summer of 1969

sists of a carefully graduated sequence of problems designed to be as familiar and interesting to the student as possible. The order of presentation of basic FORTRAN instruction in the first third of the course is:

I. Simple Math
II. Loops and Sorting Using the Computed GO TO
III. IF Statements
IV. Data Arrays and Alphameric Formats
V. Subroutines and Special Math Methods
VI. DO Loops

Because of the great significance that we attach to getting our students fully versed in all aspects of program logic, DO loops are purposely not introduced until the very end of the introductory section so that students are forced to manufacture all the looping structures they require. With a very few exceptions, all our problems are presented as word problems, both to force the development of reading skill and also to get students in the practice of working from written instructions, something most of them have done only rarely in their previous activities.

Along with this graduated difficulty in program logic goes a review of basic math concepts. This review takes place as part of the programming rather than as a separate topic since most of our students have been "turned off" by math in school, largely because they never could see any need or use for math. Having gotten the student hooked on programming, it is then possible to undertake a math review in the context of programming which students would never tolerate as a simple abstract review.

All of the early material in the course is available in English on one side of the page, and Spanish on the other. Clearly no programmer can be placed in a job in this country if he is not fluent in English. Fluency in English is a prerequisite for entrance to the course. But being fluent in English and being comfortable in English are two different things. During the 1969 Youth Corps course we found that several of our Chicano students became much more interested and did much better work when problems were available bi-lingually and when instructors showed a willingness to use Spanish. It is clear that even students who when given a choice frequently use the English version of problems nevertheless appreciate having the Spanish version

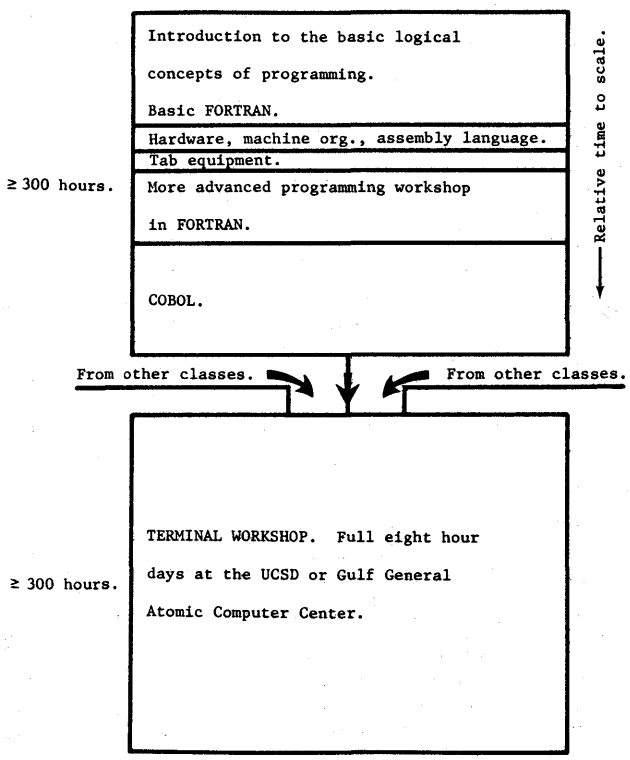


Figure 4—Diagram of course curriculum. Time runs vertically

COMPUTER JOBS THROUGH TRAINING

page 1 of 2

problem number II-6-0

Estas empleado como programador por "Bradley Imports Inc.," una compañía que hace negocio con empresas en Europa, Africa y América Latina. En este momento, la compañía tiene cuentas activas en 12 países que tienen las siguientes tarifas de cambio.

Argentina	350.000	Pesos/ $ E.U.
Brazil	2.054	Cruzeiros/ $ E.U.
Ecuador	18.180	Sucre/ $ E.U.
Francia	4.945	Francos/ $ E.U.
Alemania	4.000	Deutsche Marco/ $ E.U.
Ghana	0.980	Cedis/ $ E.U.
México	12.500	Pesos/ $ E.U.
Marruecos	5.060	Dirhan/ $ E.U.
Países Bajos	3.606	Guilders/ $ E.U.
Perú	38.700	Soles/ $ E.U.
República Arabe	2.300	Libras/ $ E.U.
Reino Unido	0.419	Libra esterlina/ $ E.U.

1. Escribe un programa que convierta precios en dólares Americanos a pesos Mexicanos.
2. Escribe un programa que convierta precios en cedis Ghaneses a dólares Americanos.
3. La compañía hace la mayor parte de su negocio con cinco países: 1) México, 2) Francia, 3) Alemania, 4) Brazil, y 5) Ghana. Las tarifas de cambio de estos países se encuentran en tarjetas en la siguiente forma

/A12.500^^^^MEXICO

Flowchart y escribe un programa que vaya a read in las tarifas de cambio desde las tarjetas (asegurate de que tengas las tarjetas en orden correcto). Después, read in un precio en dólares Americanos y un número de código que especifica el país que tu quieres, y write out el precio equivalente en la moneda del país escogido.

COMPUTER JOBS THROUGH TRAINING

page 1 of 1

problem number III-1-0

The IF statement is a FORTRAN instruction to alter the operations that a program performs depending upon the conditions which exist. Here is a simple example using the IF statement: A program computes the value of NUZ. The programmer wants to know if the value of NUZ is negative, zero, or positive. He writes:

```
      IF (NUZ) 20, 24, 28
20    WRITE (1, 21)
21    FORMAT ('NUZ IS NEGATIVE')
      GO TO 30
24    WRITE (1, 25)
25    FORMAT ('NUZ IS ZERO')
      GO TO 30
28    WRITE (1, 29)
29    FORMAT ('NUZ IS POSITIVE')
30    CONTINUE
```

If NUZ is negative the program jumps to 20, if it is zero it jumps to 24, if it is positive it jumps to 28. In place of NUZ you can put some expression like:

IF (NUZ*2-NSUM) 212, 222, 232

In this case if NUZ*2-NSUM is negative, the program goes to 212, if it is zero it goes to 222, and if it is positive it goes to 232.

Some friends of yours who are publishing a newspaper for the Black and Chicano communities have learned that you are a programmer and have asked you to help them automate the billings and records for their advertisements.

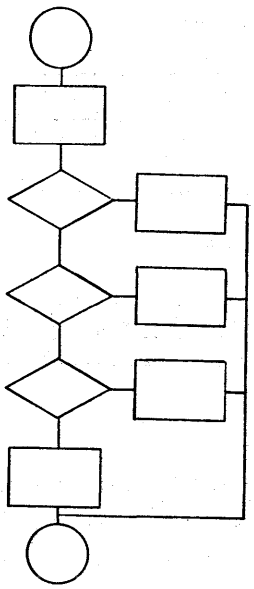
1. Advertisements are sold by the column inch. The rate is $2.50 per column inch for the first 5 inches, $2.00 per column inch for all space beyond 5 inches. Flowchart and write a program which will read in the size of an advertisement in column inches from the keyboard and type out the price. Write the program so that it will continue to loop back and read in new numbers for as long as you wish.

COMPUTER JOBS THROUGH TRAINING

page 1 of 1

problem number III-4-0

You have saved up enough money to treat yourself to dinner out. If you have $5.00 or more you can go to Sister Peewee's; if you have $2.00 or more, you can go to Huffman's Barbeque for chitlin's and greens; if you have $1.00 or more, you can go to Mac Donald's for a hamburger; if you have less than a dollar, you have to eat at home. Fill in the flowchart below and write a program to do the above.



COMPUTER JOBS THROUGH TRAINING

page 1 of 1

problem number III-8-0

You have been hired to help automate the sales records in a discount store. The sales manager does not understand too much about computers and programming and so as a simple example you are going to write him a program that will make the computer work like a very fancy cash register.

You want to be able to read in the price of a number of items, compute the tax on the taxable items, and then add things all up to figure out how much the customer owes you. You also want to figure out how many green stamps to give him. You want to do this over and over again for each customer.

In addition, you want to keep track of the total tax collected during the day, and the total cash sales in each category of item:

Code	Item	Tax
1	foods	none
2	liquor	15%
3	stationery	5%
4	hardware	5%
5	clothing	5%

Here is a step by step description of how your program should work:

1. Type in the code and price of each item.
2. When you have entered all the items make the program write out the total purchase price, the tax, and the number of green stamps (one stamp for each 10¢ paid for purchases less than $10.00, double that if you buy more than $10.00) for this sale.
3. The program goes back and is ready to work on the next customer's purchases.
4. At the end of the day you make the program write out:
   a. The total of all purchases made that day in each of the 5 categories as well as the total sales for all categories.
   b. The total of all taxes paid that day.

This problem is a little tricky. Be sure to design a correct flowchart before you try to code the program.

Figure 5—Typical problems from the early stages of the course. All early problems are available in English on one side of the page and Spanish on the other

available and feel more relaxed in the course as a consequence.

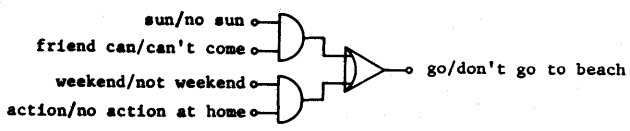
A few examples of some of the early problems for the CJTT curriculum are given in Figure 5. Along with these problems, brief non-credit drill sheets are used massively in the early portions of the course. Unfortunately it is not possible in the written version of this paper to give a proper impression of the 35mm slides which have been developed to accompany the course. This will be attempted in the oral version of the paper.

A knowledge of digital logic is hardly prerequisite to most programming jobs, though it does make some of the fine points of programming more intelligible. But hardware can serve as an excellent motivational tool. We explain to our students that computers are complicated in just the way that a house is complicated. The bricks, nails, and boards which make up a house are conceptually simple. It is only when thousands of them are combined in a building that you end up with something that is complicated. In much the same way, AND-gates, OR-gates and flip-flops are logically simple devices. It is only when thousands are wired together to make a computer that you end up with a complicated system.

We introduce hardware with as little talk about actual circuit elements and abstract symbolism as possible. The AND gate is introduced with a demonstration which solves the problem "if it's sunny AND your friend can come, you can go to the beach." The truth table is worked out in terms of English:

is sun out?	can friend come?	do you go?
no	no	no
no	yes	no
yes	no	no
yes	yes	yes

Once such a truth table is introduced for several simple problems, the jump to more abstract levels of binary notation is not too difficult. Likewise, more complex circuit configurations such as two AND gates with their outputs ORed together are used to solve day to day problems. For example the circuit:



solves the problem, "it it's sunny AND your friend can come OR if it's the weekend AND there's no action at home, you go to the beach".

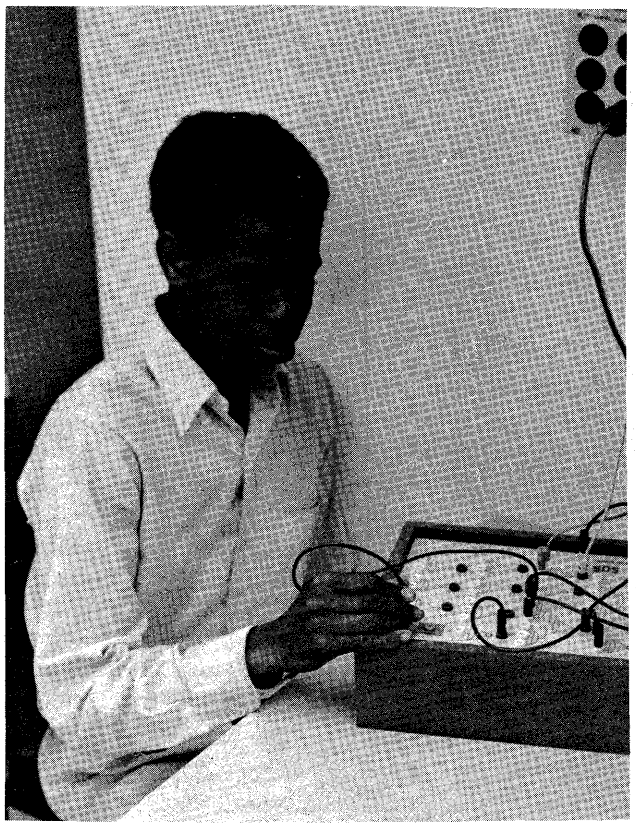


Figure 6—A student wires a simple circuit with one of the digital logic plug boards

In keeping with the hands-on philosophy of the project, individual logic plug boards which will allow each student to work up his own circuits have been developed as shown in Figure 6. The development of this equipment has been made possible with a grant of T-series logic from Xerox Data Systems.

To conclude the hardware unit there is a final class project. In an introductory class for engineers, the class might build tape controllers, parallel to serial converters, or similar devices. Many of our students find such examples rather unexciting. Instead, we have chosen a problem which is technically just as demanding, but which is substantially less abstract. A model N-guage rapid transit system for the city of San Diego has been built and outfitted with appropriate micro-switches which provide information on the trains at all times. The set up is shown in Figure 7. The class is asked as a group to develop the control logic necessary to automate the system.

With this hardware background, an introduction to machine organization and the fundamental ideas of as-

Figure 7—Student staff member, Belton Flournoy, works on the development of the model rapid transit system

sembly language programming follow in a natural way. We teach no specific assembly languages in this course, but we do try to give our students a good idea of how an assembly language works so that if later he must learn one he will know what is going on.

The tab equipment unit is designed to give students a very brief introduction—it is not designed to train experts. We quickly outline the use of the program drum on the keypunch and then briefly introduce and use the sorter and accounting machine. The project has acquired 62 surplus type 910 control boards so that each student is able to wire one simple listing problem.

As indicated in Figure 4 the middle third of the formal portion of the course is devoted to more advanced programming concepts such as file organization and maintenance, extensive use of subroutines, and similar techniques which are required for simple system work. The final third of the formal curriculum introduces COBOL. This portion of the course involves few new logical ideas. Indeed, students do many of the same problems that they have already worked in FORTRAN, so that they can develop a feel for the comparative strengths of the two languages.

Well before the end of the course it is clear which students are doing well enough for job placement. When these students complete the course they enter a Terminal Workshop of intensive full-time training for a period of about seven weeks which prepares them for their job. If the potential employer has indicated that he requires specific skills, such as COBOL proficiency or extensive magnetic tape experience, these are emphasized. The Terminal Workshop is run at the UCSD Computer Center and at nearby Gulf General Atomic. Though we have serious financial difficulties, it is our intention that all students who require financial assistance during this full time terminal workshop period will receive a stipend.

Job placement

It is not sufficient in a project such as this one to count simple placement in a programming job as success. What counts of course is the number of people who continue to work in the field long after initial placement. In order to minimize difficulties in the early months after placement our students will be followed carefully once they are out on the job by a Placement Aide who will try to detect difficulties, either of a social or technical nature, well before they become serious, and take the necessary corrective action.

Our first course for young adults got under way in February 1970 on a nighttime basis. We made arrangements with local employers for placement of the modest number of graduates that we expected from this first course before the course began. However it is not reasonable to expect to get massive commitments for placement for many students before a project such as ours has produced its first graduates. Programmers are not plumbers' aides. While industry will commit itself to hire large numbers of low entry level people from training programs, any reasonable employer will insist on seeing the quality of the graduate before he commits himself to hiring someone like a programmer.

To sell the project to employers we are using the mobile aspect of our facility—setting up in a potential employer's parking lot with the van and a number of our students and asking the employer to come have a look at what we are doing. We are also seeking employers' commitments to hire our graduates through local organizations such as the Urban Coalition. At the time of this writing the first full adult class has not yet been graduated. Further details on the placement aspects of the project will be provided in the oral version of the paper.

Other aspects of the CJTT program

While job training is the principal objective of the CJTT project, the two Youth Corps classes that we ran during the summers of 1968 and 1969 were valuable in their own right as courses which stimulated and motivated disadvantaged high school students toward further education and careers in computer science or in other technical fields. We have been operating with a small staff and a very small budget, and this together with the fact that our primary objective has been organizing the job training project has prevented a careful systematic follow-up on all of the NYC students. However, judging from those students with whom we have maintained contact, the courses have had a significant impact.

During the summer of 1970 we have made more formal arrangements to continue this motivational type of instruction for high school students. With financial support from the San Diego Unified Schools we are running three special motivational classes for credit as part of the city school's summer session. These classes, two for high school students, one for entering seventh graders, are being conducted in schools with very high enrollments of disadvantaged youths. The instruction is being done by four UCSD Black and Chicano undergraduates who are majoring in computer science. Progress in these classes has been excellent and we expect to expand our in-school activities.

CONCLUSION

We have restricted ourselves in this paper to a straightforward description of the CJTT project but it would not be proper to conclude without giving some indication of the very serious financial difficulties that we have encountered. On the local UCSD campus we have received strong moral and financial support from Professor Kenneth L. Bowles who directs the Computer Center and Professor Henry G. Booker, Chairman of the Department of Applied Physics and Information Science. Outside of this support, which had totaled just under $40,000 by July of 1970, we have pieced together an additional $40,000 from dozens of separate sources, most of which are listed in the acknowledgments.

Much of this $80,000 of support which had been organized as of July 1970 was in-kind assistance. With this support the CJTT project has accomplished what by normal University operating methods would have cost slightly more than $150,000. This savings did not come easily. It results from substituting labor for capital. It was accomplished at times by turning highly qualified programmers into painters and carpenters; by scrounging used electrical conduit from old buildings about to be razed; scrounging lumber from construction site foremen; putting Ph.D.s to work doing carpentry and digging telephone pole holes.

Despite more than a year of strenuous salesmanship and much proposal writing, the project had still not received any State or Federal anti-poverty monies as of July 1970. Our evaluation of the funding situation is that this experience is not unique, that others planning similar programs can anticipate similar very serious funding difficulties unless they can find strong sources of local or private financial support.

Our estimated costs are roughly $150,000 per year or just over $3,000 per student *placed.* Something like 80 percent of this cost is for salaries. No student stipend costs are included in these figures. While we began this work with great optimism, our experience to date has led us to seriously question whether long term funding of this magnitude can be organized. There is much talk in the country about how important it is to do this sort of thing but not very much money to do it. Future workers would do well to explore the financial climate with great care before launching new projects.

REFERENCES

1 While a number of workers have undertaken projects in this field, the literature is still very spotty. A review of some of these projects is available in:

D B MAYER
The involved generation—Computing people and the disadvantaged
AAPS Proceedings of the Fall Joint Computer Conference 1969 p 679

In addition to the work reviewed in Mayer's paper we are aware of work undertaken by:

R BELLMAN J BLOOD C FORD-LIVENE
Project Soul: Computer training for high school students from disadvantaged areas
University of Southern California Technical Report UCSEE-375 November 1969

T I BARTHA
Computer programmer training program
Report of the Computer Education and Research Center Pratt Institute Brooklyn New York 11205

L H HARRIS
of Shell Development Corporation (P. O. Box 481) in Houston, Texas, has run a training program for a number of years.

ACKNOWLEDGMENTS

We wish to acknowledge the invaluable assistance of Kenneth Bowles, Henry Booker, Jack Douglass, Frank

Saiz, Roy Cazares, Ken Hicks, Oneeta Alexander and Bob Sadler of the UCSD staff, along with the assistance of many UCSD undergraduates and Neighborhood Youth Corps students of whom Belton Flournoy, Susan Halfon, Calvin Manson, Beverly Andrews, and Boyd Pearson deserve special mention.

In addition, we gratefully acknowledge the financial assistance of the University of California, The Rosenberg Foundation, Gulf General Atomic, Safeway Stores, Xerox Data Systems, Montgomery Ward, Bekins Van Lines, Pacific Gas and Electric, Pacific Telephone, and other local supporters.

Technical and human engineering problems in connecting terminals to a time-sharing system

J. F. OSSANNA

Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

and

J. H. SALTZER

Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

Today, an increasing number of computer systems are used interactively by their user communities. Interactive use of computers, involving more prolonged man-machine contact than non-interactive use, requires a well human engineered user-system interface. The interactive user's performance—his rate of doing work and his ability and desire to utilize system capability—is a sensitive function of the success of this human engineering. In turn, the computer system's effectiveness depends on achieving a satisfactory level of user performance with reasonable efficiency.

This paper will be concerned with the human engineering of connecting typewriter-like terminals to general purpose time-sharing systems. Examples of such systems are Digital Equipment's 10/50 system for the PDP–10, IBM's Time-Sharing System for the 360/67, the Dartmouth Time-Sharing System, and the Multics system at MIT. Such systems are used by a wide range of users doing many kinds of work. Typewriter-like terminals constitute the majority of general-purpose remote terminals in use today; examples are the Model 37 teletypewriter[1] and the IBM Model 2741.[2] Although more complex terminals, such as those providing true graphical capability, are not specifically treated, many of the factors to be discussed apply to them. The special behavior and needs of specialized systems are not treated, but some of the ideas presented will apply in individual cases.

Value judgments about human engineering factors always involve a degree of individual taste which in turn depends in part on individual experience. Many of the ideas expressed here are the outgrowth of experience obtained during the growth and use of Project MAC's CTSS system[3,4] and during the development of Multics.[5]

Good user performance becomes possible when the user can easily and rapidly do what he wants to do. Consequently, many of the human engineering factors to be discussed relate to the user's ability to provide input as rapidly as desired, to control output, and to avoid unnecessary interaction.

First, we will discuss input/output strategies, since they broadly affect most of the other areas to be covered. Then we will discuss in turn, terminal features, the terminal control hardware, and the terminal control software—working from the user into the system. Finally, we will briefly mention character sets and character stream processing.

INPUT/OUTPUT STRATEGIES

The user's input consists of system commands, requests to programs, data, answers, etc. From the user's point of view, input can be divided into components according to whether or not it is expected that the component will cause output to occur. Some input is expected to cause output to occur—for example, a command to list a file directory. Other input may be expected to cause output only conditionally; for example, a command to rename a file may output an error comment only if the named file doesn't exist. Still other input may be expected to cause no output—for example, continuous text input into an editor.

From the system's point of view, the user's input can be considered a character stream containing certain characters indicating that action should be taken. In the common line-by-line input case, a return or new-line character is the only action character. In general, there may be a number of action characters. In certain applications treating all characters as action characters may be appropriate. The user ordinarily should know what action characters are currently in effect, since typing one of them initiates execution, which may in turn cause output.

The human engineering problem in collecting a user's input arises primarily because the user frequently knows much of what his input is to be well in advance. He may know the next several commands or the next several editing requests he wishes to input. In general, the components of this known-in-advance input can fall into all three output relationship classifications. Although the user often knows when to expect output, the system cannot.

The user should not be unnecessarily prevented from providing such input as fast as he can think of it and can type it. By collecting input asynchronously rather than synchronously with respect to the system's utilization of the input, the user and the computer can work asynchronously and in parallel rather than synchronously and serially.

There are four mechanisms that can individually or collectively facilitate providing input.

First, input can be collected whenever there is no output occurring. If the operation is full-duplex,* it is even possible to collect input while output is occurring. The typing of action characters should trigger program execution but not inhibit further input. Such asynchronous collection of input is usually referred to as read-ahead or type-ahead. A number of present day systems[4,5] provide a read-ahead strategy.

Read-ahead permits overlap of input with both system response time and program execution. Also, it permits programs such as text editors to gather text input continuously. Because erroneous input may be encountered, programs must be able to produce conditional output and also discard existing read-ahead to prevent compounding of errors.

A second mechanism is to allow more than one independent input component between action characters. For example, a system using new-line as an action character should permit more than one command on a line. Editors in such a system should permit more than one editor request per line. This outlook should pervade every level of programming.

Third, commands and other programs should be designed to avoid unnecessary interaction. One aid in doing this is to allow the typing of arguments to a command on the same line as the name of the command. For example, typing "edit zilch" is preferable to typing only "edit" and later answering the question, "Filename"? Default parameter values can frequently be assumed in the absence of typed arguments. Permitting both multiple commands and arguments enables various schemes for inputting factored command and argument sequences.[5]

Fourth, it is convenient if the user can create a file containing potential input and subsequently cause the system to take input from this file.

The use of these mechanisms can also improve system efficiency by reducing the number of separate program executions, since the program may find more input and be able to do more work during each execution.

The user should have reasonable control over his output. For example, whenever a stream of unwanted output occurs, it should be possible to stop it without undesirable side effects, such as losing too much of the results of immediately previous interactions. An interrupt mechanism, such as that detailed later, can be used to stop the output, cause execution to halt, and discard any read-ahead. If the system allows an interrupted program to catch the user's interrupt signal, a program desiring an extra degree of sophistication can be designed to recover from various conditions such as unintended execution loops or unwanted output due to unwise input. User control over output code conversion is desirable and will be discussed later. The ability for the user to direct program output to destination(s) other than his terminal is quite useful. For example, the output from a program which generates a large volume of output can usefully be directed to a file for later printing.

REMOTE TERMINAL CHARACTERISTICS

An excellent treatment of features desirable in typewriter-like terminals can be found in Reference 6. We will treat here certain important terminal design features which strongly affect the system designer's ability to human engineer the system-user interface.

A typewriter may be viewed as a collection of data sources—the keyboard, the receive-data lead of the modem or data set, and possibly a paper-tape reader—and data sinks—the printer, a control detector, the

* In full-duplex operation, transmission can occur independently in both directions. This requires independent keyboard and printer operation at the terminal, as well as independent input and output at the computer. The modems (or data sets) typically used to connect the kind of typewriter being discussed to the telephone line ordinarily provide full-duplex transmission.

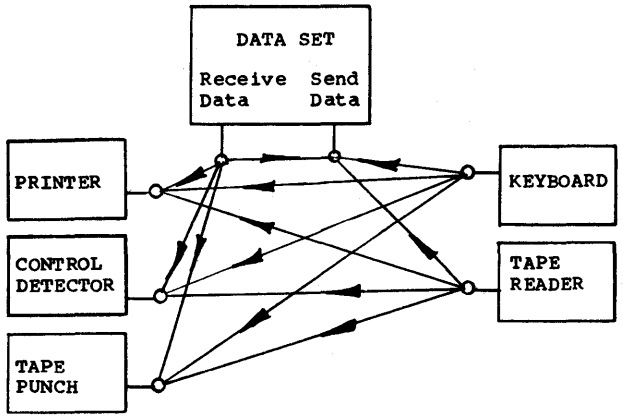


Figure (1a)—Typewriter data sources and sinks and possible interconnections

send-data lead of the data set, and possibly a paper-tape punch. Figure (1a) shows such a collection and possible interconnections. Flexible user and/or system control over these source-sink interconnections permits implementation of various input/output strategies.

As a specific example, Figure (1b) shows the interconnection control of a Model 37KSR teletypewriter. Control of the switches occurs by detection of control character sequences by the control detector associated with the printer. The interrupt detector and generator are discussed below. When the keyboard-to-printer connection is closed the terminal is in half-duplex mode and local printing of keyboarded data occurs. When this connection is open the terminal is in full-duplex mode, and the relationship between keyboarded data and printed copy is under control of the computer system. One common use of the full-duplex mode is to collect passwords without printing them. The full-duplex mode allows the printed characters to be simple mappings, or even arbitrarily elaborate functions, of the keyboarded characters. The ability to lock and unlock the keyboard allows the system to constrain the user to type only when input is able to be collected by the system.

The program interrupt ability previously mentioned can be achieved by full-duplex operation of both the terminal and computer, which permits an interrupt-implying character to be typed at any time. Another method, which does not require full-duplex operation, is the "line-break" technique,* where an always generatable unique signal can be transmitted. In addition, the ability of the terminal to respond to a break or interrupt signal from the computer regardless of its state provides a method of restoring the terminal to a desired state—typically ready to receive control or text information. As an example, the Model 37 responds to a break by locking the keyboard; the Model 37 break generator and detector are shown in Figure (1b).

The system should be able to maintain knowledge of and control over the states of the terminal. In particular, the system should be able to force the terminal into a state where the system can print on the terminal without user interference. As many terminal actions as possible—for example, those causing carriage and paper motion, color shift, source-sink interconnections—should be initiated by character sequences whether terminal or computer generated. This implies that the character set used should be sufficiently rich in control characters.

The terminal should not inherently hinder implementation of a read-ahead strategy. For example, the keyboard should not lock automatically after the typing of what the terminal assumes is an action character, such as at the end of a line; such terminal behavior is a violation of a general rule that the terminal shouldn't try to "outguess the software."[6] When a system controls input by keyboard locking the user should know when the keyboard is usable without having to test it. For example, the Model 37 lights a "proceed" lamp when the keyboard is unlocked. Using a "new-line" function (combined carriage-return and line-feed) is simpler for both man and machine than requiring both functions for starting a new line; the American National Standard X3.4–1968[7] permits the line-feed code to carry the new-line meaning. The terminal should have adequate functions for speeding up both input and output. Horizontal tabs are essential, form feed and vertical tabs are useful. They are the most useful when the user can easily set the stops himself

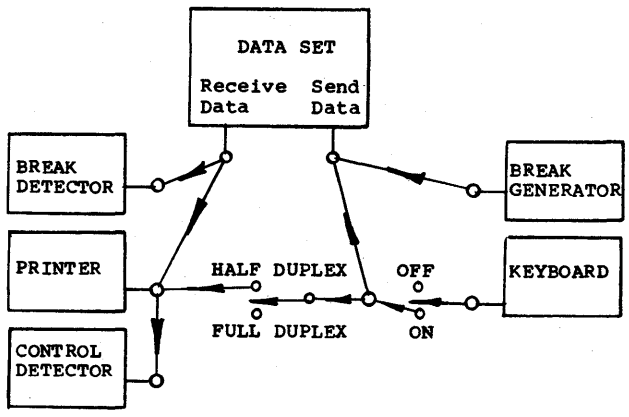


Figure (1b)—Model 37KSR teletypewriter interconnections

* The "line-break" or "break" signal usually consists of approximately 200 milliseconds of "space" ("0" bits). This is distinguishable from ordinary characters and is easily detected independently without the necessity of being able to receive characters.

using control character sequences; this is possible in some present day terminals.[1,8]

When a terminal has reached the system via a switched telephone network, the system may not *a priori* know anything about the calling terminal, and it can be useful if the terminal can send an identification sequence to the system upon demand. This sequence can be used to uniquely identify the terminal, to determine the terminal type, and to indicate terminal options. The Model 37 answer-back scheme is an example of a more than adequate identification. The economic advantage of having different terminal types statistically share computer ports is a strong motivation for the system to be able to experimentally determine the terminal type. It is necessary only that each terminal to be supported be able to respond to a transmission from the system and that either the transmission or the response be unique. Multics currently supports four types of terminals and determines which type by performing an experiment involving identification responses.

The Model 37 teletypewriter and the General Electric TermiNet–300[8] (Registered Trade Mark of the General Electric Company) provide nearly all of the above-mentioned features. Consider the standard version of IBM's Model 2741[2] terminal, which is widely used as a time-sharing terminal. This terminal can only be used in the half-duplex mode, so there is no way to inhibit direct local copy or to exploit full-duplex operation. The terminal cannot be interrupted by the system while the keyboard is unlocked; thus the system can't force the terminal to accept output while the user is able to type. This property makes read-ahead a somewhat dangerous strategy, since conditional output is impossible while the user is able to type. The keyboard locks as a result of typing "return" (new-line), and requires the system to respond and unlock the keyboard before the user can proceed. Even with instant system response, the delay before typing can continue (caused by the transmission of control characters) is noticeable, so that any read-ahead strategy is degraded. No keyboard-unlocked indication is provided for the user. Adding an identification mechanism, enabling interrupt to be always generatable and receivable, adding a local-copy suppress mode, and eliminating the automatic keyboard lock, are possible modifications; unfortunately, as is characteristic of post-initial design changes, they add significant cost.

COMPUTER SYSTEM TERMINAL CONTROL HARDWARE

The terminal control hardware used today broadly falls into two categories. One is the peripheral stored-program controller; the other is the hard-wired controller operated directly by the main computer. The major difference between these in practice is in the way the control software is modularized. The various functions to be performed by the terminal control hardware and software together can be divided between them almost arbitrarily. The decisions made when allocating logic between a main machine control program and a hard-wired or stored-program controller involve a variety of economic and other management considerations; it is not our intention here to discuss relative virtues of hard-wired and stored-program controllers. In either case, if the controller provides a primitive but complete set of functions, the terminal control program in the main computer can assume primary logistic control over the terminals. Such a controller is assumed in the following discussion, which describes suitable controller functions.

Because it may be safely assumed that new and better terminals will continue to be introduced, the terminal controller should be flexible enough to permit operating these new terminals with minimum modification. Specifically, parameters such as the number of bits per character, the character parity, and the bit rate should be program controllable or at least field modifiable. At any given time, there are usually several terminal types worth supporting. The controller must be able to handle the corresponding variety of line control

```
READ START SEQUENCE
1. SET TRANSMIT MODE.
2. TRANSMIT LITERAL "EOT" CHARACTER.
3. SET READ MODE.
4. READ ONE CHARACTER (SWALLOW "EOA" CHARACTER).
5. SET ACTION CHARACTER LIST TO (JUST) NEW-LINE.
6. TRANSFER TO READ SEQUENCE.

READ SEQUENCE
7. READ INTO BUFFER 1.
8. READ INTO BUFFER 2.
9. TRANSFER TO KEYBOARD-LOCKING SEQUENCE.

KEYBOARD-LOCKING SEQUENCE
10. SET TRANSMIT MODE.
11. TRANSMIT LITERAL "BREAK" SIGNAL.
12. STOP.
```

Figure 2—Command list to read the keyboard of an IBM 2741

requirements without undue programming effort and without undue main processor intervention; this implies suitable controller command chaining, which is described later.

When terminals reach the system via a switched telephone network, the system needs to be fully aware of call-ins, disconnects, and line failures. Thus the controller should make available to the software all status available from the modem or data set, and allow the system to enable interrupts for status changes. Similarly, the controller should allow the system to set all the control leads of the data set, so the system can control data set answering, make lines in hunt groups appear busy, and initiate disconnects. Such control allows the system to disable improperly working lines and to exercise system load control.

Certain terminal functions (tabs, form-feed, new-line, etc.), require that a delay sufficient for completion follow its initiation. If this delay is provided by the inclusion of "fill" characters (causing no terminal action), only the needed number should be transmitted. Experience suggests that accurate delay calculation, providing only the actual delay necessary, speeds up output and gives the system a smoother and speedier image.* Preferably, delays should be calculated to the nearest bit time rather than to the nearest character time.

An important controller feature is the ability to act on a list of queued "commands" from the control software. The command repertoire should include commands to set controller and data set modes, obtain controller and data set status, transmit from a buffer, read into a buffer, transmit a literal bit string, and transfer to another command. The tandem execution of two or more read *or* write commands is usually called "data chaining." The tandem execution of a list of mixed read and write commands is usually called "command chaining." A transfer command allows the list to be conveniently built of sublists and dynamically threaded together. The ability to transmit literal bit strings allows the transmission of delays (all 1s), breaks (all 0s), and canned control character sequences.

The ability to data chain while reading is an important help in allowing continuous input, because it allows a more relaxed software response to an exhausted buffer. To simplify buffer management, the controller should be able to interrupt on an action character but continue reading sequentially into the same buffer; an interrupt should also occur on data-chaining to alert the software of an exhausted buffer. It is useful if the action character(s) detected can be dynamically set by the software. If the action character(s) can be associated with each individual read command and the action to be taken individually specified, the ability to chain a list of mixed read and write commands permits handling a variety of terminal types and the design of good read-ahead strategies. The detection of a received "break" signal should halt the controller and cause an interrupt.

Figure 2 shows a hypothetical command list similar to lists implemented in Multics. The list illustrates reading the keyboard of an IBM 2741 (modified to accept break signals), and employs several sublists. After an interrupt from the controller indicating the exhaustion of buffer one, the control software would ordinarily replace the transfer in step 9 with a transfer to another read sequence. The keyboard-locking sequence stops input should the system fail to obtain another buffer prior to exhaustion of buffer two.

General Electric's General Input/Output Controller (GIOC) used with the GE 645 system (on which Multics is implemented) is an example of a communication controller that provides most of the above-mentioned controller functions. Reference 9 describes the design of the GIOC.

TERMINAL CONTROL SOFTWARE

The following discussion will be concerned with terminal control software in a main computer using a flexible terminal controller. We will discuss the need for flexibility of design and operation, the implementation of input/output strategies, some of the responsibilities to other system software, and a little about the interface to user programs.

The major areas where flexibility is important in terminal control software are the ability to operate various terminal types, and the ability to adapt to the variable behavior and needs of users.

The advantages of being able to operate a variety of terminals are: (1) freedom from dependence on one terminal supplier; (2) ability to take advantage of newer terminals; (3) user access to terminal features not all found on one terminal; and (4) satisfaction of individual user needs and preferences. The ability to operate various terminals and to easily extend operation to new terminals requires a flexible and convenient method for converting between internal system character codes and physical device codes, and for handling the different kinds of terminal control.

If the terminal control software is designed to be driven by a collection of tables, it should be possible to

* This effect was noticed during the early development and use of Project MAC's CTSS. Subsequently on both CTSS and Multics, users quickly noticed longer-than-needed delays on new terminals or due to untuned new software.

embed device differences and perhaps user options in the tables rather than in the harder-to-change program. Flexibility and extensibility can be achieved by sufficient ingenuity in choosing what information is to be relegated to tables. The generality required in such tables depends on the range of terminals to be controlled. Control driving tables can include the following:

1. Input and output code conversion tables.
2. Device parameter tables.
3. Tables of controller command sequences for identifying and operating the various devices.

The system-device code mappings contained in the code conversion tables would include suitable "escape" character sequences for handling system-defined characters not present on some terminals.* Also, additional tables could be provided for alternative conversion modes on the same terminal,** and to accommodate, for example, the user who wants to use a non-standard print element on an IBM Model 2741 or an extended-character type-box on a Model 37 teletypewriter.

The device parameter table would contain such information as default action characters, default output line overflow length, default code conversion table name, carriage return speed for delay calculations, character parity, etc.

The operating command sequence information includes sequences for initiating a write, writing, terminating a write, initiating a read, etc. The identification command sequences are the ones used for terminal type determination; often the terminal identification code is obtained as a by-product of type determination.

If the hardware controller can interrupt on an action character and otherwise continue, then only a small fixed buffer space need be associated with each active terminal—that needed for current physical input/output by the controller. All other buffer space can be pooled and assigned to individual terminals on demand. A simple read-ahead strategy can be implemented by copying input characters from physical collection buffers at action character interrupt time into a linked list of input buffers obtained dynamically from the buffer pool. When the user program requests input, the input is taken from the user's input buffer list. Similar buffer schemes have been long used for handling devices such as magnetic tapes, but are not often seen used for terminal control.

Similarly, a user program's output can be copied into a dynamically grown buffer list. Physical output occurs by refilling from the output list the physical buffer associated with each terminal every time its contents have been output. With half-duplex operation, emptying the output list should reinstate read-ahead. Letting a user program emit substantial output before suspending its execution (referred to as permitting write-behind) usually improves system efficiency by reducing the number of separate program executions. Physical output should be initiated as soon as there is any, and not delayed perhaps waiting for a buffer to fill. Aside from distorting the sense of program progress, such output delay can make program debugging very difficult. For example, debugging often involves inserting additional output statements in various program branches to obtain flow information. It is misleading not to see this flow information prior to a program entering an unintended loop, because of inappropriate output delay.

Of course, reasonable limits must be put on how much read-ahead and write-behind is permitted, lest a single user or his program seize all available buffers. Adequate total buffer space should exist to cover reasonable fluctuations in total demand. Algorithms to limit the buffer space that can be claimed by one user should be generous when conditions permit to avoid losing the advantages of read-ahead and write-behind. During peaks in total demand that tax the available space, these algorithms should be gracefully restrictive. Some successful limiting algorithms[4] involve allowing each user to accumulate a fixed fraction of either the total buffer space set aside for all such terminals, or of the current remaining space. Because the average output character rate is typically ten times the average input character rate,[11] the limiting algorithms must prevent write-behind demands from completely depleting the available buffer space, so that some space is kept available for collecting input.

The terminal control software is responsible for blocking further execution of the user's program when it requests input and none is available, and whenever it exceeds the write-behind limit. In the waiting-for-input case, the program must be restarted when an action character is detected. In the waiting-for-output case, the program should be restarted when the back-logged output has dropped to an amount whose physical output time will approximately correspond to the restart delay (system response), so that the physical output can occur continuously.

Another responsibility of the control software is to detect and report disconnects and the user's interrupt (break) signals. Disconnects should be reported to the

* For example, the sequence "¢<" could be used to represent a "[" on an IBM 2741.[10]

** If the default mode utilizes escape sequences for missing characters, an alternative mode could print blanks for such characters to permit inking them in.

system module responsible for reclaiming the communication line and making it available to other users. The interrupt should be reported to the system module responsible for suspending the execution of the user's program, pending input from the user indicating the reason for the interrupt.

The subject of the interface between the terminal control software and a user program is too large to be covered thoroughly in this paper. The flexibility built into the control software should be available to the user program. It should be possible, for example, to request a different code conversion table, specify a new line-overflow length, discard existing read-ahead input, turn off and on the terminal's local copy, disconnect the terminal (if it is on a phone line), request the terminal's identification code, etc. A particularly bad interface example occurs in some systems in use today, in which it is not possible to simply read from the terminal. The user program can only issue a write-read sequence. Output is forced to occur between each line of input. Consequently, the user program is scheduled and executed to perform this obligatory output. The overall effect is to degrade system efficiency as well as seriously slow down the user at the terminal.

The typewriter control software in the Multics system is almost completely driven by tables organized along the lines described above. A single control program currently operates the Model 37 teletypewriter, IBM Models 1050 and 2741, and the General Electric TermiNet–300. Full read-ahead and write-behind are implemented with a maximum limit which corresponds to about 700 characters for both the read and write buffer lists. A buffer pool of 250 14-character block has proven adequate in a 35 user system. In addition each active typewriter has physical read and write buffers of about 100 characters each. After a program exceeds the write-behind limit and is blocked from execution, it is restarted when the write-behind has dropped to about 60 characters.

CHARACTER SET AND CHARACTER STREAM CONSIDERATIONS

The choice of a suitable character set and suitable processing of the input and output character streams are extremely important human engineering issues which can affect the user's view of the system as much as any of the factors already discussed. An earlier paper[10] contains a detailed treatment of these issues; it includes discussion of character set choice, input and output code conversion, input text canonicalization, and input line editing.

CONCLUSIONS

The total effectiveness of a time-sharing system and its user community depends a great deal on the human engineering of the system-user interface seen by the user from the vantage point of his terminal. We have concentrated on the factors affecting the user's ability to provide input at the rate he wishes and to control output. Suitable input/output strategies can allow the user to work in parallel with the computer. We have maintained that a coordinated design of the terminal, the terminal control hardware, the terminal control software, the system's command stream interpreter, the commands, and other programs, are all necessary to achieve the desired goal.

Many of the individual factors discussed, of course, have been recognized as important in the design of various systems. It is rare, however, to find a sufficient set of these factors implemented to a satisfactory extent. One reason for this is that the system designer is often faced with using previously designed terminals and terminal control hardware, and even previously written software. Another reason is that even with experience using a variety of interactive systems it can be difficult to assess the sensitivity of the human interface to differences in design. Too often, this lack of complete design control together with insufficient experience results in a system design lacking some important features.

ACKNOWLEDGMENTS

Many of the techniques described here were developed over a several year time span by the builders of the 7094 Compatible Time-Sharing System at MIT Project MAC, and by the implementors of Multics, a cooperative research effort by the General Electric Company, the Bell Telephone Laboratories, Inc., and the Massachusetts Institute of Technology. Among those contributing to the understanding of how to effectively implement typewriter input/output were F. J. Corbató, R. C. Daley, S. D. Dunten, E. L. Glaser, R. G. Mills, D. M. Ritchie, and K. L. Thompson.

Work reported here was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction is permitted for any purpose of the United States Government.

REFERENCES

1 *Model 37 teletypewriter stations for DATA-PHONE* (Registered Trade Mark of the Bell System) *service*

Bell System Data Communications Technical Reference American Telephone and Telegraph Company September 1968

2 *IBM 2741 communications terminal*
IBM Form A24-3415-2 IBM Corporation New York

3 P A CRISMAN
The compatible time-sharing system: A programmers' guide
Second Edition MIT Computation Center The MIT Press Cambridge Massachusetts 1965

4 J H SALTZER
CTSS technical notes
MAC Technical Report No 16 Project MAC Massachusetts Institute of Technology March 15 1965

5 *The multiplexed information and computing service: Programmers' manual*
MIT Project MAC Cambridge Mass 1970 (to be published)

6 T A DOLOTTA
Functional specifications for typewriter-like time sharing terminals
Computing Surveys Vol 2 No 1 March 1970 pp 5-31

7 *American National Standard X3.4-1968*
American National Standards Institute Oct 1968

8 *Programmers' manual for the General Electric TermiNet-300 printer*
No. GEK-15002 General Electric Company 1969

9 J F OSSANNA L E MIKUS S D DUNTEN
Communications and input/output switching in a multiplex computing system
AFIPS Conference Proceedings Vol 27 Part 1 1965 (1965 Fall Joint Computer Conference) Spartan Books Washington D C pp 231-241

10 J H SALTZER J F OSSANNA
Remote terminal character stream processing in multics
AFIPS Conference Proceedings Vol 36 1970 (1970 Spring Joint Computer Conference) AFIPS Press Montvale New Jersey pp 621-627

11 P E JACKSON C D STUBBS
A study of interactive computer communication
AFIPS Conference Proceedings Vol 34 1969 1969 Spring Joint Computer Conference AFIPS Press Montvale New Jersey pp 491-504

Multiprogramming in a medium-sized hybrid environment

by W. R. DODDS

Bell Helicopter Company
Fort Worth, Texas

INTRODUCTION

Of the many roles that the digital computer plays in a full hybrid environment, few are amenable to multiprogramming. Usually, the speed mismatch between a modern high-speed analog computer and the typical scientific digital computer used in hybrid systems demands that the digital be fully dedicated to the particular task in progress. However, the use of the digital computer for analog computer setup and checkout is a function which is performed several times a day in an active hybrid computation laboratory, and this task does not make any severe demands upon the digital computer. This paper describes a multiprogramming system used to effectively set up and check out an analog computer while still performing routine background processing such as compilation.

SYSTEM CONFIGURATION

A block diagram of the Bell Hybrid Facility is shown in Figure 1. It consists of an IBM 360/44 digital computer linked to two SS-100 analog computers manufactured by Hybrid Systems, Inc. This hybrid system is applied to problems in the areas of structural design, vibration analysis, rotor dynamics and flight simulation. It is used on a one shift basis with a staff of three applications engineers.

DESIGN PHILOSOPHY

The software to be described has evolved after two years of use in an active hybrid facility. A hardware limitation of the Bell hybrid system prevents the *simultaneous* addressing of both analog consoles from the digital; hence, the software was not designed to set up two analog consoles at the same time. In fact, the operating system DAMPS does not permit the simultaneous execution of two real time jobs. As the setup and checkout software was originally specified, it resembled more of a "wire-checker", and the FORTRAN statements specified the outputs of amplifiers in terms of preceding amplifiers and potentiometers. However, it was soon found to be advantageous to specify *every* component in terms of physical variables, physical constants and scale factors. This permitted easy rescaling, facilitated reallocation of components, and provided good documentation of potentiometers and amplifier outputs.

An important feature of the setup software is that it is user-oriented. The FORTRAN definitions of pot settings, etc., are manipulated into a subroutine by a job control procedure library. Scale factors and static check conditions are usually read in at execution time on data cards, providing a rapid means of optimizing the static check. This constitutes the "paper

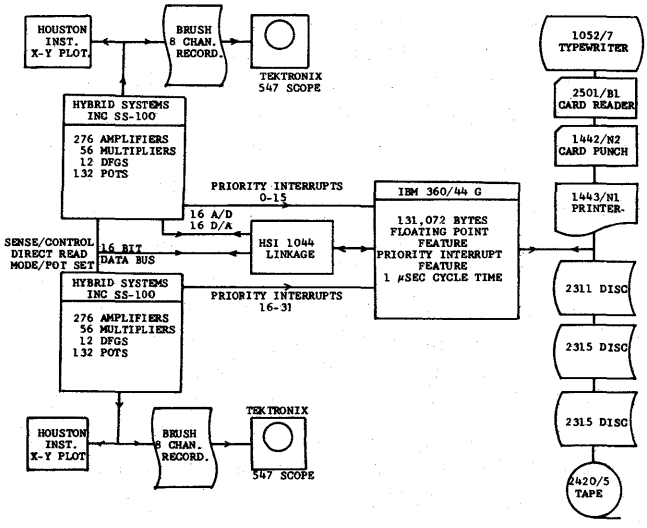


Figure 1—Configuration of Bell Helicopter hybrid computer facility

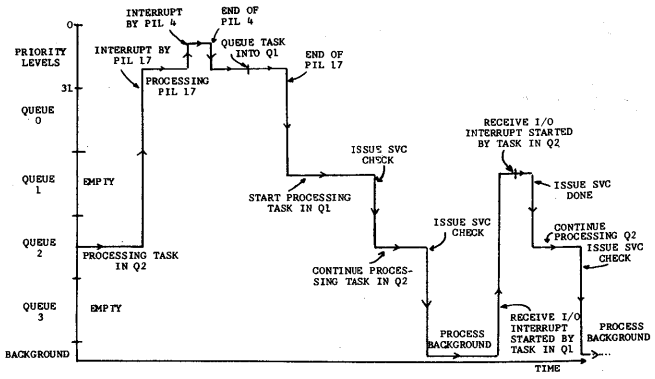


Figure 2—Illustration of task switching under various conditions

work" debug of a static check. This phase is kept distinct from the hybrid execution of a static check, at which time the actual wiring of the patch panel is checked. This natural breakdown of the debugging procedure into two phases permits one programmer to be optimizing his static check while another is debugging the wiring of his problem.

OPERATING SOFTWARE

The heart of the multiprogramming software is the Data Acquisition Multiprogramming System (DAMPS), which has been tailored to suit the specific requirements of the Bell Hybrid Facility. DAMPS is a "priority based" system operating on three levels, viz., priority, foreground and background. The system comprises 32 priority interrupt levels, four foreground queues and a background queue. Processing is performed at the highest priority active at any given instant. When a real time task (RTT) runs to completion, lower level RTT's (if any) are initiated. If there are no pending *priority* interrupts, then the foreground queues are examined and the foreground task (FGT) with the highest "priority" in a given queue gets control. While the system is waiting for completion of any data processing type I/O (DP I/O) command issued by the current FGT, control is passed to lower level queues or to the background. Foreground tasks within a given queue are executed serially according to the specified priority of each task. This system of priorities is illustrated in Figure 2.

Two fixed partitions of core storage have been allocated for the real time jobs (priority interrupt routines and foreground tasks) and the background jobs. Background jobs are initiated using the normal job control processor and jobs may be stacked in the card reader for sequential execution. Real time jobs (RTJ's) are initiated from the console typewriter using a special real time loader. Consequently, real time jobs can be initiated quite independently of any background job that may be running. Likewise, either the background or the real time job may be cancelled prematurely by the operator, each independently of the other.

It is important to understand the mechanism of switching between tasks. Only two events can cause control to pass to a lower level queue.

a. The execution of a supervisor routine 'SVC DONE' which indicates the termination of the current FGT, at which time the queues are then re-examined to determine which task should be executed next.
b. If the current FGT issues an SVC WAIT or an SVC CHECK, and the system is waiting for completion of the I/O, then control is passed to a lower level queue.

Similarly, two events can cause control to pass from a lower level queue to a higher level queue.

a. If a routine is queued by a program operating on a priority interrupt level, then upon completion of the real time task, the queues are examined to determine if the "newly queued" routine is in a higher queue than the task currently in control. If this *is* the case, the multiprogramming monitor is called and a task switching occurs.
b. If an I/O interrupt occurs, the queue from which the I/O command was issued is examined. If it is of a higher priority than the task currently in control, then again a task switching occurs, and activity on the lower level task is suspended.

INPUT/OUTPUT DEVICE ASSIGNMENTS

In any multiprogramming system, the allocation of the various I/O devices must be chosen with care.

Under DAMPS, the normal division of I/O devices between the real time jobs and the background jobs is shown below.

Real Time Job	*Background*
Console Typewriter	Console Typewriter
Card Punch	Card Punch
	Line Printer
	Card Reader

However, a modification was made which permitted the use of the line printer by both an RTJ and a back-

ground job. The resolution of a printer conflict between the RTJ and the background is presently left up to the operator.

CHANNEL OPERATION

Basic to the operation of this multiprogramming scheme is the concept of the IBM 360 channel. In this particular application, for example, a channel program is set up to set all 120 servo-set potentiometers, or to scan all 346 addressable components. Once the I/O is initiated, the channel accesses main storage on a cycle stealing basis, and the CPU is free to perform other processing. The I/O operations involved in setting up an analog computer typically take several minutes (pot set, pot scan, amplifier scan) and this time can be used to advantage for background processing. However, the multiprogramming scheme offered by DAMPS was only designed to work on DP I/O type operations and not "real time" I/O operations. A special program was written which made a real time request control block (used for starting up an I/O operation) resemble a DP I/O request control block, prior to issuing an SVC CHECK.

ANALOG SETUP SOFTWARE

The essential requirements of a software package to set up and check out an analog computer include the following:

1. A means of defining all potentiometer settings in terms of physical variables and scale factors.
2. A means of defining all amplifier outputs, either in terms of physical variables, or as a product of a preceding amplifier output and a pot setting.
3. Tabulation of theoretical values of all pots and all addressable components with flagging of outputs exceeding the scaling of the variable.
4. A means of setting up all the servo set potentiometers.
5. Provision of a potentiometer scan and comparison of theoretical values and actual analog values.
6. Provision of a component scan with a comparison between theoretical values and actual analog values with error flagging.

There are two distinct stages in debugging a static check program. The first task is to ensure that the problem is correctly scaled and that static check values have been realistically chosen. This constitutes a "paper work debug" of a static check program. The second phase is to perform the static check on the analog computer, checking the wiring and correct functioning of the components. This can sometimes be a time-consuming stage requiring several iterations. The Bell Analog Setup Software Package (ANASET) thus consists of two main stages.

The first program uses pot setting and amplifier output definitions to generate three major arrays—an array containing the theoretical pot settings in floating point format, an array containing the theoretical amplifier outputs in floating point format, and an array containing the servo-set pot addresses interleaved with pot settings in BCD format for only the pots being used. An option is available to write these three arrays out to a disc. This is used when the "paper work debug" is finally complete. This first program also prints out a listing of the theoretical values of all pot settings and amplifier outputs. Perhaps the most important feature of this program is that it can be executed at the background level.

The second program is compiled to be executed as a real time job using the real time partition. It accesses the three arrays on the disc, sets up the potentiometers, scans the potentiometers and compares the theoretical values with the read values. The program then pauses to permit adjustment of any erroneous pots.

The static check is then performed by scanning all components in the analog STATIC CHECK mode, and comparing the theoretical values with the actual analog values. Being an RTJ, it is initiated from the console typewriter, and during the periods of prolonged real time I/O, the background is free to continue processing. Options are programmed into the RTJ to bypass any of the I/O operations. This facility is provided to permit re-execution of the static check without the setting up of potentiometers which may have just been set up and checked. Some salient details of these two programs will now be discussed.

Anaset (background array preparation)

A detailed flow chart of this program is shown in Figure 3.

An intrinsic part of the ANASET program is a subroutine called STMTS. This subroutine contains all the user-defined pot definitions and amplifier outputs. The parameter list of this subroutine consists of the names of all the arrays used to define components. A typical (and simplified) STMTS subroutine is shown in Figure 4. Its operation is best described by a simple analog program.

Let $dV/dt = -C_1V + C_2F$

and

$dX/dt = C_3V$

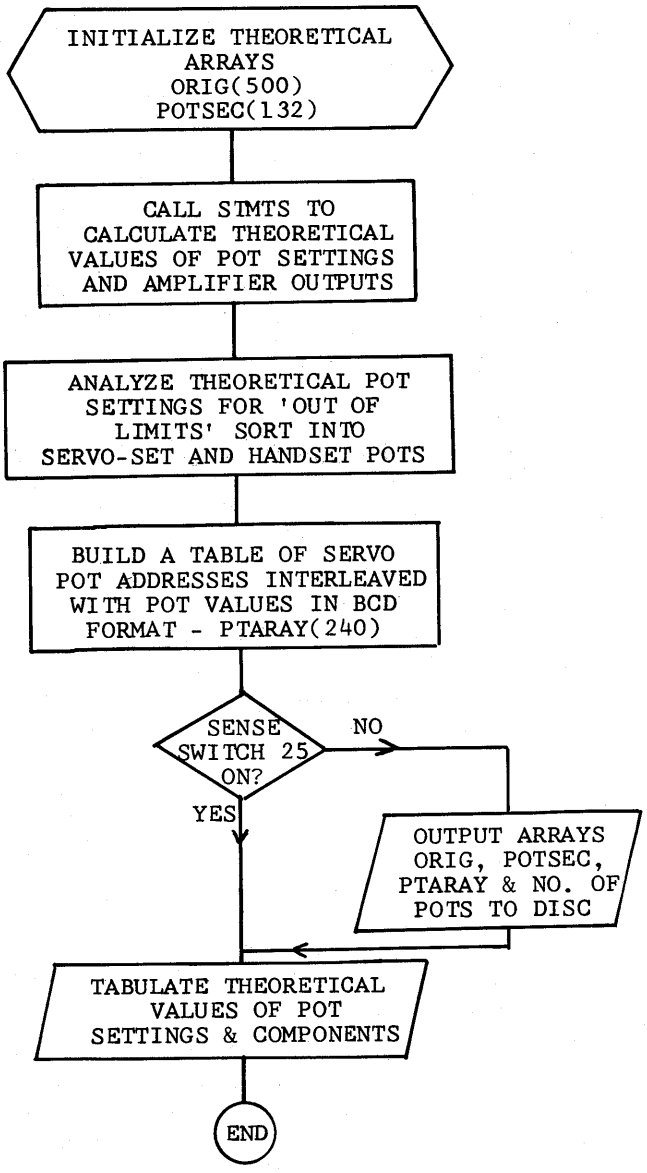


Figure 3—Flow chart of program 'Anaset'

The analog program is shown in Figure 5.

Referring to Figure 4, statements 10–12 specify initial conditions of prime variables used for the static check. Statements 15–17 specify the values of any constants. All of the above could also be read in using the card reader. Statement 18 represents a physical equation describing the time derivative of V. Statements 20–22 are analog scale factors, etc. Statements 30–35 are the pot definitions in terms of the physical constants. Statements 40–41 define amplifier outputs. Statement 42 defines a check derivative in terms of the physical equation. Statement 43 shows an alternative means of specifying a check derivative in terms of previously defined amplifier outputs and pot settings. The first method is more rigorous as it relates a specific component to a physical equation, facilitating debugging.

```
      SUBROUTINE STMTS (P0,P1,P2,P3,P4,P5,M0,M1,M2,
     1  M3,M4,M5,A0,A1,A2,A3,A4,A5,DA1,DA2,DA3,DA4,DA5,
     2  VG0,VG1,VG2,VG3,VG4,VG5,SC4,SC5,SH,DR,RM4,RM5,DI)
      REAL*4 P0(26),P1(26),P2(26),P3(24),P4(15),P5(15),
     1  M0(6),M1(6),M2(6),M3(6),M4(3),M5(3),A0(26),A1(29),A2(29),
     2  A3(31),A4(32),A5(35),DA0(15),DA1(15),DA2(15),DA3(15),
     3  DA4(9),DA5(9),VG0(3),VG1(3),VG2(3),VG3(3),
     4  VG4(10),VG5(10),SC4(42),SC5(24),SH(16),DR(16),
     5  RM4(19),RM5(19)

10    F = 4.92
11    V = 29.3
12    X = 19.002

15    C1 = 8.294
16    C2 = 3.142
17    C3 = 16.8

18    DVDT = -C1*V+C2*F

20    EK1 = 50.
21    EK2 = 1000.
22    VOLT = 100.

30    P1(01) = C2*F/EK1
      P1(04) = C1/10.
      P3(09) = C3*EK1/EK2
35    P4(02) = V/EK(1)

40    A2(01) = V/EK1*VOLT
41    A2(03) = -X/EK2*VOLT

42    DA2(01) = -DVDT/EK1*VOLT
      OR
43    DA2(01) = -P1(01)*100.+P1(04)*A2(01)
        .
        .
        .
      RETURN
```

Figure 4—Subroutine STMTS showing parameter list and typical statements specifying pot settings and amplifier outputs

The remaining routines in program ANASET employ usual FORTRAN techniques to manipulate the arrays produced by STMTS, to test for components specified out of limits and to tabulate the theoretical values of all components.

Program SSS (real time setup program)

This is the program which uses the three major arrays previously stored on the disc by ANASET. All messages in this program are directed to the console typewriter with the exception of the final static check output which uses the printer. A flow chart of program SSS is shown in Figure 6.

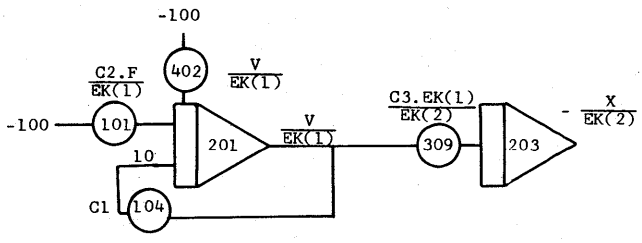


Figure 5—Analog diagram of sample program

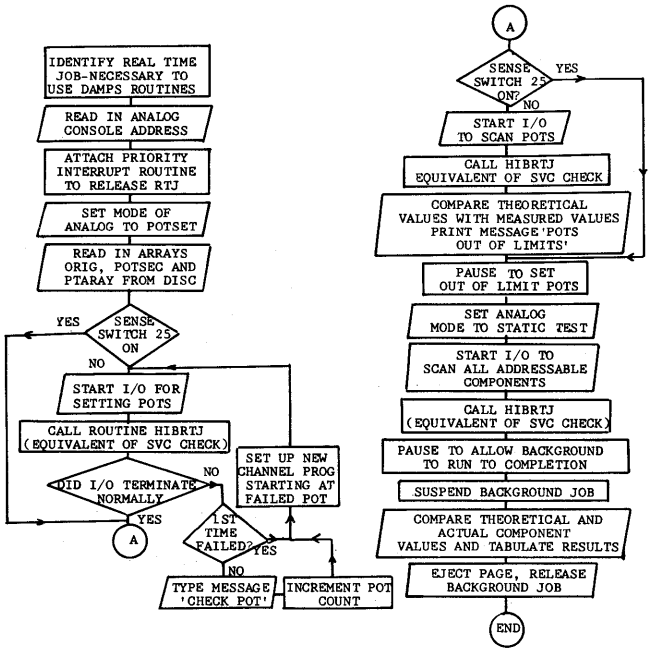


Figure 6—Flow chart of program SSS

As discussed earlier, a special subroutine is used to effectively issue an SVC CHECK after a real time I/O operation has been initiated. This allows control to pass to any background processing until the I/O is complete. During pot set, the I/O can be terminated abnormally if a certain pot fails to set properly. If this occurs, the program SSS regains control from the background, determines which pot did not set (by examining the residual byte count from the channel status word) and sets up a new channel program to continue setting pots starting at the failed pot. If the same pot still fails to set correctly, a warning message is output on the typewriter and the program continues to set up the remaining pots. This type of abnormal I/O recovery procedure is so rapid that it causes no *apparent* delay to the background.

There is one very important problem area related

FUNCTION	MEMORY REQUIREMENTS	TIME FOR EXECUTION
ANASET Compilation and Linkage Edit	Background Partition 48K Bytes	20 Minutes
ANASET Execution	Background Partition 31K Bytes	2 Minutes
SSS Execution	Real Time Partition 34K Bytes	Approximately 9 Minutes
1. Set 190 Servo Pots		6 Minutes
2. Scan 132 Pots		16 Seconds
3. Adjust Out of Limit Pots		1 Minute
4. Scan All 346 Addressable Components		41 Seconds
5. Print Out Static Check Results		1 Minute

Figure 7—Execution times and memory requirements for Anaset and SSS

```
THEORETICAL VALUES OF POTENTIOMETERS

          P108    =    0.5000
          P114    =    0.2020
          P121    =    0.7642
HANDSET   P125    =    0.1000
          P302    =    0.6459
          P401    =    1.6321 **OUT OF LIMITS**
          P512    =    0.0
```

```
THEORETICAL VALUES OF COMPONENTS

Component        Theoretical Value

   M001              -1.324
   M002              36.429
    :
    :
   A001              10.000
   A002             -15.462
   A109             168.562 **OUT OF LIMITS**
DFG 410             -16.829
S/H 004               1.621
D/R 009              49.998
```

Figure 8—Typical theoretical printout generated by Anaset

to the execution of program SSS. Toward the end of the routine after the amplifier scan is complete, SSS tabulates the results on the printer. However, if the background is processing a compilation and the real time job were given immediate control, the results of the static check would be interleaved with the back-

```
TYPEWRITER LISTING                                  COMMENT

IA55I 135106                                        Time of day output by system
//XFHD01 JOB, 671531DDT0100038                      Job card of background job
rtload                                              Call real time loader
//rtj                                               Indicate that an RTJ is to be executed
IA55I 135329                                        System provides time of day
//exec rrr                                          Start execution of RRR (provides standard job control)
***END OF REAL TIME JOB***                          Indicates end of job
rtload                                              Call real time loader
IA55I 135412                                        System provides time of day
//sys018 access dsdick,091='reltim'                 Access disc data set for arrays
IA89I M091 RELTIM                                   Message for information
//exec sss                                          Start execution of SSS
IA90A M ALL REQ DISCS                               Mount the disc specified above

ENTER CONSOLE NUMBER..0 OR 1                        Indicate which analog is to be set up
0                                                   Analog console 0
CHECK P113=5000                                     Pot P113 did not set properly
IA55I 140943                                        System provides time of day for new background job
//AGHD09 JOB,676728DDP0200006                       Job card for background job
CHECK P320=6782                                     Pot P320 did not set properly
PAUSE TO SET OUT OF LIMIT POTS                      All pots set up, adjust faulty ones
THESE POTS OUT OF LIMITS                            Pot scan complete, list out of limit pots
P026=.5000 DIFF=.0011                               Correct value of P206 is .5000
PAUSE TO ADJUST POTS                                Pause to adjust pots just listed
PAUSE TO ALLOW BKGR. TO FINISH                      Amplifier scan complete, allow background to complete
                                                      if necessary
FA83A INT REQ 00C                                   Background job complete
STOP 0                                              Printout of static check complete
END OF REAL TIME JOB                                Program SSS complete

        Note:  Only messages appearing in small letters are input by the operator.
```

Figure 9—Typewriter log during multiprogramming an analog setup and a background job

STATIC CHECK RESULTS

Component	Theoretical Values	Actual Analog Values	Difference
M001	-1.324	-1.320	
M002	36.429	36.300	
⋮			
A001	10.000	9.542	0.458
A002	-15.462	+15.462	30.924
⋮			
DFG 410	-16.829	-16.729	
S/H 004	1.621	0.0	1.621
D/R 009	49.998	49.800	0.198
⋮			

Note: The fourth column is only printed out if the difference exceeds 150 MV.

Figure 10—Typical static check results produced by program SSS

ground compilation listing, which is, of course, totally unacceptable. Prior to the listing of the static check results, a pause message is output on the typewriter, giving the operator the option of waiting until completion of the background job. If this is impossible, then before the real time job gets control of the printer, the background job is suspended. The static check results start at top of form and at completion of the printout, the last page is ejected. Thus, the background job can continue processing and any output appears on a new page.

PROGRAM TIMINGS AND MEMORY REQUIREMENTS

Figure 7 shows the various steps involved in setting up an analog computer program using the full complement of the computer. Once the static check program is debugged, it is rarely changed, and only program SSS is executed on a recurring basis. Experience has shown that a complex analog simulation using the full complement of a computer takes about 30–40 minutes to set up. Since the complement of the analog computer is fixed, the memory requirements of program SSS do not vary and the operator merely accesses his own particular data set containing his arrays. The memory requirements of ANASET depend upon the size of the user's STMTS routine defining the pot settings and amplifier outputs. An upper limit on this routine (STMTS) is of the order of 25K bytes (defining *every* component in terms of physical variables).

Figure 8 shows a typical printout of the theoretical values of components produced by ANASET. Figure 9 shows part of the typewriter listing produced while multiprogramming a background job with the execution of program SSS. Although this may appear confusing at first glance, the messages in lower case letters are actually input by the operator, the remainder are messages output by the system or the real time program. Figure 10 shows some typical static check results produced by program SSS.

CONCLUSIONS

A sophisticated software package for setting up and checking out an analog computer while multiprogramming background tasks has been outlined. The operating software is based upon the 360/44 DAMPS along with special modifications to permit multiprogramming on *real time* I/O operations. The analog setup software provides a means of checking analog wiring directly against physical equations, offers excellent analog program documentation and facilitates paper work debugging of a static check program without tying up the real time facilities of the hybrid system. An entire static check of a program using the full complement of the analog computer can be performed in less than forty minutes.

ACKNOWLEDGMENTS

This material is presented by kind permission of Bell Helicopter Company.

The binary floating point digital differential analyzer

by J. L. ELSHOFF and P. T. HULINA

The Pennsylvania State University
University Park, Pennsylvania

INTRODUCTION

Twenty years ago the digital differential analyzer, DDA, was developed to replace the analog computer in the solution of differential equations. Although the DDA is slower than the analog computer, the DDA is capable of more accurate results since its accuracy is not bounded by its component characteristics. The cost of solving differential equations with the DDA is quite low compared with other methods such as a general purpose machine, since the DDA is a more simple device.

As time has passed, advances have been made in DDA technology. These advances have resulted in increased speed and accuracy,[1,2] reductions in cost,[1] and improvements in man-machine interface.[3] Still the DDA is seldom used except as a special purpose device. Despite the dependence of the problem solution on the quality of the components and the higher cost of the analog computer, analog computation continues to grow in popularity. Similarly, general purpose computers continue to be more widely used even though they cost more and solve differential equations at a slower rate than the DDA.

In recent years analog and digital computers have been combined into hybrid systems.[3,4] In theory, the hybrid system takes advantage of the high speed of the analog computer and the easy programmability and decision capabilities of the digital computer. In practice, however, the speed of the analog computer is greatly reduced in operational performance by digital software and the digital-to-analog and the analog-to-digital conversion hardware. The general purpose digital computer can be programmed in an easy problem-oriented language like Fortran while the analog portion of the problem must be physically patched.

This paper concerns itself with a brief review of DDA technology and an investigation of ways in which to expand that technology. The emphasis is placed on increasing the speed, reducing the cost, and improving the utility of the DDA in such a way that the DDA would replace the analog computer and provide a more practical hybrid system.

THE DDA

The vector form of the general linear homogeneous constant coefficient ordinary differential equation can be written

$$\dot{x}=Ax, \qquad x(0)=x_0 \tag{1}$$

where A is a constant $m\times m$ matrix, and x and $\dot{x}$ are $m\times 1$ column vectors. In rectangular integration the vector difference equation that replaces equation (1) is

$$y(n+1)=y(n)+\dot{y}(n)\,\Delta t=y(n)+\Delta y(n) \tag{2}$$

where from equation (1)

$$\Delta y(n)=\dot{y}(n)\,\Delta t=(\Delta t)Ay(n).$$

For any given value of $y(0)$, an iterative solution of equation (2) can be obtained. If $y(0)=x_0$, then $x(t)$ is approximated through the relation

$$x(n\Delta t)=y(n)+0((\Delta t)^2)$$

where $0((\Delta t)^2)$ represents the truncation error and $n\Delta t=t$.

In a DDA the fractional part of $y(n)\,\Delta t$ is held in a residue register (R register) and only the integer part is used in equation (2). Let $R(n)$ be the contents of the R register at $t=n\Delta t$. Then the equation for $y(n)\,\Delta t$ is modified to

$$y(n)\,\Delta t+R(n-1)=\Delta Z(n)+R(n)$$

where $\Delta Z(n)$ is a signed integer and $|R(n)|<1$. Bartee, Lebow, and Reed,[5] Huskey and Korn,[6] and

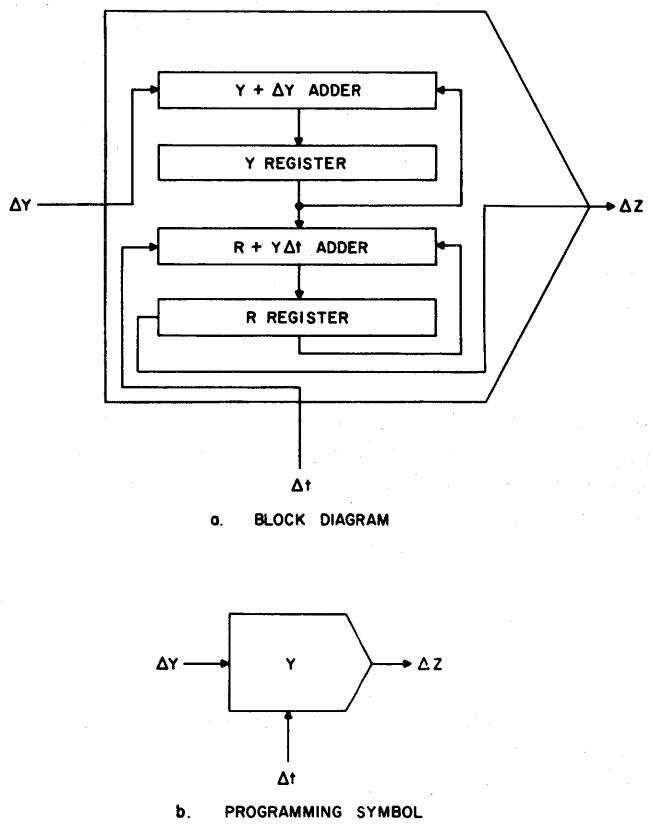


Figure 1—Basic DDA integrator

McGhee and Nilsen[2] explain the mathematical principle of the DDA in detail.

Figure 1 displays the block diagram and a programming symbol for the DDA integrator. The Y register contains the value of $y(n)$. The ΔY input holds the value of Δy, the incremental change of the integrand, at each step. The ΔZ output and R register contain the integral and residue of $y(n)\Delta t$, respectively. Finally, the Δt input holds the value of Δt, the step size of the independent variable.

With these values available, the iterative solution to equation (2) can be completed. A single step of rectangular integration is performed by the transfer equations

$$\text{Integration phase: } Y\Delta t+R\rightarrow\Delta Z+R$$

$$\text{Incrementation phase: } Y+\Delta Y\rightarrow Y$$

where ΔY is a weighted sum of ΔZ outputs. By alternating the integration and incrementation phases, the solution to equation (2) is iteratively realized.

In operation the inputs and outputs in a DDA are represented by two binary bits, the sign bit and coefficient bit. For an arbitrary problem the input and output increments each represent a fixed magnitude. For example, if $\Delta Y=c$, where c is a constant, then the coefficient bit is one or zero depending on whether or not there is a ΔY during each particular incrementation phase. The sign bit is one or zero depending on whether ΔY is negative or positive. Thus, the value c is fixed for the problem during the programming and is not actually transferred during the solution. Only a signed coefficient of one or zero is transferred.

In practice the inputs and outputs in a base e DDA are coefficients of incremental values that are equal to integer powers of e. By choosing a step size equal to 2^{-i} in a binary DDA the $Y\Delta t$ term can be calculated by a simple shift instead of a multiplication. Let $\Delta t=2^{-i}$ for i a positive integer, then the Y register is assumed to be shifted right i positions so that the R and Y registers have their bits aligned. Thus, the integration phase is reduced to a simple addition.

The method of programming a DDA resembles that of programming an analog computer. A certain quantity is assumed to be known. The other values are calculated from the assumed value. Finally, the assumed value is derived back from the known values. The actual program must then be physically patched.

The fixed point arithmetic used in the DDA is a major disadvantage of the DDA. Problems must be magnitude scaled for solution. Although Gill[7] and Knudsen[8] have developed a completely systematic procedure for scaling a DDA, the scaling problem is difficult and solution accuracy depends upon estimated maximum values.

Since the DDA has not enjoyed widespread use, most of the developments in DDA's have been pointed at particular problem solutions. Usually emphasis is placed on increasing the speed and accuracy of the DDA, which happen to be inversely proportional. An improvement in one aspect of DDA's is often compromised by added complications and new problems in other aspects. Like the analog, the inconvenience in using the DDA contributes to its lack of popularity. Because of the lack of popularity, only slight attention has been focused on improving user convenience.

The emphasis in this work was aimed at user convenience in a hybrid computing system. Since the major problems seemed to be in programming and scaling, they were given a high priority. Being digital, two components can be connected or disconnected by passing their connection time through an AND gate with a switching variable that is either on or off, respectively. Thus, in a hybrid system, the general purpose computer can be used to program the DDA. The obvious answer to the scaling problems seemed to be floating point arithmetic. The use of floating point arithmetic was also expected to be more accurate which

is a very desirable effect. Floating point arithmetic was implemented in a DDA design and the design simulated on a general purpose digital computer. The implementation and simulation results appear in the remainder of this paper.

THE BINARY FLOATING POINT DDA

The purpose of this section is to present the binary floating point digital differential analyzer (BFPDDA). The BFPDDA differs from the conventional DDA in that the incremental units being transferred between the components are exponents. Multiple bits must be used to transmit an exponent instead of the usual one or two transmission bits in the regular DDA. Yet with as few as seven bits, signed quantities ranging from 2^{-31} to 2^{+31} can be passed from one component to another component in the BFPDDA.

In the BFPDDA floating point arithmetic is introduced into the conventional DDA structure in place of the normal fixed point arithmetic. The floating point arithmetic transforms the conventional DDA in many ways without losing its basic structure. The altered structure, the mathematical algorithms, and the operation of the BFPDDA are presented in the following sections.

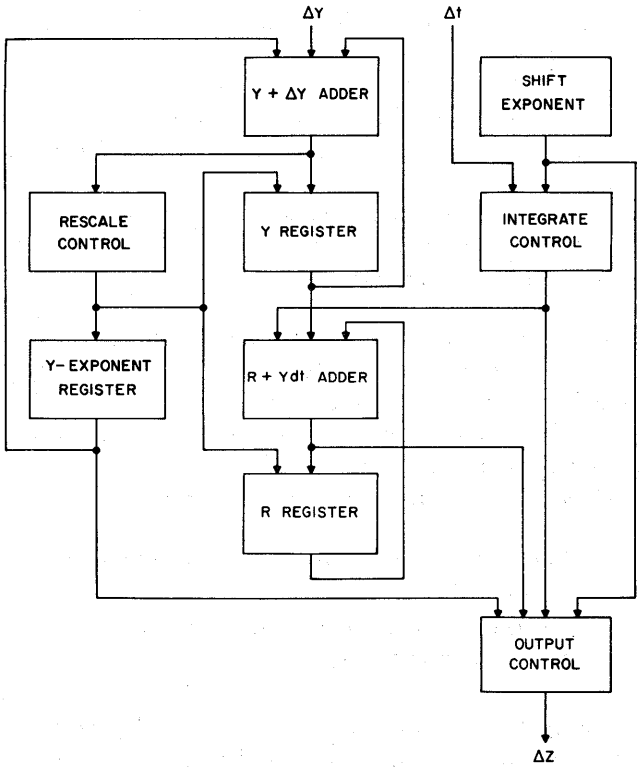


Figure 2—BFPDDA integrator block diagram

THE BFPDDA INTEGRATOR STRUCTURE

The BFPDDA should operate at the fastest speed possible in order to effectively replace the analog computer; therefore, the integrators should operate in parallel. Each integrator with its own adders and control units is assumed to be on an integrated circuit chip. Figure 2 is a block diagram of a proposed BFPDDA integrator showing its basic units and the lines of communication among these units. Note that the four units directly under the ΔY input are the same as the units in a DDA.

Let the current value of the integrand Y be represented by

$$Y = \pm .yyyy * 2^k$$

where yyy is the mantissa and k is the characteristic. Similarly, the residue R is represented by

$$R = \pm .rrrr * 2^{k+j}$$

where $|j|$ is the number of positions the Y register is shifted right so its bits are aligned with the bits of the R register. The value of j is negative so that $R < Y$. Using these definitions, general descriptions of each of the units making up the integrator are briefly given as follows.

$Y+\Delta Y$ Adder—This adder is used to increment the value of the integrand.

Y Register—The Y register contains the mantissa of the integrand.

$R+Y\Delta t$ Adder—This adder performs the integration.

R Register—The R register contains the mantissa of the residue.

Y-exponent Register—The Y-exponent is the characteristic of the integrand.

Rescale Control—The rescale control normalizes the integrand.

Shift Exponent—The shift exponent is the number of places which the R register is shifted left in order to be aligned with the Y register.

Integrate Control—This unit controls the information flow during each iteration.

Output Control—This unit calculates the output increment.

The Y and R registers contain mantissas of floating point numbers in binary coded form. In this paper the left most bit of the register is assumed to be the sign bit. The radix point is assumed to lie between the sign bit and the second bit of the register. Thus, the high order significant bit of the Y and R registers is the second bit. Thus, the $R+Y\Delta t$ adder is a simple integer adder.

Similarly, the Y-exponent and shift exponent registers contain exponents in a binary coded form. The Y-

exponent register contains the number of shift positions the Y register must be shifted to the left for the radix point to be properly positioned. The shift exponent is the number of places the R register is shifted from the Y register.

THE MATHEMATICAL ALGORITHMS FOR THE BFPDDA

The implementation of floating point arithmetic in the BFPDDA slightly alter the integration calculations used in the conventional DDA. The change in number representation requires an additional calculation to determine the output exponent. Finally, in order to make effective use of the dynamic scaling capabilities of a DDA with floating point arithmetic, algorithms for rescaling are included.

The integration phase of each iteration realizes the transfer function

$$R+Y\Delta t\rightarrow\Delta Z+R.$$

The Y and R values are represented by

$$Y=\pm.yyyy*2^{k}$$

and

$$R=\pm.rrrr*2^{k+j}$$

where $yyyy$ and $rrrr$ are the contents of the Y and R registers respectively. The Y-exponent register contains k and the shift exponent register contains j. Let

$$\Delta t=\pm 1.0*2^{i}$$

where $i\leq j$. Then the integration phase is described in Algorithms I and II, where the carry flag is a simple set and reset flip–flop.

Algorithm I.—Integration

1. Shift the Y register $j-i$ positions to the right.
2. Add the shifted Y register to the R register.
3. If the R register does not overflow, reset the carry flag;
   Otherwise,
   a. If the R register is positive,
      i. Decrement the R register by 1.0.
      ii. Set the sign bit associated with ΔZ to positive.
      iii. Set the carry flag.
   b. If the R register is negative,
      i. Increment the R register by 1.0.
      ii. Set the sign bit associated with ΔZ to negative.
      iii. Set the carry flag.

Noticing that requiring $i\leq j$ is a very practical restriction in the BFPDDA structure. The Y register is considered to be shifted $|j|$ positions left in order to be aligned with the R register. Therefore, if the step size of the independent variable is not at least as small as 2^{j}, a multiple bit overflow, which the DDA is not prepared to handle, could occur.

Algorthim II.—Output calculation

1. If the carry flag is set, transmit $k+j$ as Δj along with the sign bit.
2. If the carry flag is reset, transmit no output.

During each iteration, the integrand must be updated so that it is as accurate as possible. The incrementation phase performs the transfer function

$$Y+\Delta Y\rightarrow Y.$$

The value of the integrand Y is the same as previously defined. The value of ΔY is of the form

$$\Delta Y=\pm 1.0*2^{m}$$

where m is the exponent being received on the ΔY input lines along with the correct sign. The procedure used for the incrementation of the integrand is now given in Algorithm III.

Algorithm III.—Incrementation of the integrand

1. If $k\leq m$, invoke Algorithm IV;
   Otherwise,
   a. Shift ± 1.0 right $k-m-l$ positions.
   b. Add the shifted value to the Y register.
   c. If the Y register overflows, invoke Algorithm V.
   d. If the magnitude of the Y register is less than one-half without considering the Y-exponent, invoke Algorithm VI.

An overflow condition occurs when the magnitude of ΔY is larger than the magnitude of Y. Since this means that the change in Y is larger than Y itself, the Y value is considered to be negligible. In this case the integrator is reset as defined in Algorithm IV.

Algorithm IV.—Resetting the integrand

1. Set the Y register to ± 0.5 depending on the sign of ΔY.
2. Set the Y-exponent register to $m+1$.
3. Set the R register to zero.

During the incrementation phase of each iteration, the Y register is treated as a fixed point value. When the Y register overflows, a rescaling of the integrator is performed as described in Algorithm V.

Algorithm V.—Rescaling for an increasing integrand

1. Shift the Y register one position to the right.
2. Shift the R register one position to the right.
3. Increment the Y-exponent register by one.

Just as the value of the integrand may get larger in magnitude, it may also get smaller. By keeping the fractional value of the integrand normalized in the Y register, output overflows will tend to be smaller but will occur more frequently. The procedure for rescaling for a decreasing integrand is shown in Algorithm VI. Notice that this algorithm does not alter the R register. Many tests seemed to indicate that ignoring the R register gave the best results.

Algorithm VI.—Rescaling for a decreasing integrand

1. Shift the Y register one position to the left.
2. Decrement the Y-exponent register by one.

THE OPERATION OF A BFPDDA

As with the conventional DDA, the basic cycle of the BFPDDA is a two phase iteration. The integration and incrementation phases are outlined in Table I.

In phase I the integration is performed. While the Y register is being shifted and added to the R register, the output value is calculated. Then if an overflow occurs, the calculated output value is transmitted, otherwise, no output is transmitted and the calculated output value is ignored.

In phase II the integrand is incremented. An overflow in the addition causes the incremented value to be stored one position to the right and Y-exponent register to be incremented by one. An increment that is larger than the current value of the integrand causes the Y, Y-exponent, and R registers to be reset. On the other hand, if the mantissa is small enough, the result is stored one position to the left in the Y register and the Y-exponent

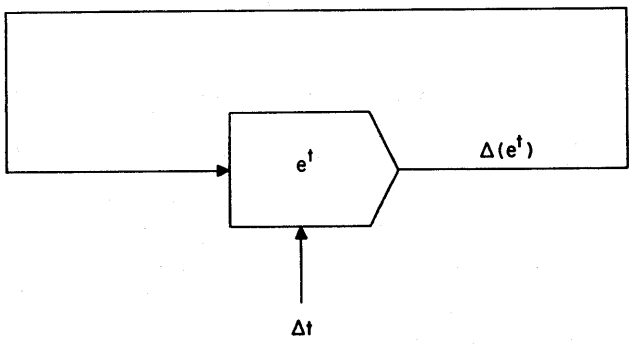


Figure 3—DDA program for e^t

TABLE I—BFPDDA Operational Iteration

I. Integration Phase	II. Incrementation Phase
$R+Y\Delta t \rightarrow \Delta Z+R$	$Y+\Delta Y \rightarrow Y$
1. Integration	1. Incrementation
2. Output Calculation	2. Rescaling

is decremented by one. In this way the integrand remains normalized.

THE ELIMINATION OF MAGNITUDE SCALING

One of the major drawbacks of the ordinary DDA is the fixed point arithmetic it employs. Each integrator must be magnitude scaled in order that the integrand register doesn't overflow during the problem solution causing the program to abort. On the other hand, if the maximum value is overestimated by a significant amount, the error being delayed in the R register of the integrator is much larger, causing the problem error to be increased.

Consider the DDA solution of $\dot{y}-y=0$. The program for this equation is very simple as is shown in Figure 3. Despite the simple program, e^t is one of the more difficult solutions for the DDA to calculate. The exponential represents a continuously increasing function with a large variation in magnitude. Since the accumulated error will never be canceled, the magnitude of the error increases as the value of the independent variable t increases. Also, the initial value of the exponential function is its minimum value which results in very large initial errors since the integrator must be scaled for the values of increasing magnitude of the function.

Table II illustrates the effect of magnitude scaling on accuracy. When the integrator was scaled for a maximum value of 2^3, the error in calculating e^2 with a step size of 2^{-8} was approximately equal to the error in calculating e^2 with a step size of 2^{-13} in an integrator scaled for a maximum value of 2^8. Although e^t may be the extreme case, the exponential demonstrates the loss of accuracy in DDA problem solutions with variables that have a large variation in their magnitude.

The standard DDA has another scaling problem which the exponential exhibits. When the DDA is programmed, the step size must be fixed since the shift between the Y and R registers of the DDA is directly related to the step size. Thus the output line represents a fixed quantity and each output pulse is one unit of that quantity. If the step size is then changed, the out-

TABLE II—Comparison of Maximum Errors in Calculating $e^2=7.3891$ with a DDA Using Varying Magnitude Scalings

Step Size	Maximum Integrator Value $2^3=8$	$2^8=256$
2^{-8}	0.1553	3.3891
2^{-9}	0.0803	1.8891
2^{-10}	0.0406	1.0103
2^{-11}	0.0201	0.5141
2^{-12}	0.0102	0.2623
2^{-13}	0.0051	0.1309
2^{-14}	0.0026	0.0657

put quantity also changes. The new output quantity then must be reprogrammed at each point where it is used as an input to another component.

In calculating the exponential the integrator is magnitude scaled. Suppose the maximum value is set at 2^3 and a step size of 2^{-10} is selected, then each overflow or output bit represents 2^{-7}. The value of e^0 is loaded as the initial condition and e^2 is calculated. According to Table II an error of 0.0392 has occurred. If this error is determined to be too large, a smaller interval must be chosen. Choosing an interval of 2^{-12} makes each output bit represent 2^{-9}. Each increment of Y is then one-fourth of the previous increment size and the integrator must be reconnected to allow for these smaller incremental values.

Fixed point arithmetic in the regular DDA also causes scaling problems for integrators with more than one input incrementing its integrand value. Since the DDA is incremented by receiving a pulse, the increment value is fixed within the integrator being incremented. Therefore, when the integrator receives increments from two or more components, the inputs must represent the same value.

The magnitude scaling problem does not exist in the BFPDDA. As the integrand values become larger the integrator rescales upward. Similarly, the integrator rescales downward as the values become smaller in magnitude. The user does not have to estimate the maximum values or even know much about the relationships among the integrators since each integrator functions independently of the other integrators without regard to the integrand magnitude.

Since the BFPDDA integrators rescale themselves, the accuracy of the BFPDDA is better than that of a regular DDA. When the integrand magnitude becomes smaller, the integrator rescales downward causing the fractional error of the integral in the R register to be reduced. A smaller overflow then occurs in the integration; however, the overflow occurs much sooner and introduces much less delay error than in the standard DDA. Thus the BFPDDA causes more, but smaller overflows than the ordinary DDA in solving most integrals which leads to more accurate solutions.

Two solutions of e^t were calculated using the BFPDDA. After each iteration the calculated value was compared with the actual value and the error determined. The maximum errors in calculating e^2 and e^5 for varying interval sizes is shown in Table III. The same values were also calculated with a regular DDA and also appear. When the magnitude scaling was done for e^2, only a three bit binary shift was necessary and the DDA was practically the same as the BFPDDA. However, when an allowance of eight bits was made for e^5 in the DDA and two and one-half times as many iterations were performed, the BFPDDA was over ten times as accurate as the DDA. The BFPDDA could then solve e^5 ten times as fast as the DDA for the same accuracy, since speed and accuracy are inversely proportional.

The values appearing in Table III indicate that the BFPDDA is more accurate because of its rescaling techniques and floating point arithmetic. Over the shorter solution of e^2 the accumulative error wasn't too critical. When a slightly longer problem was solved with a larger range in integrand magnitudes, the accumulative error in the DDA greatly impaired its advantages but only minimally effected the BFPDDA.

Other factors not appearing in Table III also make the BFPDDA preferable. In changing from one interval size to another, the shift exponent register was merely reloaded with the exponent of the step size chosen for maximum accuracy. Otherwise, there was no reconnection for the new problem solution. The values taken from the BFPDDA were directly readable. There was no multiplication by a scaling factor necessary to put the calculated result in a form relative to that of the unscaled, original equation.

TABLE III—Comparison of BFPDDA and DDA in Accuracy Solving e^t

Maximum Error in Calculating $e^2=7.3891$		Step Size	Maximum Error in Calculating $e^5=148.41$	
DDA	BFPDDA		DDA	BFPDDA
0.1553	0.1524	2^{-8}	65.413	5.543
0.0803	0.0790	2^{-9}	35.624	2.835
0.0406	0.0392	2^{-10}	18.413	1.413
0.0201	0.0198	2^{-11}	9.393	0.716
0.0102	0.0099	2^{-12}	4.726	0.359
0.0051	0.0050	2^{-13}	2.367	0.180
0.0026	0.0025	2^{-14}	1.187	0.091

Another advantage of the BFPDDA appears when second or higher order equations are solved. An integrator receives its integrand increments in the form of an exponent instead of a pulse. Since the magnitude of the increment is passed between the integrators, the integrator may receive increments of different magnitudes during the problem solution. Thus, the integrator may have two or more increment inputs without having the multiple inputs scaled to represent the same magnitude.

In the BFPDDA magnitude scaling has been eliminated. Each integrator dynamically rescales itself independently as the magnitude of its integrand varies. The accuracy in the BFPDDA is not dependent on the estimated maximum value used in the magnitude scaling. With the BFPDDA the independent variable step size may be changed easily with no reconnection necessary. Finally, an integrator receiving an increment from more than one component does not require all of the incremental values to be of equal magnitude.

BFPDDA SOLUTION OF THE HARMONIC EQUATION

The BFPDDA program for solving the harmonic equation, $\ddot{x}+\omega^2 x=0$ is shown in Figure 4. The result of the solution is obtained at any given time during the solution by reading the current value contained in the sin ωt integrator, which in this problem is simply used as a summer. In the results shown in this section, sin ωt was read after each iteration and the error calculated at each point.

The BFPDDA can solve the same problem with many different parameters without being rescaled or reprogrammed. Even large variations in the magnitudes of the parameters have almost no effect. Table

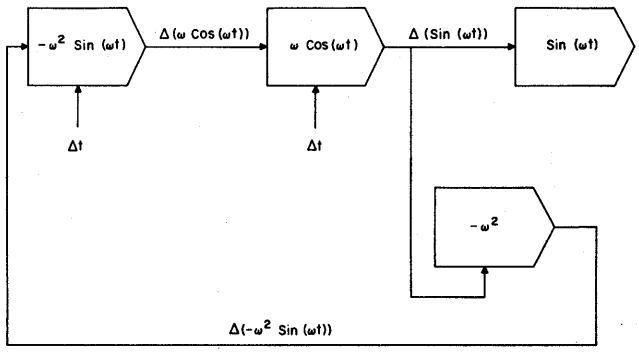


Figure 4—Interconnection of BFPDDA integrators to solve harmonic equation

TABLE IV—Maximum Errors in One Cycle Sine Wave Solutions

ω^2	Step Size 2^{-10}	2^{-12}	2^{-14}
0.4437	0.00662	0.00165	0.00040
0.5653	0.00881	0.00217	0.00055
1.8120	0.01093	0.00271	0.00064
3.3890	0.01328	0.00361	0.00081

IV displays the maximum error that occurred during the one cycle solution of a sine wave. The frequencies were generated randomly from successive ranges bounded by powers of two.

In the BFPDDA all the values shown in Table IV were calculated without reprogramming or rescaling. In the standard DDA a change to a smaller step size would necessitate reconnection so that the proper values were represented. Similarly, the DDA has to be rescaled each time the step size changes or the magnitude of the frequency is significantly altered. In the case of the BFPDDA the step size is changed by simply reloading the shift exponent registers and setting a smaller exponent on the independent variable input lines. A new frequency is simply reloaded in the Y register of the integrator being used for constant multiplication and the initial condition reset on the $\omega\cos(\omega t)$ integrator in order to solve the harmonic equation for a new ω^2 value.

The maximum errors increase as the frequency increases since one cycle of the sine wave is broken into fewer intervals. Thus, as higher frequency sine waves are generated, smaller intervals may be necessary in order to maintain accuracy. The smaller intervals will not increase the overall problem time, however, since the range of one cycle is much smaller.

The harmonic equation was also used for comparing the accuracy of the BFPDDA against the accuracy of the DDA. For this test $\omega^2=1$ and the integrator containing ω^2 was replaced by an inverter on the sign transfer bit. The results which appear in Table V show that the BFPDDA is twice as accurate as the DDA.

CONCLUSIONS

The emphasis in designing the BFPDDA has been on improving user convenience.[9] By reconstructing a standard DDA to use floating point arithmetic and to transfer exponents between its components, an easily programmable device requiring no magnitude scaling resulted. Moreover, the floating point arithmetic proved the BFPDDA to be more accurate than the

TABLE V—Maximum Errors in a One Cycle Solution of the Harmonic Equation with $\omega = 1$

Step Size	DDA	BFPDDA
2^{-8}	0.03209	0.01605
2^{-9}	0.01477	0.00797
2^{-10}	0.00709	0.00406

ordinary DDA. The accuracy improvement becomes more significant as the variation in magnitude of the problem variables increases. The BFPDDA also allows for alterations in the rate of integration during a problem solution since no new scaling is necessary.

Considering current technology and the BFPDDA, the hybrid computing system could be headed for new horizons. The BFPDDA with all main advantages of digital computation in an analog environment will be an excellent special purpose differential equation solver on-line with a time-shared digital computer. Dynamical systems with unknown solutions can quickly be solved since the solution will not depend on estimated maximum parameter values. Being digital the whole field of automatic patching by program can make the BFPDDA easier to use. Making the DDA floating point and greatly increasing user convenience at only a very slight cost increase should make hybrid computation very popular.

REFERENCES

1 M W GOLDMAN
Design of a high speed DDA
AFIPS Conference Proceedings Fall Joint Computer Conference pp 929–949 1965

2 R B McGHEE R N NILSEN
The extended resolution digital differential analyzer: a new computing structure for solving differential equations
IEEE Trans on Computers Vol C-19 pp 1-9 January 1970

3 T C BARTEE J B LEWIS
A digital system for on-line studies of dynamical systems
AFIPS Conference Proceedings Spring Joint Computer Conference pp 105–111 1966

4 M W HOYT W T LEC O A REICHARDT
The parallel digital differential analyzer and its application as a hybrid computing system element
Simulation Vol 4 pp 104–113 February 1965

5 T C BARTEE I L LEBOW I S REED
Theory and design of digital machines
McGraw-Hill New York pp 252–265 1962

6 H D HUSKEY G N KORN
Computer handbook
McGraw-Hill New York Chapter 3 pp 14–74 1962

7 A GILL
Systematic scaling for digital differential analyzers
IRE Trans on Electronic Computers Vol EC-8 pp 486-489 December 1959

8 H K KNUDSEN
The scaling of digital differential analyzers
IEEE Trans on Electronic Computers Vol EC-14 pp 583-590 August 1965

9 J L ELSHOFF
The binary floating point digital differential analyzer
PhD dissertation The Pennsylvania State University University Park Pennsylvania September 1970

Time sharing of hybrid computers using electronic patching

by ROBERT M. HOWE

University of Michigan
Ann Arbor, Michigan

and

RICHARD A. MORAN and THOMAS D. BERGE

Applied Dynamics
Ann Arbor, Michigan

INTRODUCTION

Ever since the introduction of patchboards for allowing storage of analog computer programs, the desirability of having a remotely-controlled switch matrix to replace the analog patchboard has been evident. Only recently, however, has automatic patching received widespread interest and study.[1,2,3] One reason for this is the current widespread availability of hybrid computer systems, with the result that the automatic-patching program can be stored and implemented through the general purpose digital computer. Not only have hybrid computers made automatic patching of the analog subsystem, therefore, more feasible, but also hybrid computers have emphasized the desirability of having automatic patching.

Additional technological developments have made automatic patching feasible. The availability of low-cost, high-performance solid-state switches for implementing the necessary switching matrices is very important. Also, low-cost integrated circuit chips can be used to provide the storage of the switch settings. Finally, the availability and widespread use of digitally-set coefficient devices for the analog subsystem allows high-speed setup of all the parameter values as well as component interconnections. This in turn allows complete problem reprogramming within milliseconds which means that time sharing of a single hybrid computer through remote stations is practical. The resulting increase in computer cost effectiveness far exceeds the extra cost of the hardware and software necessary to implement automatic patching of the analog subsystem. In the paragraphs to follow we will describe the details of an automatic-patching system for the AD/FOUR analog hybrid computer shown in Figure 1. As of the writing of this paper more than 60 AD/FOUR computer systems are operating in the field, many of them interfaced with general-purpose digital computers. Because the AD/FOUR design tends to minimize the number of switch elements needed to mechanize an efficient automatic-patching system, it was decided to add this capability to the existing system as opposed to designing a completely new system from scratch. This has the further advantage that any AD/FOUR systems in the field can be updated to include automatic patching.

The next section describes the configuration of the AD/FOUR automatic-patching system. Following sections present an example application, discuss diagnostic considerations, and summarize the system capabilities when operating in a time-shared mode using remote terminals.

DESCRIPTION OF THE SYSTEM

In the AD/FOUR conventional analog programming is achieved using patchcords for interconnections between holes in the large removable patchboard in the center of the console shown in Figure 1. This patchboard is divided into four quadrants (hereafter called fields), with a matrix of $32 \times 30 = 960$ holes in each field. The fields are numbered 0, 1, 2, and 3, as shown in Figure 2, and the fields are normally filled with computing components in the order 2, 3, 0, and 1. Thus field 1, in the upper right corner of the patchboard, is

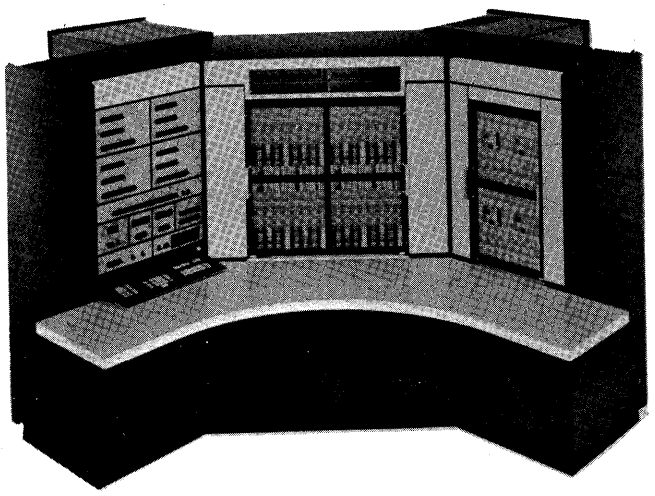


Figure 1—AD-4 computer system

the last to be filled with analog components, and very few AD/FOUR systems are expanded into field 1. For this reason it was decided to terminate the electronic-switch matrices for automatic programming in field 1. These are then patched to the appropriate holes in fields 0, 2, and 3 to mechanize the automatic-patching capability. Thus a single analog patchboard is permanently wired with the automatic program configuration. Note that this patchboard can be replaced by a conventionally-wired patchboard at any time when it is desired to operate the AD/FOUR in the normal patchboard mode.

As can be seen in Figure 1, there is a second patchboard on the right side of the console. This is just half the size of the central analog patchboard, and terminates the logic-level signals associated with the hybrid and digital computing elements in the AD/FOUR. No attempt has been made to implement general-purpose automatic-patching of the logic elements. Instead, it is proposed to accomplish all logic functions in the digital subsystem of the hybrid computer. The logic patchboard is wired to implement the necessary control and timing functions when the AD/FOUR is operating in the automatic-patching mode.

If completely flexible automatic patching were needed, then it would be necessary to be able to connect every analog element to every other element. Since this would require a prohibitively large number of switches, one invariably divides the analog system into identical modules when mechanizing an automatic patching system. Flexible interconnection of analog components within each module is then provided, along with programmable interconnections between modules. In the AD/FOUR computer the module size for automatic patching is an analog field, which includes the following component count:

16	bipolar summer integrators
12	multipliers, each with a bipolar input buffer amplifier
64	digital coefficient units (DCU's)

In addition, each of the 16 summer integrators and 12 multiplier buffer amplifiers has a digital reference unit (28 in all). Function generation is performed using DCU's updated from the digital computer, with special dual comparators used to provide the necessary interrupts for DCU updating.

Figure 3 shows the circuit schematic within a given AD/FOUR field for the summer-integrator and DCU switch matrix. Each bipolar output of the summer-integrators in the field is permanently patched to a pair of DCU's, utilizing $16\times2=32$ DCU's (amplifiers 200, 201, etc., in Figure 3). Each output of an additional 8 bipolar amplifiers (100, 101, etc., in Figure 3) is patched permanently to four DCU's, utilizing $8\times4=32$ DCU's. The input to each of these 8 amplifiers can be switched to any of the 16 summer-integrator outputs. Thus all summer integrators drive a minimum of 2 DCU's, and can be programmed additionally to drive a total of 6, 10, 14, etc., DCU's.

The output of each of the 64 DCU's in the AD/FOUR in Figure 3 can be programmed to any of the 16 summer-integrator input summing junctions, or any of the 12 multiplier Y-input buffer-amplifier summing junctions (see Figure 4). Thus a 64×28 summing-junction switch matrix for DCU outputs is represented by the circuit in Figure 3, as well as a 16×8 switch matrix for the DCU buffer amplifier inputs. Each DCU is the standard 15 bit AD/FOUR single-buffered MDAC, with the most significant bit providing a negative analog polarity. Thus a given DCU can be programmed between a gain of -1.6384 and $+1.6383$ using two's complement logic and with a minimum bit size of 0.0001. Since the DCU is a summing-junction

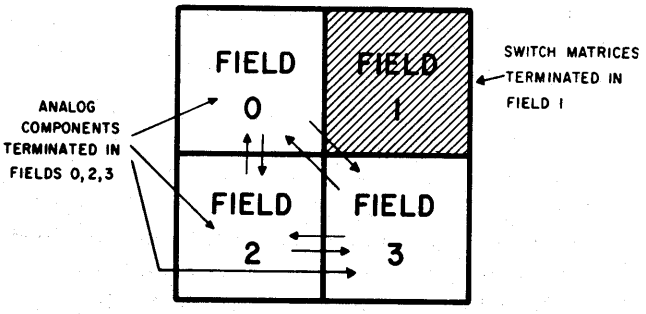


Figure 2—Analog patchboard

output device, summing-junction switching can be used throughout the switch matrix.

A digital reference unit (DRU) is permanently patched to each of the 16 summer-integrator amplifiers, allowing a programmable bias. Also, the output V_{IC}, of an initial-condition amplifier is patched to all of the 16 integrator-summer input terminals. This allows initial conditions for the entire field to be set sequentially using the single DRU which drives the initial condition amplifier. The procedure starts with all integrators returned to the IC (Initial Condition) mode with the DRU set at zero (i.e., $V_{IC}=0$). This puts zero initial conditions on all integrators, which are then all switched to Hold. Following this, each integrator is switched to IC sequentially with the DRU programmed to yield the desired initial condition for that particular integrator. The integrator is then returned to Hold and the next integrator switched to IC, etc. After all initial conditions have been established in this manner, the integrators can all be switched to Operate.

Figure 4 shows the schematic for the connections to the 12 multipliers within a given AD/FOUR field. The bipolar X inputs for each multiplier are permanently wired to specific summer-integrator outputs. The bipolar Y inputs are each programmed to the outputs of the 12 bipolar summers in the AD/FOUR field. The summing junctions of these amplifiers are in turn

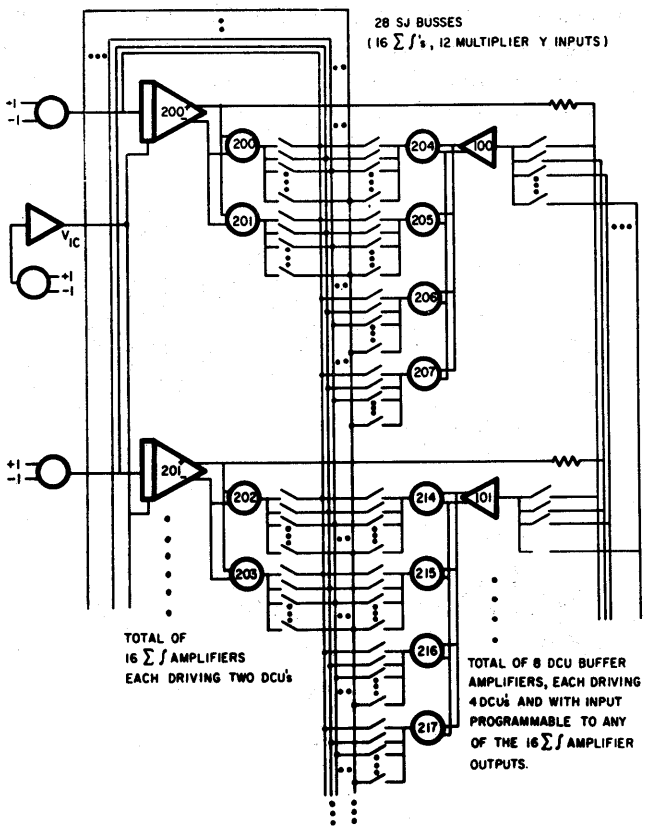


Figure 3—Circuit schematic for DCU switch matrix

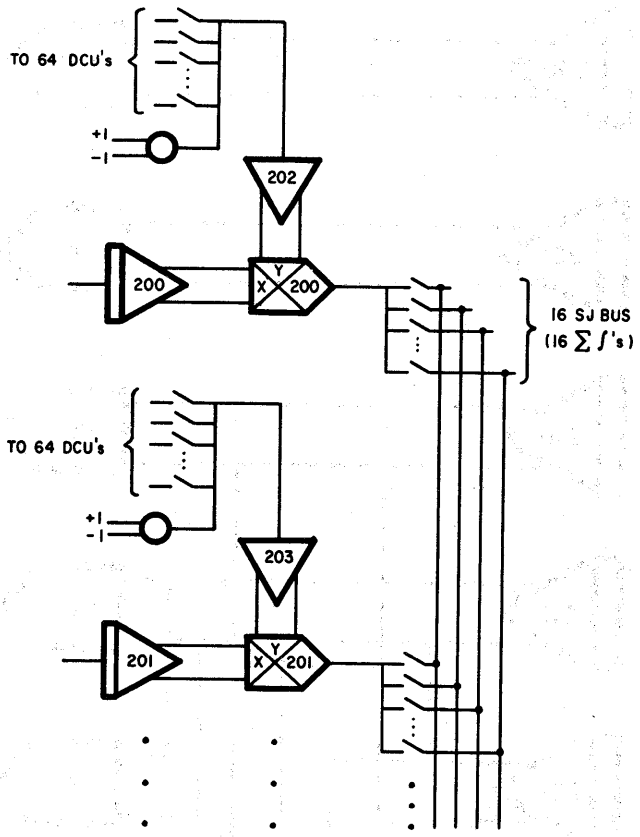


Figure 4—Circuit schematic showing multiplier switch matrix

switchable to the outputs of any of the 64 DCU's in the field (see Figure 3). In addition, there is a DRU permanently patched to the input of each of the 12 Y input buffer amplifiers, allowing a programmable bias into each multiplier Y input.

Extensive study of typical analog programs has shown that the X inputs to multipliers are seldom all independent. For this reason the multiplier X inputs are assigned to summer-integrator outputs using the configuration shown in Figure 5. The first four multipliers are assigned, respectively, to the first four summer-integrators. The fifth multiplier (no. 221) has its X input assigned to the fourth amplifier (no. 211) and the sixth multiplier (no. 222) has its bipolar X input patched to the common terminal of a 3-position, double pole relay (nos. 220, 221). The position of this relay, part of the standard AD/FOUR equipment complement, is controlled by registers set from the digital computer. In this way the X input to multiplier 222 can be programmed to amplifier 210, 211, or 222. Thus the configuration of multiplier X inputs assigned, respectively, to summer integrators in half a field can be programmed to be 1, 1, 1, 2, 1; 1, 1, 1, 3; or 1, 1, 2, 2. Later examples show the utility of this scheme.

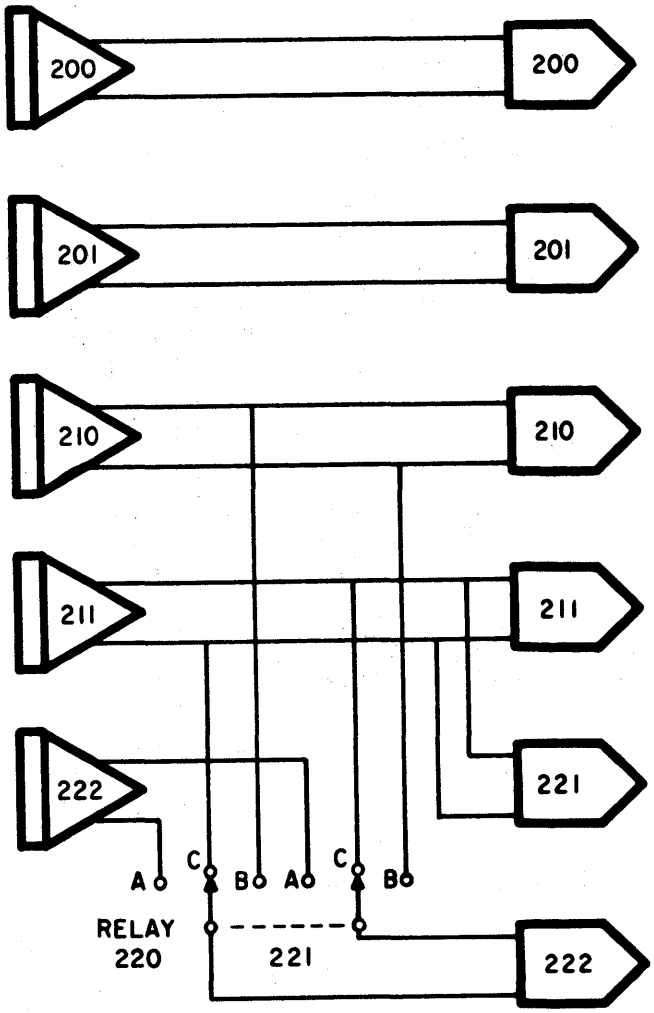


Figure 5—Schematic showing assignment of amplifier outputs to multiplier X inputs in one-half of field 2

Four hard feedback-limiters are permanently programmed, respectively, around four of the 16 summer integrators in each field in the automatic-patching system. DRU (digital reference unit) outputs are permanently programmed in pairs to each hard limiter to allow digitally controlled setup of the + and − limits. In the AD/FOUR computer each summer-integrator is normally in the summer configuration. By grounding, respectively, either one of two patchboard holes associated with each summer integrator, the amplifier can be changed to an integrator or high-gain configuration. In the automatic patching system these amplifier configuration holes in each field are patched to corresponding holes in field 1. These field 1 holes, under register control from the digital computer, provide program-controlled electronic grounds. Thus any of the 16 summer-integrators in each field can be configured as summers, integrators, or high-gain amplifiers.

FUNCTION GENERATION

Generation of precision functions of one or more variables on analog computers has always presented major difficulties. In fact, one of the principal tasks of the digital subsystem in many hybrid problems has been the generation of multivariable functions for use by the analog subsystem. However, this approach has serious dynamic limitations. Since it is desirable to have the automatically-patched analog computer operate at relatively high speed in order to permit time sharing, pure digital function generation is undesirable. Instead, a hybrid method analogous to that first proposed by Rubin[4] is utilized.

The graph in Figure 6 illustrates the function-generation method when applied to a function of one variable. Between breakpoints x_i and x_{i+1} the function $f(x)$ is represented as having intercept a_i and slope b_i, i.e., $f(x)=a_i+b_ix$. The mechanization is achieved by terminating two DCU's in amplifier 1, as shown in the figure. Whenever x crosses over into a new zone, e.g., between x_{i+1} and x_{i+2}, the two DCU's are updated to represent a_{i+1} and b_{i+1}, respectively, the intercept and slope in the new zone. High speed detection of the zone in which x is located is achieved by a special dual comparator with digitally-programmed bias inputs x_i and x_{i+1}. Whenever x passes outside the zone bounded by x_i and x_{i+1}, the gate shown in Figure 6 throws the analog system into Hold. By sensing the state of comparators A and B the digital computer determines whether x now lies in the $i-1$ or $i+1$ zone. It then looks

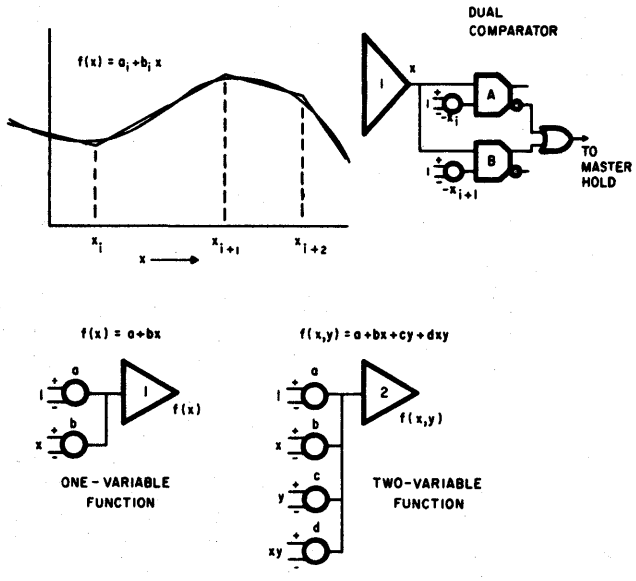


Figure 6—Circuit schematic for function generation

up the values for intercept a and slope b in the new zone and sets the corresponding DCU's terminated in amplifier 1 to the new values. It also sets the bias inputs into comparators A and B to the corresponding values for the new zone. After completing these operations the digital computer restores the analog system to the Operate mode.

Because the analog computer is in the Hold mode while the digital computer is accomplishing the necessary DCU and DRU updatings, any dynamic errors which would otherwise result from the time required by the digital computer to accomplish the updatings are eliminated. The only significant sources of dynamic error include the lag in dual comparator response (the order of one microsecond), the delays in Hold mode control switches (the order of one microsecond), and differential timing errors in activation of both Hold and Operate mode control (less than one microsecond). Also, offsets caused by cycling the integrators back and forth between Hold and Operate must be considered. In the AD/FOUR each such mode cycle causes the order of 0.5 millivolts of additional integrator output offset for integrators programmed with a gain of 1000 (typical for high-speed problem solution). At foreseeable Hold-Operate cycle rates in implementing the function generation as described here, the equivalent steady offset represents only from 0.01 percent to 0.1 percent of full scale. Even with these offsets there will be no significant effect except where open-ended integrations are involved. Study has shown that integrator drift during Hold is completely negligible over the time required for updating DCU's and DRU's.

Generation of functions of two variables is implemented using the formula $f(x, y) = a+bx+cy+dxy$. The circuit is shown in Figure 6 using 4 DCU's terminated in amplifier 2. The DCU settings correspond to the function $f(x, y)$ for $x_i \leq x \leq x_{i+1}$, $y_i \leq y \leq y_{i+1}$. When either x or y moves into a new zone, as detected by the respective dual comparator, the 4 DCU's are updated while the analog computer is in Hold. The resulting $f(x, y)$ analog function is equivalent to linear interpolation in x and y. A function of three variables can be generated using the formula $f(x, y, z) = a+bx+cy+dz+exy+fxz+gyz+hxyz$. As before, the 8 DCU settings needed to generate the three-variable function correspond to the appropriate x, y, and z zones.

Since each summer-integrator in the automatic-patching system can terminate any number of DCU's and has a permanently assigned DRU (digital reference unit), each summer integrator can be used to generate any multivariable function or any sum of multivariable functions. Reference to Figure 4 shows that each Y-input bipolar buffer amplifier for multipliers terminates any number of DCU's as well as an assigned DRU. Thus it can be used to generate the sum of multivariable functions, which in turn are multiplied by the other multiplier input, X. In the AD/FOUR automatic patching system 8 dual comparators, each with an assigned DRU pair, as shown in Figure 6, are available in each analog field. Inputs for these 8 dual comparators can be assigned to any of the 16 summer integrators in the field using switches driven by digitally-set registers. Fixed function generators, e.g., $\sin x$, $\cos x$, and $\log x$ function generators, can be terminated in multiplier locations in the AD/FOUR. In this case the general I/O format shown in Figure 4 for multipliers is preserved. For example, the Y-input bipolar buffer amplifier is used to terminate the $\sin x$ and $\cos x$ fixed *dfg* (still with summing-junction output and hence output switches).

INTERFIELD PATCHING

Up to now we have only described the circuit configuration needed to patch automatically the connections within an AD/FOUR field. It is, of course, also necessary to implement interconnections between each of the fields (three-fields maximum; see Figure 2). This is accomplished in the following two ways:

I. The first of the two DCU's permanently assigned to the output of each summer integrator (e.g., DCU 200, 202, etc., in Figure 3) has its output programmable to the summing-junction input of any summer integrator in the other two fields as well as summing junctions in its own field.
II. The input to each quad DCU buffer amplifier can be switched to any summer integrator amplifier output in the other two fields as well as in its own field (see Figure 3).

The effectiveness of the above automatic interfield patching capability is best appreciated by studying example problems. Extension of II above to amplifier outputs trunked in from adjacent consoles provides effective interconsole trunking capability.

DIAGNOSTICS

Diagnostics, both to determine proper component functioning and to debug analog programs, is appreciably simpler with the automatic-patching system than with a conventional analog patchboard program. First of all, the patchboard on which the automatic patching program is wired also serves as the diagnostic patch board, i.e., no special diagnostic patchboard is

required. Secondly, because of the fixed configuration, the complete computer control of all component modes and interconnections, the presence of programmable initial conditions and bias inputs for every bipolar amplifier, etc., it is extremely straightforward to write the software for checking every analog amplifier, DCU, DRU, and nonlinear element as well as every matrix switch. For example, proper functioning of every DCU and DCU output switch can be implemented by setting a one machine-unit initial condition on the integrator driving each DCU, with all other summer-integrators programmed as summers. Each bit of the DCU can be checked individually by programming the DCU back to the driving-amplifier input and monitoring that amplifier input. Next the DCU can be set at, say, unity gain and its output switched successively to every amplifier summing junction (both summer-integrators and multiplier Y-input buffers). Proper matrix-switch functioning is assured by checking the respective amplifier outputs. Equally straightforward checks can be implemented for other components, including a rate test for all integrators using the programmable bias (DRU) input. The thousands of individual checks making up an overall three-field diagnostic will take only seconds, with easily-identified printout of any malfunctions.

A similar scheme is proposed for program verification, where, successively, a given initial condition (usually one machine unit) is applied to every integrator with the output (or summing-junction input) to all amplifiers measured and stored. This allows checking of the setting and input assignment of every DCU, as well as its output assignment. By programming unit X and Y inputs to each multiplier, successively, and monitoring all summer outputs and integrator inputs, multiplier output assignments can be checked. Function generator set-up can be checked by observing amplifier outputs for successively programmed values of the input variable. It is believed that this type of program verification is even more powerful and easily debugged than the more conventional static check.

TIME SHARED OPERATION WITH TERMINALS

It is estimated that complete setup time for all component configurations, switch-matrix registers, and DCU and DRU settings will be approximately 10 milliseconds for the AD/FOUR automatic-programming system. With integrators programmed at nominal gain settings of 1000, this implies solution times of perhaps 10 to 100 milliseconds for typical systems on nonlinear differential equations. Such rapid program turn-around time, in turn, suggests that it should be quite feasible and extremely cost effective to time-share a single AD/FOUR hybrid system among a number of users stationed at remote terminals.

A relatively simple terminal system suitable for time-sharing is shown in Figure 7. This is the AD Dynamics Terminal, originally developed primarily for the educational market in order to allow individual terminal operators to time share a single problem programmed on the AD/FOUR or AD/FIVE hybrid computer. Across the top of the front panel of the terminal are eight 3-digit-plus-sign parameter entry thumbwheel switch sets which are assigned, respectively, to 8 problem parameters. To the right are 8 pushbuttons which control singly or in combination logic signals on the hybrid system which in turn control problem configuration. On the lower panel are channel selector, gain, and bias controls for the x and y axes of the memory scope used for solution readout. Also on the lower panel of the terminal are the computer mode-control switches.

A number of terminals (up to 16 or more) can be connected to a single AD/FOUR hybrid computer. The computer interrogates each terminal in sequence to see whether the operator has requested a solution since the last interrogation of his terminal. If he has, the computer sets his parameter values and proceeds to generate a solution, which is stored on the memory scope and takes, perhaps, 50 milliseconds. If the operator has not requested a solution, the computer wastes no time and skips to the next terminal for interrogation. Under these conditions each terminal operator usually receives a response to his solution request in a fraction of a second, and can obtain and display multiple solutions for different parameter settings about as fast as

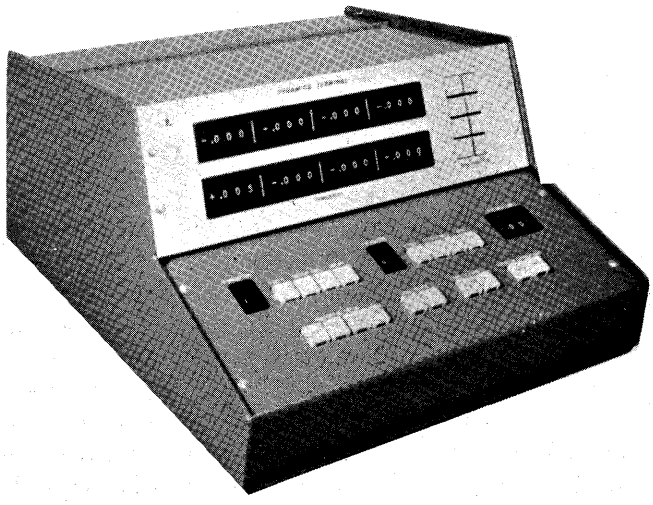


Figure 7—Dynamics terminal

he can reset the parameter-entry wheels and push the solution button.

When operating with the AD/FOUR automatic-programming system, the Dynamics Terminal will be used to call up various programs using the 8 logic pushbutton switches (256 combinations). Several of the pushbuttons can be used to assign the 8 parameter inputs to different groups of problem parameters. If a given terminal operator calls for a solution, his problem is programmed on the AD/FOUR computer upon interrogation of his terminal. If the problem has been stored in core (roughly 500 words required for a typical large problem), then the program setup takes only about 10 milliseconds. The net result is an access time essentially comparable to that now enjoyed with the Dynamics Terminal System, except that each user receives his own problem when he obtains control of the computer.

When the relatively simple Dynamics Terminals as described above are used for time sharing, initial setup and debugging of each problem must be done using the conventional hybrid I—O and not through the terminal. Obviously, it would be advantageous to have a more sophisticated terminal which also allows problem setup and debugging, in addition to problem running. This more sophisticated terminal will probably require a keyboard, alphanumeric display, and perhaps even a mini-digital computer. In any case, if problem setup and debugging is to be achieved through terminals while time sharing the hybrid computer with other terminals, a very extensive and powerful software system for time sharing will have to be available for the digital subsystem.

A SIMPLE EXAMPLE

As a simple example for the programming of a nonlinear differential equation, consider the Vander Pol equation:

$$\ddot{x}=\mu(1-x^2)\dot{x}-x \qquad (1)$$

A circuit for solving this equation using the AD/FOUR automatic-patching setup is shown in Figure 8. Since the solution is known to approach a limit cycle of amplitude 2, we have indicated that $x/2$ is computed. Thus $x=2$ corresponds to $x/2=1$ (one machine unit) at the output of integrator 201. Since DCU's can be set to gains between -1.6384 and $+1.6383$, the value of the parameter μ, as set on DCU 212, can range up to $1.6383/4\cong 0.4096$. Although the gain (time scale) of the two integrators is under digital program control, integrator gain constants of 1000 would normally be used for a high speed solution. Under these conditions the resulting oscillation would have a period of approximately 6 milliseconds, which means that for five cycles to be displayed the solution would take about 30 milliseconds. The problem can be rescaled to allow higher values of μ by simply reducing each integrator gain by the same factor. For example, if the gain of each integrator is reduced by a factor of 4, DCU's 200, 203, and 212 would be reset to 0.25, -0.25, and μ, respectively. Now μ can range up to 1.6383, and the computer solution is one-fourth as fast as before. Thus about 120 milliseconds are required for some five cycles of the solution. Or by programming the basic integrator time scales to $\times 10{,}000$ instead of $\times 1000$, about 12 milliseconds is required for five cycles of the solution.

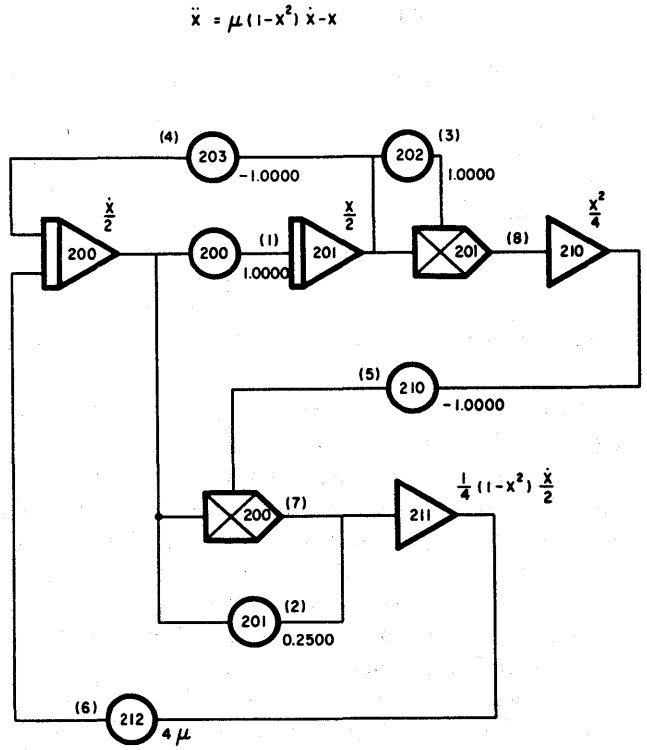


Figure 8—Automatic patching circuit for Vanderpol's equation

It should be noted that the address and data format currently used for setting DCU's in the AD/FOUR is used for setting the switch registers needed to program the connections shown in Figure 8. The address indicates the device to which the switch common is connected (i.e., the signal source). The data word indicates the component to which the device is connected (i.e., the signal destination). A simple table stored in the digital computer transforms each such statement pair to a pseudo DCU address and data word, and the switch register setting is implemented with standard HCR's (hybrid communication routines) as if it were

setting a DCU. Thus the following list would be required to implement the switch settings in Figure 8:

Setting No.	Source	Destination	
(1) DCU	200	AMP	201
(2) DCU	201	AMP	211
(3) DCU	202	MULT	201
(4) DCU	203	AMP	200
(5) DCU	210	MULT	200
(6) DCU	212	AMP	200
(7) MULT	200	AMP	211
(8) MULT	201	AMP	210

For clarity the switch setting numbers (1) thru (8) are shown in Figure 8 to allow correlation between the above table and the settings. Actually, in implementing the switch settings the digital computer merely thinks it is addressing and setting DCU's, so that existing HCR's (hybrid communication routines) can be used. Although the example in this section is very simple, it illustrates the implementation of the AD/FOUR automatic patching scheme for solving a nonlinear differential equation. The actual switching configuration described in this paper evolved from considering the program for much larger problems, e.g., nonlinear partial differential equations, six-degree-of-freedom flight equations, complex control systems, etc.

AN EXAMPLE SOLUTION FOR NONLINEAR FLIGHT EQUATIONS

As a second example, consider the solution of the six-degree-of-freedom flight equations. The automatic patching program for these equations requires approximately three fields of an AD/Four, the exact number of components depending on the complexity of the aerodynamic functions. One field of the program is illustrated in Figure 9, where the program for the translational and rotational equations is illustrated. The following equations result from the summation of forces along the flight-path axes and moments about the body axes:[5]

$$\dot{v}_p = \frac{X_w}{mg}\frac{g}{V_{\max}} \tag{2}$$

$$\dot{\alpha} = Q_b - P_b^s\beta + \frac{Z_w}{mg^v p}\frac{g}{V_{\max}} \tag{3}$$

$$\dot{\beta} = \frac{Y_w}{mg^v p}\frac{g}{V_{\max}} - R_b^s \tag{4}$$

$$\dot{P}_b = \frac{I_{xz}}{I_{xx}}\dot{R}_b + \frac{L_b}{I_{xx}} \tag{5}$$

$$\dot{Q}_b = \frac{I_{zz}-I_{xx}}{I_{yy}} R_b P_b + \frac{M_b}{I_{yy}} \tag{6}$$

$$\dot{R}_b = \frac{I_{xx}-I_{yy}}{I_{zz}} P_b Q_b + \frac{I_{xz}}{I_{zz}}\dot{P}_b + \frac{N_b}{I_{zz}} \tag{7}$$

Here v_p is dimensionless total velocity; α and β are angles of attack and sideslip, respectively; P_b, Q_b, and R_b are body-axis angular-rate components along the respective, x, y, and z body axes; P_b^s and R_b^s are roll and yaw rates along stability axes; X_w, Y_w, and Z_w are external forces along the respective flight-path axes; L_b, M_b, and N_b are external moments along the respective body axes.

In writing Eq. (3) we have assumed $\beta \ll 1$, and in writing Eqs. (5), (6), and (7) we have omitted nonlinear terms which are negligible in effect.[5]

The external force components X_w, Y_w, and Z_w along the flight-path axes are derived from force components X_s, Y_s, and Z_s along the stability axes by resolution through the sideslip angle β. Thus

$$X_w = X_s\cos\beta + Y_s\sin\beta \tag{8}$$

$$Y_w = -X_s\sin\beta + Y_s\cos\beta \tag{9}$$

$$Z_w = Z_s \tag{10}$$

The force components along the stability axes are obtained from body-axis force components X_b, Y_b, Z_b, respectively, by resolution through the angle of attack α. Thus

$$X_s = X_b\cos\alpha + Z_b\sin\alpha \tag{11}$$

$$Y_s = Y_b \tag{12}$$

$$Z_s = -X_b\sin\alpha + Z_b\cos\alpha \tag{13}$$

Finally, the body-axis force components include gravity, propulsive, and aerodynamic terms, as shown in the following equations:

$$\frac{X_b}{mg} = -\sin\theta + \frac{X_p}{mg} - \frac{qS}{mg}C_D(M,\alpha) \tag{14}$$

$$\frac{Y_b}{mg} = \cos\theta\sin\phi + \frac{qS}{mg}C_Y(M,\beta) \tag{15}$$

$$\frac{Z_b}{mg} = \cos\theta\cos\phi - \frac{qS}{mg}C_L(M,\alpha) \tag{16}$$

where θ and ϕ are conventional pitch and bank angles respectively, q is the dynamic pressure, and S is the wing area. Also contained in Eqs. (14), (15), and (16) are three aerodynamic functions of two variables each involving Mach number M along with α and β.

The moments L_b, M_b, and N_b in Eqs. (5), (6), and

(7) are obtained from the following equations:

$$L_b = qSbC_l \tag{17}$$

$$M_b = qScC_m \tag{18}$$

$$N_b = qSbC_n \tag{19}$$

where b is the semispan, c is the characteristic wing chord, and c_l, c_m, and c_n are, respectively, the aerodynamic coefficients for roll, pitch, and yaw moments about the body axes.

The circuit for implementing the solution of Eqs. (2)—(19) using automatic patching is shown in Figure 9. Components are numbered on the basis that the program is implemented in field 2 of the AD/Four. Fields 0 and 3 are used to solve the remainder of the six-degree-of-freedom flight equations. Shown in Figure 9 is the circuit for automatic patching of two resolvers in a field (resolver 670 and 671 in the figure).

The resolver input angles, respectively, are driven by amplifiers 230 and 231. X and Y inputs to resolver 670 are buffered with bipolar summing amplifiers 202 and 203, respectively. The Y inputs of multipliers 200 and 201, normally driven by these amplifiers, are instead both driven by quad DCU amplifier 120. Similarly, the X and Y inputs to resolver 671 are buffered with bipolar summing amplifiers 242 and 243, respectively, while the Y inputs to multipliers 240 and 241 are driven by quad DCU amplifier 121. The outputs X' and Y' of resolver 670 are terminated in DCU pairs numbered 246, 247, and 256, 257, respectively. Similarly the outputs of resolver 671 are terminated in DCU pairs 266, 267, and 276, 277, respectively. Each of these DCU pairs lowers the numbers of DCU's driven by the corresponding quad DCU amplifiers from 4 to 2 DCU's.

Note in Figure 9 that there are 3 signals from adjacent fields terminated in quad DCU amplifiers (numbers 122, 123, and 130). Note also that there are 6 signals from an adjacent field originating in DCU's in that field (numbers 100, 102, 110, 112, 120, and 122). These are terminated in field 2 as currents into summing junctions. This illustrates the typical interfield trunking capability of the AD/Four automatic patching system.

The circuit in Figure 9 includes three 2-variable function generators and illustrates the power and flexibility of the automatic patching system. For example, amplifier 200 along with multiplier 251 implements Eq. (14) in its entirety, i.e., generates a function of two variables, multiplies the function by a third variable, and sums the result with two additional terms.

Altogether the circuit in Figure 9 utilizes 15 out of the 16 summer integrators in field 2, all 12 multipliers, both resolvers, and 5 out of the 8 quad DCU amplifiers.

Following the format given in the previous example, we have the following list required to implement the autopatch settings in Figure 9:

Setting No.	Source		Destination	
(1)	DCU	204	MULT	251
(2)	AMP	300	AMP	122
(3)	DCU	224	MULT	251
(4)	AMP	301	AMP	123
(5)	DCU	234	MULT	251
(6)	MULT	251	AMP	200
(7)	DCU	102	AMP	200
(8)	DCU	201	REX	670
(9)	DCU	256	AMP	240
•	•		•	
•	•		•	
•	•		•	
	etc.			

DCU settings, including those of reference DCU's such as number 600, are implemented in the usual manner, using the DCU address and 15 bit data word.

SUMMARY

There is a trade-off between hardware and software in any automatic patching system. By choosing a system of only three large modules of sixteen integrators each we have enormously simplified the setup procedure, allowing a direct programming technique without the need for any fancy interpreter, i.e., standard HCR's are utilized to control the interconnections. Indeed the setup procedure is so direct that it is felt that there is no additional training burden in implementing this system.

The number of switches in the system is modest, according to the following list:

Switches per Field:

64 DCU's to 28 S.J.'s	1792
12 MULT S.J.'s to 16 S.J.'s	192
8 QUAD DCU AMPL to 16 Outputs	128
8 DUAL COMP to 9 AMPL	72
	2184

Interconnection between Two Fields:

(16 DCU's×16)×2	512
(8 QUAD DCU AMPL to 16 Outputs)×2	256
Total Switches	768

One Field System	
(16 Integrators)	2184
Two Field System	
(32 Integrators)	2184
	2184
	768
	5136
Three Field System	
(48 Integrators)	2184
	2184
	2184
	768
	768
	768
	8856

These numbers compare very favorably with those of other systems previously cited. Reference (3) points out the advantage of a current-patched computer in an automatic patching system. However, the profound advantage of implementing such patching in a computer equipped with only digital coefficient devices and controlling the interconnections ("patching") with the same software which sets the coefficient devices has been overlooked.

It can be seen from the switch quantities that the system will add about 20 percent to the cost of the hybrid system, a figure which can be substantially recovered in the saving in patching hardware alone.

The system is flexible. Since it is implemented on a patchboard, the patchboard can be modified easily. Moreover, this patchboard can be removed and the computer can be used in the traditional way. Also large numbers of consoles can be easily and economically retro-fitted with this system.

Finally, the system uses modern switches, e.g., electronic. This will allow true hybrid time sharing providing the rather large software investment is made. Even without such an investment the system will provide time sharing of the analog console with complete re-patching between runs.

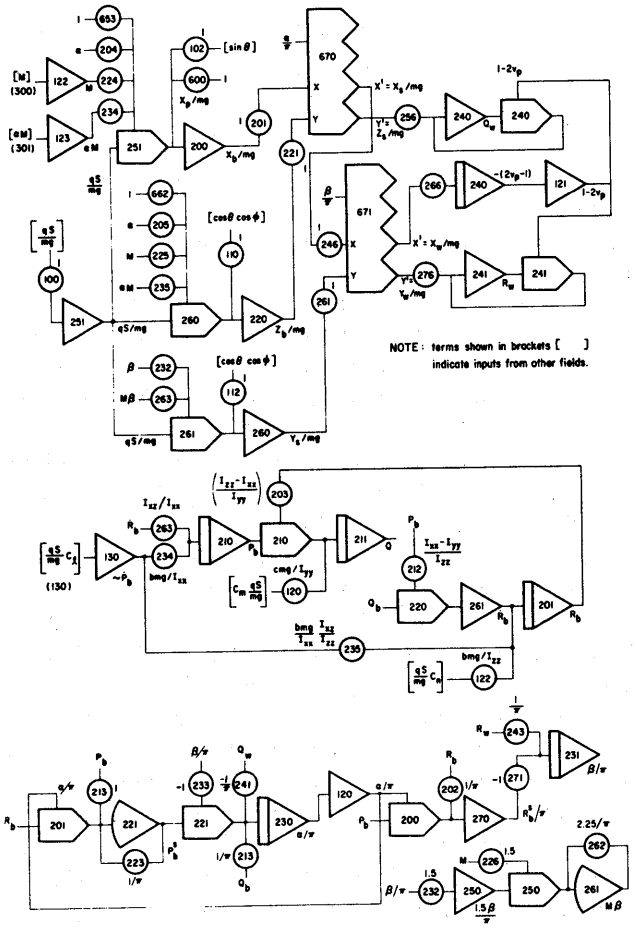


Figure 9—Field 2 automatic patching circuit for translational and rotational flight equations

REFERENCES

1 D A STARR J J JOHNSON
The design of an automatic patching system
Simulation June 1968

2 G HANNAUER
Automatic patching for analog and hybrid computers
Simulation May 1969

3 T J GRACON J C STRAUSS
A decision procedure for selecting among proposed automatic analog computer patching systems
Simulation September 1969

4 A I RUBIN
Hybrid techniques for generation of arbitrary functions
Simulation December 1966

5 L E FOGARTY R M HOWE
Computer mechanization of six-degree-of-freedom flight equations
Simulation October 1968

Digital voice processing with a wave function representation of speech*

by JOHN D. MARKEL and BERNARD CAREY

University of California
Santa Barbara, California

INTRODUCTION

Digital voice processing has advanced to a relatively high level of sophistication due to the development of the fast Fourier transform (FFT) algorithm.[1,2] Complete vocoder systems have been developed around the FFT[3] and with the advent of high-speed integrated circuits and read-only memories to implement sine and cosine tables for the FFT, actual real-time hardware processing has been accomplished.[4]

Anyone familiar with analog vocoder systems is aware of the somewhat unnatural sounding synthetic speech output. There are two major reasons for this: first, pitch must be extracted and reinserted quite accurately, and secondly, the consonant sounds must be accurately represented because of their relatively short duration but yet important perceptual quality. The problem of accurate pitch extraction performed by a fundamental frequency tracking filter in the analog systems is effectively eliminated in the digital vododers by the application of ceptral techniques in conjunction with the FFT.[5] However, the difficulty with generating natural sounding consonants is still present which is an inherent difficulty due to the frequency domain method of generating the transient sounds.

In this paper we will describe a basically new approach to digital voice processing which will be called a "Wave Function Analysis and Synthesis System," due to the appearance of the basis elements (or functions) which are used in the representation of the speech. It is basically a time domain system as opposed to the standard vocoder which performs the representation mainly in the frequency domain. With this system we will demonstrate the capability of quite accurate and natural sounding generation of synthetic speech.

* This work was supported by Advanced Research Projects Agency, Contract AF 19(628)-6004.

Development of a wave function approach

Several years ago Dr. Glen J. Culler of UC-Santa Barbara observed that many sounds of speech, when properly filtered, could be represented by derivatives of the Gaussian function $\exp(-t^2/2)$.[6] It was discovered that the nth derivative could be described explicitly by a Gaussian function multiplied by the Hermite polynomial of degree n. These functions are known to satisfy the differential equation[7]

$$\ddot{u}(t)+at\ \dot{u}(t)+b\ u(t)=0 \quad u(0)=c_1 \quad \dot{u}(0)=c_2 \tag{1}$$

with the particular form

$$\ddot{g}_n(t)+t\ \dot{g}_n(t)+(n+1)g_n(t)=0 \tag{2}$$

where n is a nonnegative integer and $g_n(t)$ is the nth derivative of the normalized Gaussian function. From this point the equation was generalized to allow for noninteger coefficients, expansion or contraction of the solutions, time shifts and phase shifts. The general form is[6]

$$\ddot{u}(t)+(2\pi/S)^2(t-C)\dot{u}(t)+(2\pi/S)^2(N^2-\tfrac{1}{2})u(t)=0 \tag{3}$$

with initial conditions given by

$$u(C)=A\cos\phi$$
$$\dot{u}(C)=A\,(2\pi N/S)\sin\phi$$

The basic hypothesis behind the use of this differential equation for analysis of speech is that properly filtered speech can be represented by members of the wave function family described from its solution. The parameters of this equation have been chosen so that physically meaningful relationships are being used. Figure 1 illustrates a representative solution to equation 3. The rationale behind the use of these parameters is that A corresponds to the peak of the envelope which encloses the wave function, S is a spread factor within

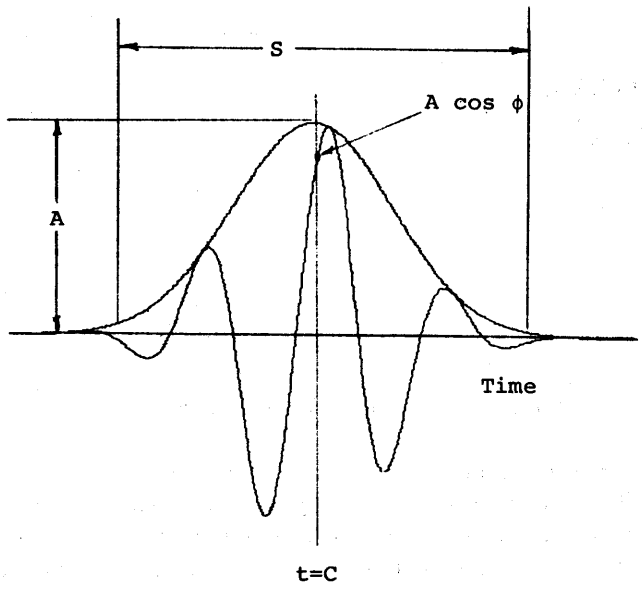


Figure 1—Representative wave function showing parameters (N is approximately the number of half cycles under the envelope)

whose value nearly all of the energy of the solution lies, C is the center of the envelope of the wave function, ϕ is the effective phase of the solution with respect to C, and N is roughly the number of half cycles that would be measured under the envelope function. Using equation 3 as a reference, the major part of the analysis is based upon development of algorithms for automatically extracting the five parameters mnemonically referred to as the ASCON parameters from the speech wave.

Our motivation for simulating an automatic voice processing system based upon the wave function approach is based upon two main factors: (1) Using the UCSB speech on-line station developed by Culler it has been demonstrated that high-quality synthetic speech can be generated from the wave function parameters extracted from real speech, and (2) an asymptotic solution to equation 3 has been obtained which is of closed form and yet is valid with respect to it over nearly the whole range of analysis, thus considerably simplifying the algorithmic difficulties in transforming speech into wave function parameters.

One of the major problems in the wave function analysis-synthesis method is that of performing the mapping from the speech wave to the parameters of the differential equation (or wave function solution). Since the desired goal is to make a "best" fit to the speech wave in some sense with a known basis set, the criteria for "best" will determine the algorithm. In this paper we will describe a procedure for extracting the desired wave functions based upon a four-point analysis around each major sweep (to be defined shortly). First, a brief discussion of the basis set chosen for the analysis and synthesis will be presented and then the algorithms pertaining to the automatic voice processing system will be considered.

The Gaussian cosine modulation (GCM) wavefunction formulation

The basis elements we will use in the algorithm are generated as asymptotic solutions to equation 3. Through the use of an asymptotic form of the confluent hypergeometric function (which is required for expressing the exact time domain solution to equation 3), the surprisingly simple result[8]

$$u_a(t) = A \exp[a(t-C)^2/4] \cos[w_0(t-C) - \phi] \quad (4)$$

is obtained where $u_a(t)$ is the asymptotic solution,

$$a = (2\pi/S)^2 \quad \text{and} \quad w_0 = (2\pi/S)^2(N^2-1)^{1/2}.$$

The solution is thus a Gaussian probability function with a mean of C (sec.) and standard deviation of $(2/a)^{1/2}$ (sec.) modulated by a cosine wave of peak amplitude A with center frequency w_0 (rad/sec.).

This is a rather concise result to be derived from such forbidding sounding names as confluent hypergeometric functions. However, if the requirement for validity of the asymptotic solution ($2N^2-\frac{1}{2}$ is "large") is unrealistic with respect to physical speech, then the formulation is of limited value. It has been experimentally demonstrated that these asymptotic solutions match the solutions generated recursively from equation 3 over nearly the whole range of speech. The two representations have the most error for small N. It has been observed that the smallest N values (around 2-2.5) occur in the (100-400) Hz segment of the speech range of interest. Figure 2 shows the asymptotic solutions and the recursive solutions for $N=2$ and $N=3.6$. For N greater than 4 the two solutions are indistinguishable. Therefore it is proposed that a more meaningful differential equation to use as the mathematical foundation from which the wave function family $w(t) = u_a(t)$ is derived is the one which has the solutions in equation 4 as exact and not asymptotic solutions. By differentiation of equation 4 and substitution into equation 5, one can verify that the GCM differential equation is

$$\ddot{w}(t) + a(t-C)\dot{w}(t) + \{w_0^2 + a/2 + [a^2(t-C)^2/4]\}w(t) = 0 \quad (5)$$

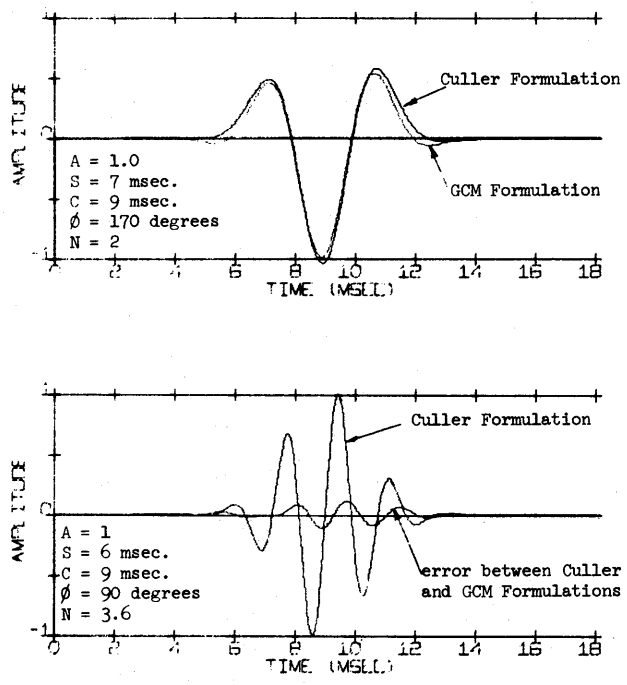


Figure 2—Comparison of recursive solutions to Culler formulation and exact solution to GCM formulation

with initial conditions specified by

$$w(C) = A\cos\phi$$

$$\dot{w}(C) = A\,w_0\sin\phi.$$

THE GCM ANALYSIS/SYNTHESIS SYSTEM

There are five basic processes involved in the proposed wave function analysis algorithm.

1. Preprocessing—The speech is filtered into appropriate frequency regions. Each of the filtered regions in the time domain is denoted as a "substring."
2. Major Sweep Detection—The substring is searched for a location in which a wave function behavior can be isolated.
3. Gaussian Envelope Representation—Based upon the extrema and corresponding time values around the major sweep, a Gaussian envelope is calculated which fits the data with some predefined error criteria.
4. Cosine Wave Representation—Based upon extrema near the peak of the calculated envelope a frequency and phase term are generated which then uniquely describes the wave function.
5. Residue Calculation—After the envelope and cosine wave parameters have been extracted, the corresponding wave function is synthesized and subtracted from the filtered speech.

Figure 3 shows the proposed block diagram of the analysis system. The preprocessor performs the digital filtering of the signal into sub-strings covering the (100,3000) Hz range. The major sweep detector performs the task of isolating a region around which a wave function will be fit. Calculation of A, S, and C corresponds to fitting an envelope around the major sweep which satisfies some predefined error criteria. Calculation of Phase ϕ and Frequency F corresponds to fitting a cosine wave to the speech which matches closely in the region of the major sweep.

Once the parameters have been generated, the corresponding wave function is generated over the region $|t-C| < S/2$. By subtracting this function from the incoming channel signal, the error or residue is generated which can then be represented by the next wave function. Conceptually, the representation process is carried on independently for each substring (as denoted by the bold arrows) implying a multiple input-output configuration. The process of fitting a single wave function to local segments of the substrings is repeated over and over until the total time interval of interest has been analyzed. Each of these processes will now be considered in somewhat more detail.

Preprocessing

The purpose of the preprocessor is to transform the acoustic waveform denoted as a "string" into "substrings" that are amenable to wave function analysis. It has been demonstrated elsewhere[9] that a sufficient condition for obtaining high-quality wave function representation is that the string be filtered into four substrings that cover the frequency range (100,3000) Hz. Let $s(t)$ be the original string with frequency components in the range (100,3000) Hz. Then

$$s(t) = \sum_{n=1}^{4} s_n(t)$$

where $s_n(t)$ is the nth substring. In the frequency domain

$$S(jw) = \sum_{n=1}^{4} S_n(jw)$$

The frequency regions R_n corresponding to $S_n(jw)$,

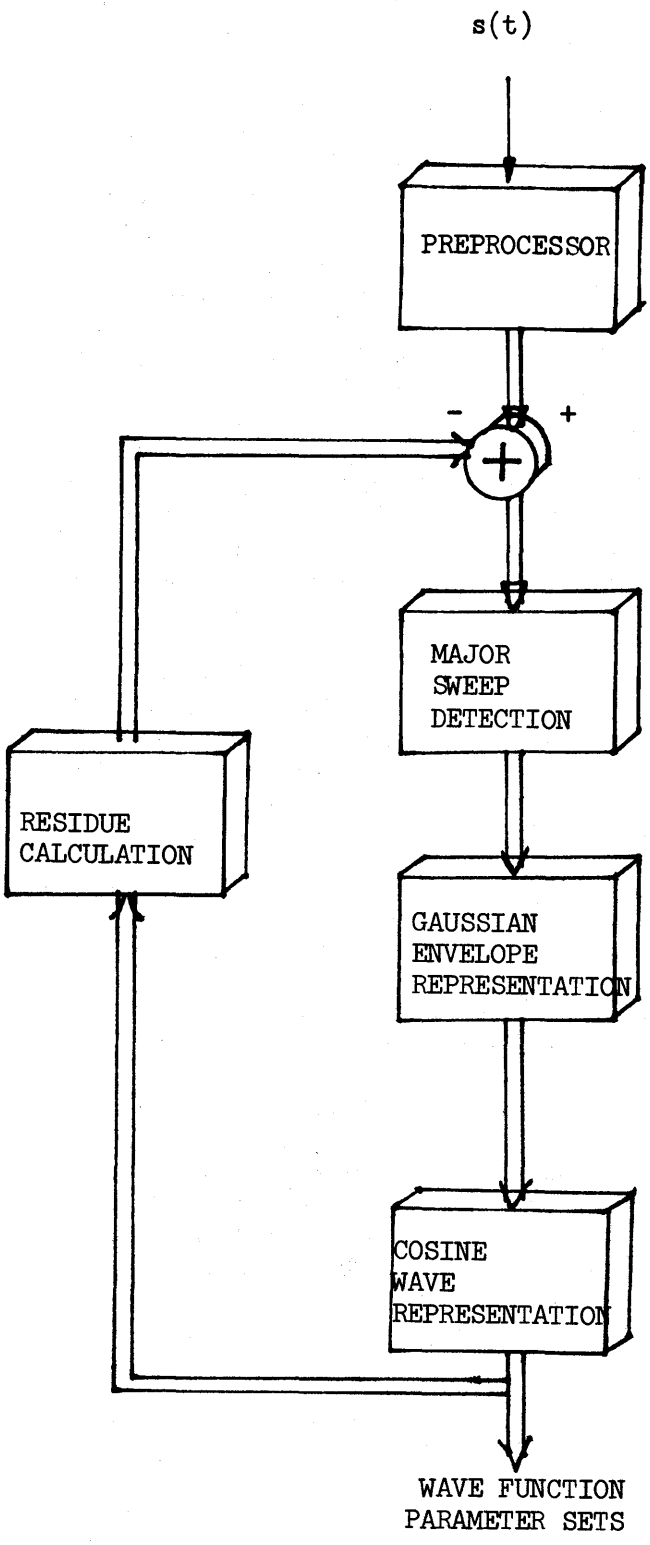


Figure 3—Block diagram of GCM analyzer system. Bold arrows denote multiple input-output situation.

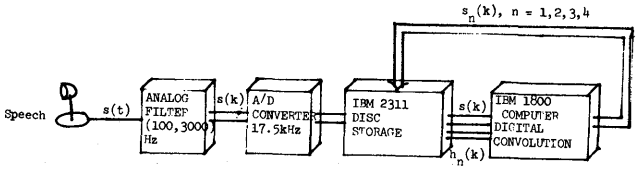


Figure 4—Detailed block diagram of preprocessing section of GCM analyzer

$n=1, \ldots, 4$ are divided in the following manner:

$$100 \leq R_1 < 400 \text{ Hz}$$

$$400 \leq R_2 < 900 \text{ Hz}$$

$$900 \leq R_3 < 1700 \text{ Hz}$$

$$1700 \leq R_4 < 3000 \text{ Hz}$$

This separation into four contiguous frequency regions corresponds to convolution of sin x/x type bandpass filters with $s(t)$ to obtain the substrings. By applying the discrete convolution equation, each of the four substrings for the system simulation is obtained as

$$s_n(k) = \sum_{j=-62}^{62} h(j)s(n-j)$$

where

$$s_n(k) = s_n(t) \mid_{t=kT} \quad k=0, 1, 2, \ldots,$$

$$h_n(j) = (B_n/2)[\sin(\pi B_n jT)/\pi B_n jT] \cos(2\pi F_n jT),$$

and T is the discrete sampling period. The parameters of the nth convolution kernal $h_n(t)$ are defined from the nth region. For example $B_2 = 900-400 = 500$ Hz and $F_2 = (400+900)/2 = 650$ Hz. The preprocessor portion of the wave function analysis system is simulated in the form of Figure 4. In the present configuration a string of up to 420 msec. can be recorded and stored on the disk. Once stored, $s(t)$ is then digitally convolved using the above algorithm to obtain the sub-strings $s_n(t)$, $n=1, \ldots, 4$. At the present sampling rate of 17.5 kHz each substring corresponds to 7440 16 bit computer words. Each substring is also stored on disk where it then becomes available for the wave function analysis programs.

To illustrate the effect of the preprocessor two examples are presented, one for a vowel and the other for a consonant. Shown in Figure 5 is a 35 msec portion of the vowel /i/ as in "eve" recorded from a male speaker. Examination of the general structure shows a stable repetition of a complex waveform. The substrings obtained by the just discussed algorithm are of a much less complicated form as can be seen from the figures.

In Figure 6 is shown a 35 msec. portion of the fricative consonant /s/ as in "she" from the same speaker. Examination of this string shows it to be of a somewhat random nature. Again the resultant substrings are of a much simpler form. Now the wave function analysis algorithms will be considered starting with the major sweep detection.

Major sweep detection

To perform wave function analysis certain segments of the filtered speech wave must be isolated as the regions over which a wave function will be fit. Based upon their time domain appearance we look for the time at which the envelope of the speech wave reaches a local maximum and define this as the location from which wave function parameters will be extracted. One method for doing this is to determine the absolute values of the extrema as the definition of the points defining the envelope which conceptually corresponds to full wave rectification and low pass filtering. The three main aspects of the algorithm namely (1) extrema detection (2) constraints on extrema and (3) criteria for major sweep will now be considered.

Initially the substring is tested to see whether the first sample is greater than zero. If so, each succeeding point is tested until a value is found which is less than the preceding value. This point defines a local maximum

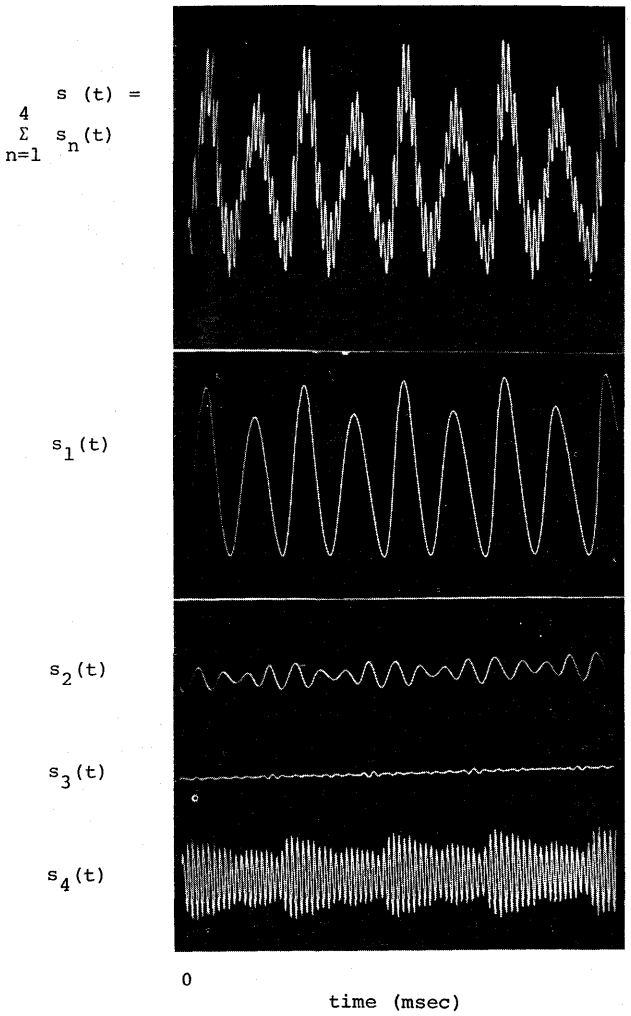


Figure 5—Representative portion of spoken vowel /i/ shown with substrings $s_n(t)$, $n=1, \ldots, 4$

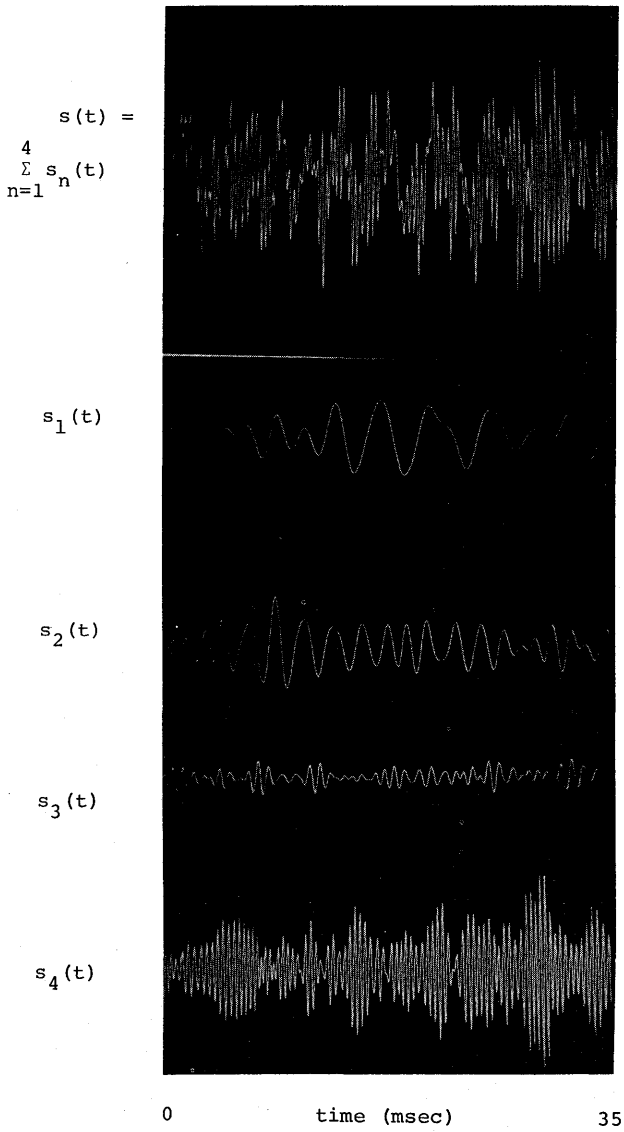


Figure 6—Representative portion of spoken consonant /s/ shown with substrings $s_n(t)$, $n=1, \ldots, 4$

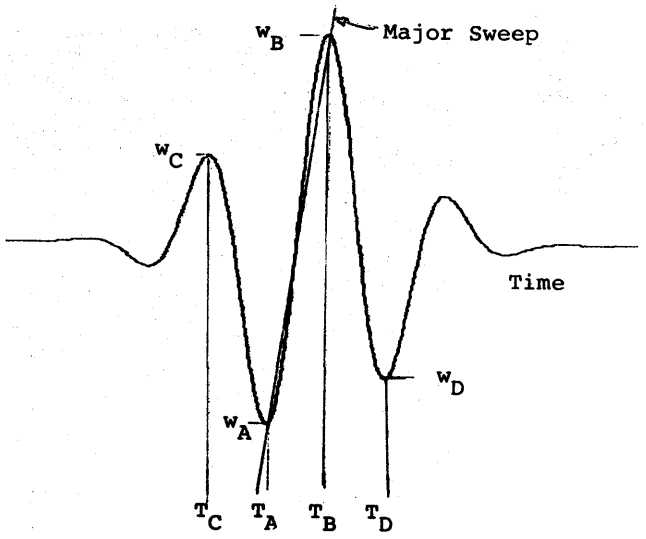


Figure 7—Extrema and time of occurrence definitions as applied to wave function behavior isolated around the major sweep

and a possible extremum value. A test is applied to insure that the time of occurrence satisfies the extrema constraints (explained below). If these are satisfied, an extrema time and slope value are defined. Then a test is applied for the major sweep (explained below). If the constraints on extrema are not satisfied the index counter is incremented and the next value is tested to see if x_{j+1} is less than x_j. The counter is incremented and this test is performed again until x_{j+1} is greater than x_j. At this point a local minimum has been found and it will then be tested as described above. If the first value of the signal is less than or equal to zero each succeeding point is tested until a value is found which is greater than the preceding value. Then the process continues as above. Thus the extrema detector can be viewed as a flip-flop arrangement whose states change whenever a local minimum or maximum is found. In any extrema detection scheme it is necessary to place certain constraints due to slight perturbations of insignificant perceptual value on the speech wave. It is desirable to constrain both the minimum amplitude difference and time difference between a potential extrema but since the system is bandlimited, use of time difference only has been shown to be acceptable. Since the system we are considering is bandlimited to 3 kHz, 0.1 msec is a reasonable value to use for the minimum allowable time before the next extremum can be defined. A sweep is defined as the distance from the previous extremum to the following extremum. The major sweep is defined by requiring from a sweep at time t, the following sweep to be greater in value and the previous sweep to be smaller in value. The region of the largest sweep then defines the major sweep.

Based upon the sweep as defined by the sweep detector output, each isolated segment is given the following definitions as shown in Figure 7 below. The region (T_A, T_B) defines the location of the major sweep and the location in which C must reside. The time T_D is the next extremum location past T_B and T_C denotes the extremum closest to the left of T_A. For the analysis procedure presented here, w_A through w_D correspond to the values of the extrema evaluated at the respective times, T_A to T_D.

Gaussian envelope representation

It has been previously demonstrated[10] that the exact (in the sense of representing wave functions with members of the same family) analysis applied to real filtered speech is inadequate. In this section an iterative solution to the problem of finding the best (in the sense of some predefined error criteria) wave function over a segment of filtered speech is presented. The problem we are working with falls under the classification of nonlinear programming with constraints, that is, problems in which mathematical programming techniques are used to minimize a given function $f(x_1, x_2, \ldots x_n)$ nonlinear in the independent variables by proper choice of a vector $x=(x_1, x_2, \ldots x_n)$, where the variables are constrained.

If the measured extremum w_i does not lie exactly on some particular wave function, there will be an associated error $e_i=f_i-\hat{f}_i$ where

$$\hat{f}_i=A\exp[(-\pi^2/S^2)(t-C)^2]\cos(w_0(t-C)-\phi)$$

and $f_i=|w_i|$ defines the absolute value of the ith extremum. Since we are measuring peaks we assume $\cos(\cdot)=\pm 1$. By defining $f_i=|w_i|$, the error e_i measures the deviation from the positive envelope and the measured extremum. It is desired to minimize in some sense the error between the envelope value and the speech waveform value at several different points by choosing the parameters A, S, and C properly. Given the original function f_n, $n=1, 2, \ldots N$ and estimate $\hat{f}_n$, $n=1, 2, \ldots, N$ the mean square error in discrete form is defined by

$$\epsilon=N^{-1}\sum_{n=1}^{N}(f_n-\hat{f}_n)^2$$

There are many other measures but we shall restrict our discussion to this one since it is probably the easiest and most meaningful to implement for our problem.

Using the mean square error criterion, analysis of real speech to obtain the A, S, C parameters for a selected segment implies minimization of the function

$$f(A, S, C) = \sum_{i=1}^{4} \{ \mid w_i \mid - A \exp[(-\pi^2/S^2)(t_i - C)^2] \}^2.$$

The most obvious method for finding the values of the independent variables which minimize a function is by setting the partial derivatives equal to zero and then solving for A, S, and C, i.e.,

$$\partial f/\partial A = 0 \qquad \partial f/\partial S = 0 \qquad \partial f/\partial C = 0$$

However, for this problem the function f is highly nonlinear in both variables S and C which for even small K (on the order of three or four) would become a very unwieldy algebraic problem. In addition this procedure does not allow constraints to be applied. As an example, the parameter N might be a negative number which is physically meaningless.

We therefore consider numerical techniques for minimizing functions subject to constraints. This subject is a complete field in itself and there are an endless variety of approaches which can be applied to this problem. For simplicity a steepest descent method will be used with the inclusion of upper and lower bounds on the variables A, S, and C. Although steepest descent methods are far from optimum in terms of speed of convergence (for ill-conditioned problems) it has been verified that by judicious choice of constraint equations and variables (in particular, the optimization of $1/S$ instead of S) quite satisfactory results can be obtained with relatively few iterations.

The modified steepest descent method used in this procedure is outlined below.

1. Define a set of lower bounds bl_i and upper bounds bu_i corresponding to variables x_i,

$$bl_i \leq x_i \leq bu_i \quad i = 1, 2, 3.$$

2. Starting with an initial estimate $\mathbf{x}(0) = [x_1(0), x_2(0), x_3(0)]$ the ith iteration is as follows:
3. The normalized gradient of f,

$$\nabla f(\mathbf{x}) / \| f(\mathbf{x}) \| = \frac{(\partial f/\partial x_1)\mathbf{i} + (\partial f/\partial x_2)\mathbf{j} + (\partial f/\partial x_3)\mathbf{k}}{\left[\sum_{i=1}^{3} (\partial f/\partial x_i)^2\right]^{1/2}}$$

is numerically calculated using central difference forms and the direction of the search is then

$$\mathbf{s}(i) = -\nabla f(\mathbf{x}) / \| f(\mathbf{x}) \|$$

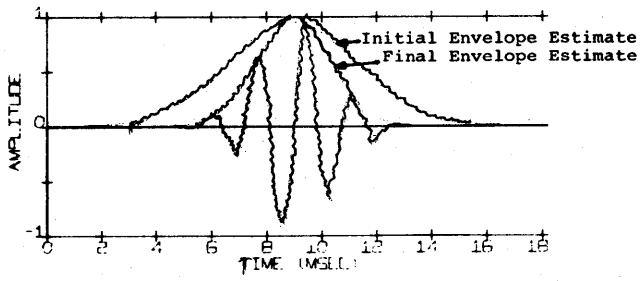


Figure 8—Example of GCM analyzer convergence capability. Signal described by $\{A, S, C, \phi, F\} = \{1, 6, 9, 90, 0.576\}$. Initial estimates $\{A_0, S_0, C_0,\} = \{0.95, 10.4, 8.97\}$. Final estimates (four iterations) $= \{A_f, S_f, C_f\} = \{.99, 5.9, 9.0\}$.

4. Starting with initial step length α_i, a new vector $\mathbf{x}(i)$ is calculated from

$$\mathbf{x}(i) = \mathbf{x}(i-1) + \mathbf{s}(i)\alpha_i$$

and using this new vector, $f[\mathbf{x}(i)]$ is evaluated.
5. If $f[\mathbf{x}(i)] < f[\mathbf{x}(i-1)]$ the move was successful and thus $\mathbf{s}(i)$ is doubled in length to give $\mathbf{s}(i) = 2\mathbf{s}(i-1)$ and step (4) is repeated. If $f[\mathbf{x}(i)] \geq f[\mathbf{x}(i-1)]$ the move was unsuccessful and thus the step length is reduced by some factor $\rho < 1$, i.e., $\alpha_i = \rho\alpha_{i-1}$ and (4) is repeated.
6. Procedure stops when some predetermined criteria on the number of iterations and/or error has been reached.

Figure 8 illustrates the capability of the analyzer to converge upon correct parameter values based upon initial conditions which are in considerable error with respect to the correct values.

Next the determination of boundary conditions on the parameters will be discussed and then the calculation of the remaining two parameters ϕ and F will be considered.

Bounding of A—First it is noted that the extrema fix a lower bound on A, i.e., $A \geq \langle \mid w_B \mid, \mid w_A \mid \rangle$. Thus

$$BL(A) = \max\langle \mid w_B \mid, \mid w_A \mid \rangle$$

Based upon the requirement that at least one full cycle must be contained within the envelope of the wave function a worst case is described by the situation where $\phi = \pm 90°$. Experimentally it has been determined that the bound $BU(A) = 1.5BL(A)$ is sufficient.

Bounding of C—Using the previously defined terms (T_A, w_A), (T_B, w_B), and (T_C, w_C), the center will always lie between T_A and T_B. Thus we have the very simple bounds $BL(C) = T_A$ and $BU(C) = T_B$.

Bounding of S—Bounding of S is the most difficult

part of the procedure since only the highest amplitude information near C is used in setting up the effective width of the envelope near zero amplitude. A lower bound on S can be easily obtained by observing that $S>T_B-T_A$ in all cases. As a first attempt at finding an upper bound to S it has been decided to measure the distances of straight line intersections from T_A with the time axis. A problem arises when two of the extrema happen to take on nearly equal values. In that case an upper bound is defined which depends on the effective frequency of speech sample and a pre-determined maximum N value. The bound is set at

$$BU(S)=N_{\max}/f_e=2N_{\max}(T_B-T_A)$$

To obtain rapid convergence in the algorithm it is necessary to operate upon $1/S$ instead of S since S takes on a relatively large range of values. If $S>1$ (msec.) for all conditions, then $0<1/S<1$, a much smaller range of values. Therefore $BL(S)$ is defined as $1+T_D-T_A$ to insure that $1/S$ is less than one. The initial value of S is chosen between $BL(S)$ and $BU(S)$. These values are not critical and several other methods might be just as good or better. For example, the initial condition and bounds of S could be defined by

$$1/S_{\text{initial}}=.5$$
$$BL(1/S)=0$$
$$BU(1/S)=1\ .$$

However by using the bounds described, better initial estimates (and thus faster convergence) is usually obtained. The important point is that $1/S$ must be used as the variable which is iterated upon and it must satisfy the bounds $0<1/S<1$.

Cosine wave representation

In most discussions of the wave function analysis technique the relevant parameters were the ASCON set.[6] However for the GCM formulation, a more meaningful parameter set is the ASCOF set where F is a frequency term that replaces N. The reason for this change is that in the representation the envelope behavior characterized by A, S, and C can be completely isolated from the cyclic behavior characterized by ϕ and F. In terms of equation 3,

$$F=(N^2-1)^{1/2}/S.$$

A logical criterion to apply in the determination of F is that near the center of the wave function the effective frequency equal F. The word effective is used because we are defining frequency from only four points in time, nearest C. If the speech were a cosine wave then the terms frequency (as applied to the cosine wave) and effective frequency (as applied to four points) would be identical. Assume that t_n, $n=0, 1, 2, 3$ are the times for which extrema are measured in the vicinity of the major sweep. The frequency estimator will be defined as

$$F=\tfrac{1}{6}\sum_{n=1}^{3}(t_n-t_{n-1})^{-1}.$$

The phase will be defined so that a close fit is made at $t=T_2$ ($T_2=T_B$ in the previous section), i.e.,

$$\cos[2\pi F(T_2-C)-\phi]=\pm 1$$

depending on the major sweep slope. Therefore

$$\phi=\begin{cases}2\pi F(T_2-C) & w_2>w_1\\ 2\pi F(T_2-C)+\pi & w_2<w_1\end{cases}$$

Residue calculation

To make an accurate wave function representation from filtered speech the wave function corresponding to the extracted parameters around a major sweep must be subtracted from the original waveform before proceeding in the analysis for the next wave function. This is because the parameters extracted over a limited interval of time describe a wave function that extends over a larger interval, i.e., for a small portion of time past the major sweep, the future values of the signal are being predicted. Thus subtraction of the wave function from the original signal generates the error or residue which is then analyzed for the next wave function. An additional task of the residue calculation portion is to set the index at which the sweep detector will start next. Its choice to a large extent determines the efficiency (and also accuracy) in the representation. The minimum value for the starting index of the next wave function in this study was set at $(C+T_D)/F_s$ where F_s is the sampling frequency, T_D is the last extrema point (largest time value considered) and C is the center of the wave function envelope. Figure 9 shows the representation and residue from filtered vowel /Λ/ as in "up" over (900,2000) Hz. Summarizing the operation of the GCM analyzer, the algorithmic procedure for extracting the string of ASCOF parameters is as follows: The peak values and corresponding times of the sampled substring $s_n(k)$, $k=1, \ldots, 4$ are obtained. Then a major sweep is isolated. Applying the algorithms just described, the wave function parameters are calculated. Using these parameters, the corresponding wave function is calculated and then subtracted out from the original speech data. The residue

is then operated upon in the same fashion to obtain the next set of parameters from the substring. Thus the process is basically an on-line system that conceptually with appropriate buffering could be a real time process since the analyzer inputs $s_n(k)$ are also conceptually real time operations. One can visualize the system then as a 4 analyzers $\times 5$ parameters/wave function $=20$ channel system whose bit rate will vary depending upon the incoming information rate. This is a rather significant aspect of the system. Note that no voiced/unvoiced decisions are required as in a channel vocoder. All time references are contained in the C parameters.

Wave function synthesis system

As opposed to the various types of voice processors now in use, by far the major share of computation is required in the analysis system. Assuming that the twenty channel signals have been correctly sorted into the four different time varying parameter sets, the synthesis is nearly a trivial matter. The synthesized substrings $\hat{s}_n(t)$, $n=1, \ldots, 4$ are obtained by application of equation 4. Thus

$$\hat{s}_n(k) = \sum_i \phi[\Omega(i, n), k]$$

where $\Omega(i, n)$ denotes the ith set of wave function parameters corresponding to the nth substring, and k denotes the discrete time index, i.e., $t=kT$, $k=0, 1, \ldots$, where T is the sampling period. The symbol ϕ denotes the wave function as a function of these variables. In computational form,

$$\phi[\Omega(i, n), k] = A(i, n) \times \exp\{[kT - C(i, n)]^2 \cdot \pi^2 / S(i, n)^2\} \cdot \cos\{2\pi F(i, n) \cdot [kT - C(i, n)] - \phi(i, n)\}$$

where

$$\Omega(i, n) = \{A(i, n), S(i, n), C(i, n), \phi(i, n), F(i, n)\}.$$

Since each wave function amplitude decreases as an exponential squared, only the wave functions located nearest to the corresponding present time t need to be evaluated at the index k.

The total synthesis of the estimated string is denoted by $\hat{s}(k)$ and is calculated in discrete form by summing the substrings. Thus

$$\hat{s}(k) = \sum_{n=1}^{4} \hat{s}_n(k)$$

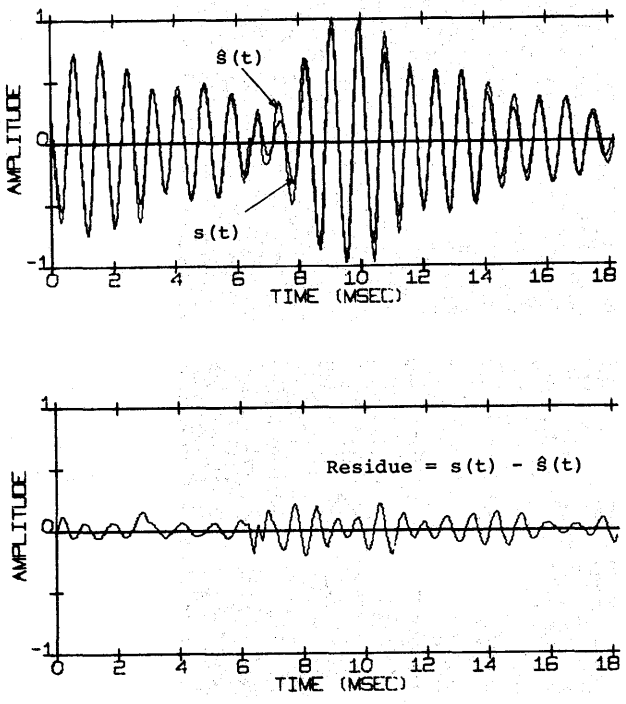


Figure 9—Comparison of filtered region of /ʌ/ as in up, $R=(900,2000)$ Hz and the wave function representation along with the residue. Representation composed of eight wave functions.

EXAMPLES FROM SYSTEM SIMULATION

As a demonstration of the wave function approach to speech analysis and synthesis, two multiphoneme words were chosen. The first is the word "large" which consists of the semi-vowel /l/, the vowel /a/, the semi-vowel /r/, and the plosive consonant /g/. The second word is "men" which consists of the closed mouth nasal consonant /m/, the vowel /I/, and the open mouth nasal consonant /n/. These words contain enough of a variety of the basic phonetic sounds to illustrate the accuracy of the wave function speech analysis and synthesis system.

Each word was uttered by a male speaker. Figure 10 shows a block diagram of the computer system that is used to simulate and observe the wave function speech analysis and synthesis system which has been described. The speech passed through an A/D converter which sampled the acoustic waveform at a 17.5 kHz rate with an 11 bit quantization precision. 7,440 16 bit words, which represent 420 milliseconds of the sampled speech, are stored in the IBM 2311 magnetic disk storage system. The sampled speech word is filtered four separate times into the four substrings $s_n(t)$, $n=1, \ldots, 4$. These represent the (100,3000) Hz frequency region.

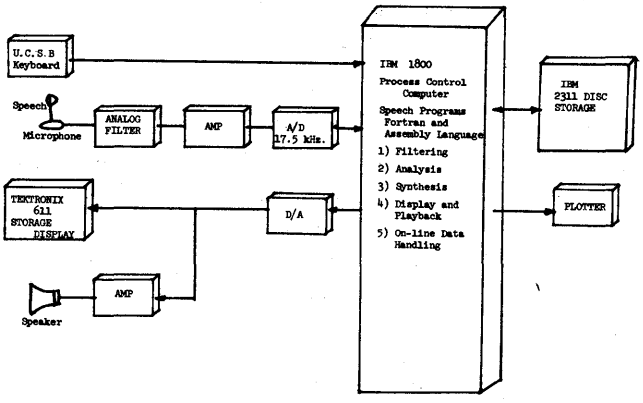


Figure 10—Block diagram of system used to implement the wave function analysis synthesis system

Each substring is then analyzed into its set of wave function parameters which are also stored in the 2311 system. Each parameter set is operated upon by the wave function synthesis program to generate its corresponding synthetic substring which is stored in the

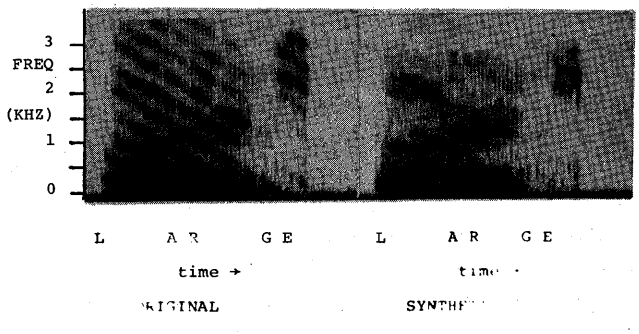


Figure 12—Comparison of spectrograms for word "large"

2311 system. The four synthetic substrings are summed together to form the synthetic string $\hat{s}(t)$ which is also stored in sampled form. At this point the original speech string, the four synthetic substrings, and the synthetic speech are all available for comparison and examination. They can be examined by visual observation on the Tektronix 611 storage display or listened to after the digital word is passed through a D/A converter-amplifier-speaker system.

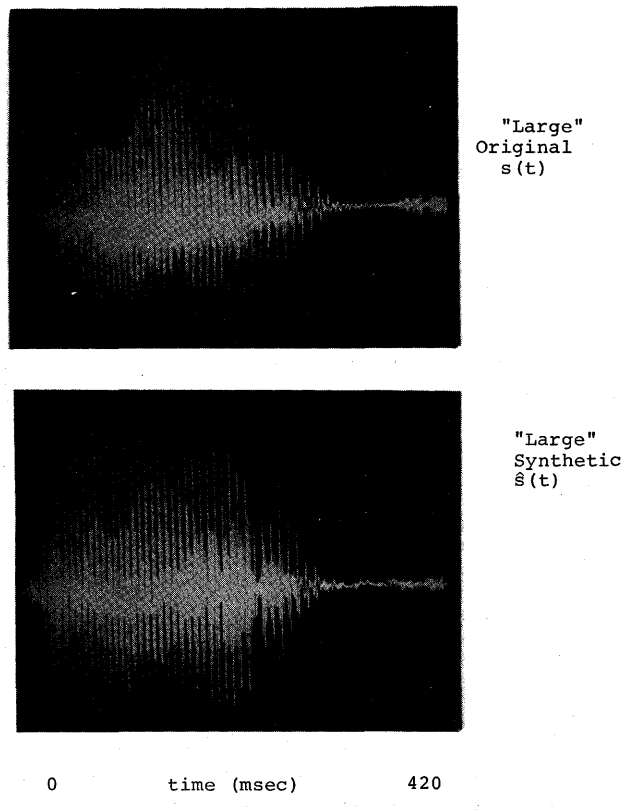


Figure 11—Comparison of original versus synthetic for word "large"

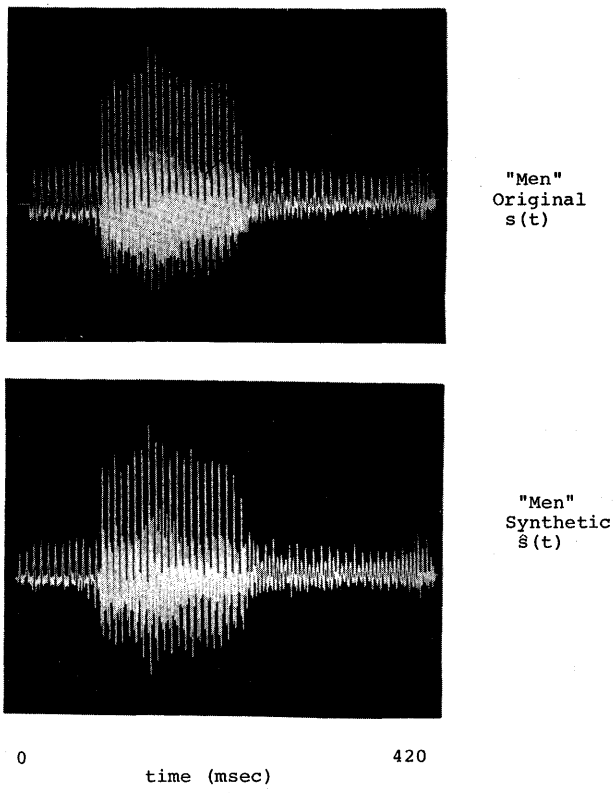


Figure 13—Comparison of original vs synthetic for word "men" in range (100,3000) Hz

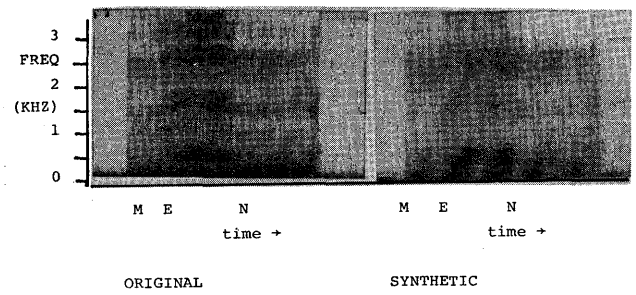


Figure 14—Comparison of spectrograms for word "men"

Figure 11 shows the original word "large" and the synthetic word "large" for comparison. As can be seen, there are detectable differences between the two time waveforms but a consistent agreement in their general structure exists. Figure 12 shows spectrograms of the original "large" and the synthetic "large" that were taken on a Kay Electric Company 6061B Sona-graph. It is apparent that excellent agreement exists between the frequency structure of the original versus synthetic words.

Figure 13 shows the time waveforms of the original "men" and the synthetic "men." Excellent agreement exists between the two time waveforms and only close examination can show the differences. Figure 14 shows the spectrograms of the original versus synthetic "men." The frequency structure of the synthetic word is in close agreement with that of the original word.

REFERENCES

1 J W COOLEY J W TUKEY
An algorithm for the machine calculation of complex fourier series
Mathematics of Computation Vol 19 No 90 April 1965

2 W T COCHRAN et al
What is the fast fourier transform?
IEEE Transactions on Audio and Electro-acoustics Vol AU-15 No 2 June 1967

3 A V OPPENHEIM
Speech analysis system based on homomorphic filtering
JASA Vol 45 No 2 1969

4 Applied Research Laboratory, Sylvania Electronic Systems
Real time digital vocoder demonstrated at 78th meeting of Acoustical Society of America San Diego Calif November 1969

5 A M NOLL
Short-time spectrum and cepstrum techniques for vocal-pitch detection
JASA Vol 36 No 2 pp 296-302 1964

6 G J CULLER
An attack on the problems of speech analysis and synthesis with the power of an on-line system
Presented to the International Joint Conference on Artificial Intelligence May 8 1969 Washington DC

7 J A HOWARD R C WOOD
Hybrid simulation of speech waveforms utilizing a gaussian wave function representation
Simulation Vol 11 No 3 pps 117-124 September 1968

8 J MARKEL
A gaussian cosine modulation (GCM) formulation for speech analysis and synthesis
Computer Research Laboratory University of California at Santa Barbara August 1969

9 B CAREY
Separation of the acoustic waveform into substrings
Computer Research Laboratory University of California at Santa Barbara March 1970

10 J MARKEL
Considerations in the development of a wave function analyzer
Computer Research Laboratory University of California at Santa Barbara January 1970

SIMCON—An advancement in the simulation of physical systems

by B. E. TOSSMAN, C. E. WILLIAMS, and N. K. BROWN

Johns Hopkins University
Silver Spring, Maryland

INTRODUCTION

The simulation of physical systems, once exclusively the domain of analog computers, is also being performed today by a variety of large-scale digital systems. Many specialized programs have been developed to permit the study of electronic circuits, biological organisms, and chemical processes, etc., by mathematical and empirical modeling techniques. So often these programs are written in procedural computer languages and may require months of developmental investment (and debugging) to yield a handful of computer-produced results. In addition, if simulation parameters are not well known, it may take hundreds of test runs to produce a simulation model that realistically portrays the actual processes involved. When the digital computer utilized to perform the simulation is operated in a batch-processing environment, such determination of model fidelity may take weeks because of the time lag from submission of test data to receipt of program results. Smaller computer systems may permit almost instantaneous turnaround and even on-line interaction, but too often do not have sufficient memory size, precision, or speed to be of much use in complicated digital simulations. The SIMCON simulation system is an integrated hardware/software system developed for the purpose of overcoming the difficulties of applying large-scale digital computers to the simulation of physical systems. The SIMCON system, which is used in conjunction with the IBM 360/91 computer, consists of the DSL/91 programming language, the SIMCON control console, a hybrid data interface, and a number of analog graphical peripherals. The SIMCON simulation system shown in Figure 1 includes the SIMCON console, incremental recorder and duel-channel X-Y plotter. The discussion of the SIMCON system presented herein includes the results from several hybrid satellite study programs, as examples of its application.

SIMCON SYSTEM DESCRIPTION

DSL/91 language

Quite often the simulation engineer or programmer will find that a number of the blocks of his mathematical model involve representation of simple electrical or mechanical devices like flip-flops, relays, or moderately complex transfer functions. Such application-oriented subprograms are rarely included in PL/1 or FORTRAN libraries distributed by computer manufacturers. In most cases, the routines will have to be generated for the simulation program, and will increase the amount of development time.

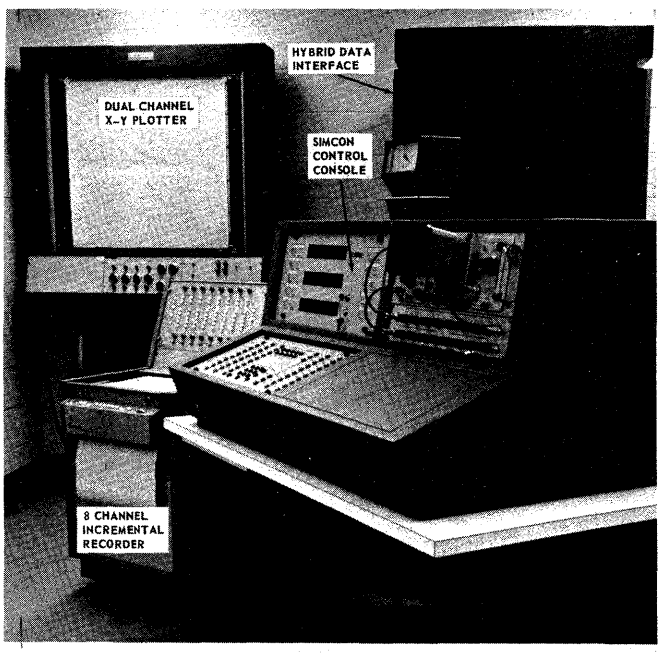


Figure 1—SIMCON simulation center

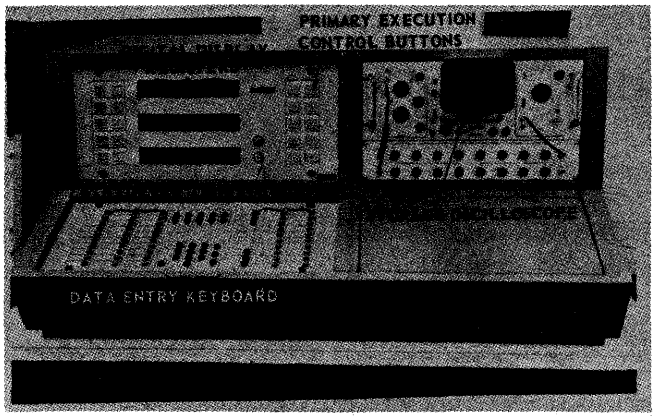


Figure 2—Simulation console

The SIMCON system offers a significant contribution by including a programming language for the express purpose of digital simulation of continuous system dynamics.

DSL/91 (**D**igital **S**imulation **L**anguage/91) is a non-procedural, problem-oriented computer language[1] designed to operate either in the batch-processing mode or in conjunction with the SIMCON console[2] in the time-shared mode of the Model 91's operation. Simulation models may be expressed either at the block diagram level or as systems of ordinary differential and algebraic equations. Since the language is basically non-procedural, equations and data may be entered in any order, with the task of proper statement sequencing being performed automatically by the language processor. DSL includes the FORTRAN IV procedural language as a subset, thereby extending its power to handle non-linear and time-variant problems of considerable complexity. Furthermore, many frequently occurring simulation blocks such as relays, switches, transfer functions, and function generators are provided in a simulation-oriented subprogram library.

Significant features of DSL/91 include a centralized built-in numerical integration system providing a choice of seven analytical techniques. These range from a simple rectangular method to the very stable, variable step-size, 5th order Milne Predictor-Corrector method. Input and output of simulation data and results are accomplished by pre-formatted I/O routines with emphasis on ease of use, not versatility.

Simulation control console(SIMCON)

In general, a simulation rarely performs in a manner realistic to the physical system the first time, even if correctly programmed. It is often necessary to run a simulation many times before the values of the parameters that cause behavior equivalent to the real system are found. In many cases the express purpose of performing a simulation is to determine if an improvement in the real system's performance can be attained by mere parameter optimization, rather than complete redesign. However, both of these processes require the results to be scrutinized by a trained and knowledgeable analyst, who can, in cases of multi-parameter optimization, play an important role in the problem solution. Thus, the ability of an engineer to observe and control the simulation's execution enhances the applicability of the digital computer to simulation, especially if he can iteratively rerun the problem and conveniently change data between, or even during, the runs.

The SIMCON console, shown in Figure 2, is designed to operate in conjunction with DSL/91 simulation programs, providing man-machine interaction during simulation execution. This would not be economically feasible on such a large system as the Model 91 were it not for the OS/MVT monitor system. The time-sharing environment permits other programs to run concurrently with the SIMCON system, and thus no computer time is wasted while the engineer studies his results or adjusts parameters at leisurely speeds. Furthermore, to increase the speed of man-machine interaction, the console is organized around a function keyboard concept, rather than a typewriter. Thus, standard commands are initiated by depressing buttons, rather than typing keyword phrases. Display of problem parameters is accomplished by alphanumeric NIXIE tube registers, rather than typewriter printed output. Hard-copy results, if desired, are available on the high speed system line printer.

The console is organized around an IBM 1892 Process Operator's Console (Model 11) with customer-specified button labeling, housed in a specially designed cabinet. One section of the console buttons is provided for control of program execution. Using these buttons, it is possible to start, stop, single-cycle, and restart the solution of the problem. Another button group controls the display of problem data in the NIXIE registers. The operator can select dynamically any three of 50 preassigned program variables, and simultaneously display these in floating-point format. Furthermore, certain special function buttons allow the display of any program variable whose relative address within the program is known, and the on-line assignment of this variable to any unused address button of the 50 available. Other buttons of this group initiate program dumps in scientific data formats, enable on-line selec-

tion of integration methods, and cause job termination. Finally, simulation data may be entered while the simulation is halted via a 14 button numerical keyboard, or dynamically modified by an incremental potentiometer.

Analog graphical peripherals

In close proximity to the console are a number of analog plotting systems capable of producing hard-copy graphics. A strip-chart recorder permits 8 problem variables to be plotted parallel to each other on a single continuous roll of reproducible paper. The variables are graphed vs. the simulation time base, with the paper chart being advanced by an incremental drive. This technique prevents distortion due to multi-programming during the simulation's execution. An analog memory scope allows simultaneous plotting of two variables against a third, with the resultant plot being retainable for the purposes of photography. Finally, a 30-inch vertical X-Y plotter can be used to produce two simultaneous indpendent cross plots which may overlap without damage to the plotting mechanism.

Hybrid data interface

The hybrid data interface, an IBM 1827 Data Control Unit, contains eight 13-bit Digital-to-Analog converters and a sixteen channel 14-bit Analog-to-Digital converter. Additional equipment, within the 1827, interfaces the 1892 Process Operator's Console (SIMCON) to the System 360/91 computer as shown in Figure 3. The D/A converters may be used to drive external analog equipment other than plotters. Thus, simulations may be interfaced to real analog hardware to test the performance of prototype and final designs functioning in an otherwise all digitally simulated environment. This mode of simulation is currently being used in the study of magnetically damped satellites, wherein actual flight hardware is coupled to the SIMCON system. Specific examples are covered in the applications section of this paper.

SIMCON SYSTEM OPERATION

Application programs for SIMCON are written in DSL/91 and are translated in FORTRAN IV by the DSL language translator, a self-contained program. The regular OS/360 FORTRAN compiler and linkage editor are then used to produce an interactive load module. The load module is a completely contained

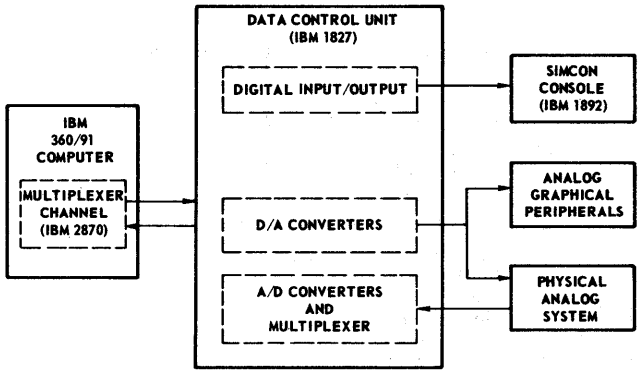


Figure 3—Block diagram of SIMCON simulation system

executable program containing DSL system routines, the SIMCON executive program, and a subroutine representing the simulation mathematical model equations.

The load module's execution is controlled from the SIMCON console by depressing preassigned function keys. Each of the 104 keys presents a unique 16-bit code to a digital input register in the 1827. The SIMCON executive program interprets the button codes and performs the desired action, providing feedback to the operator at all times by means of backlighted push buttons and NIXIE tube displays.

The software also contains terse diagnostic messages which are displayed using the alphanumeric NIXIE tubes and are accompanied by an audible alarm. These features serve to guide the user toward correct operation of the console.

Several manuals have been written to explain the operation of the system to non-computer oriented persons.[1,2] With these manuals and occasional assistance from their authors, the system is used at APL in an "open-shop" environment by engineers.

SIMCON SYSTEM DEVELOPMENT

The development of the SIMCON system was a joint undertaking by IBM, Harry Belock Associates, and the Applied Physics Laboratory. The console system was originally configured around IBM and EAI components similar to the DES-1[3] configuration by D. Stemmer. HBA used the SIMCON system in conjunction with a system 360/Model 44 computer. APL became interested in the system when it was first used by HBA to perform weapons system simulations under APL subcontract. In August, 1968, HBA returned their leased Model/44 computer to IBM and transferred the console system to APL.

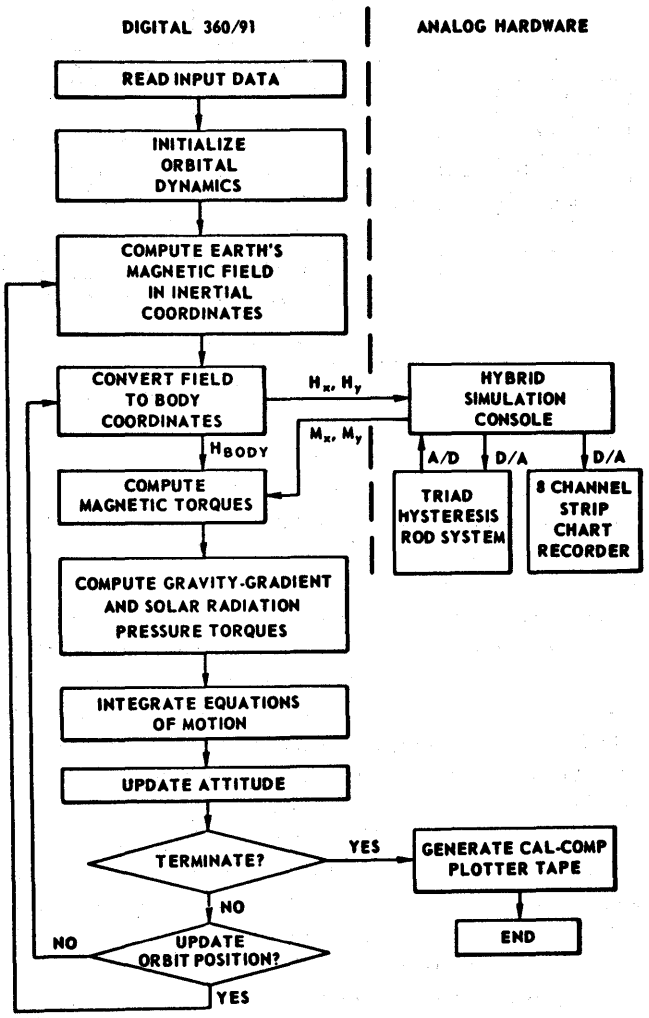


Figure 4—Simplified flow chart of TRIAD hybrid simulation

The original software for the SIMCON system included a simulation language and a set of programs which interfaced the DSL language to the SIMCON hardware. The DSL/44 simulation language[4] was written by D. G. Wyman and W. M. Syn of IBM. The interface programs were called Process Operator's Console Support 1, written by R. Bloom of IBM.[5] At HBA, these programs operated under 44PS, an executive system for the 360/44, and occasionally operated in supervisory state.

DSL/91 is an outgrowth of that simulation language, with language extensions developed at APL and an improved interface to the console program. A totally new set of console programs was developed by HBA under subcontract to APL, to replace the original P.O.C.S. 1 programs. The integrated system, including simulation language (DSL/91), console executive routine (SIMCON), 1827 and 1892 Input/Output Access Routines, runs under OS/MVT (or MFT) in "program state," using the OS input/output supervisor for I/O services. The development, testing, and debugging of the redesigned system at APL required about 18 man-months effort over a six month period.

SIMCON SYSTEM APPLICATIONS

Since most physical systems are described mathematically by means of differential equations, DSL/91 is of great utility in their representation and solution by digital computer. Aerospace, bio-medical, and transportation system engineers are primarily interested in the design and optimization of control systems. Here, DSL's transfer function blocks permit easy representation of complex transfer function networks. Today, however, we find increasing use of discrete control systems often based on the application of a digital computer as a part of the system itself. The ability to model complicated logical procedures by intermixing DSL and FORTRAN statements permit the complete digital simulation of such discrete hybrid control systems. At APL, the SIMCON system has been utilized in simulations of a fire-control radar system, guided missile target engagements, projectile rotational dynamics, and closed-loop satellite attitude control systems. Of these simulation programs, the most challenging and comprehensive have been the satellite attitude control programs. These programs have been designed around the hybrid interface to include actual hardware in the simulation and make maximum use of the SIMCON's parameter change, run control, and concurrent data display capabilities.

Satellite attitude control system simulations

This section presents the application of the SIMCON system to hybrid simulation of spacecraft with magnetic damping systems. The hybrid simulation consists of a digital solution of rigid body attitude dynamics, coupled, via SIMCON and its D/A and A/D interfaces, to an analog spacecraft attitude control system. This type of simulation was used in a study of the NASA GSFC Radio Astonomy Explorer-B,[6] APL's OSCAR and TRIAD[7] spacecraft and earlier by Gluck and Wong (without SIMCON).[8]

A simplified flow chart of the APL hybrid simulation is shown in Figure 4. The bulk of the simulation is performed digitally, while the analog portion consists of analog-type attitude control systems. In general,

the control system in its space environment receives as inputs the earth's magnetic field, as a vector in body coordinates. The control system includes elements which exhibit magnetic hysteresis and which cannot be satisfactorily modeled digitally. The outputs of the control system are signals proportional to magnetic dipole moments which, interacting with earth's magnetic field, produce desired torques. The digital portion of the simulation includes:

1. Computation of earth's magnetic field vector at any point in the satellite orbit via a 48 term Legendre expansion;
2. computation of attitude disturbance torques deriving from aerodynamic, gravity-gradient and solar radiation pressure; and
3. integration of the equations of motion using elements of the attitude transformation matrix as integration variables.

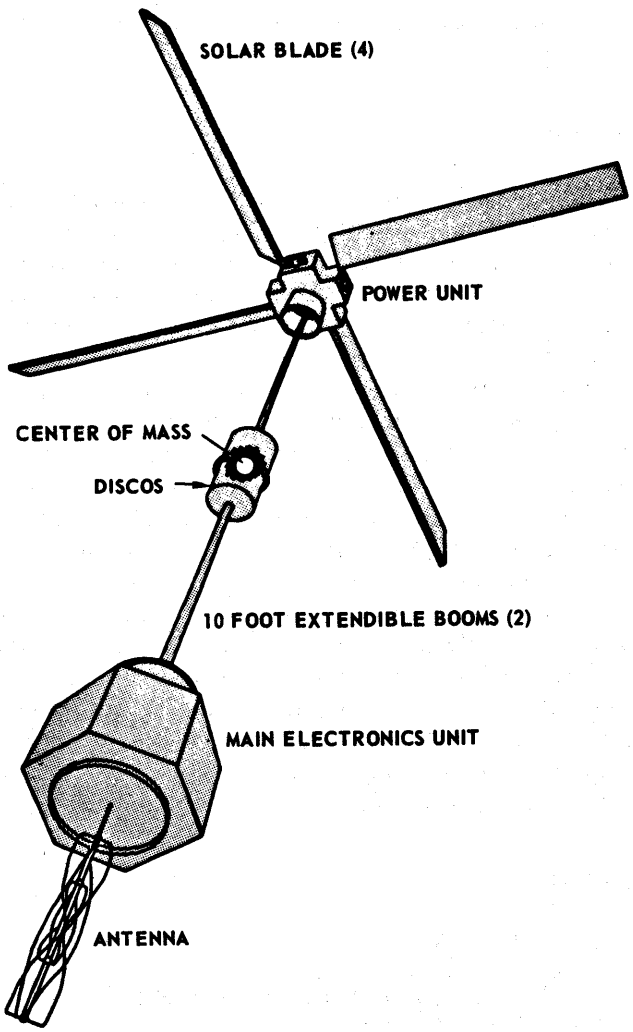


Figure 5—TRIAD orbit configuration

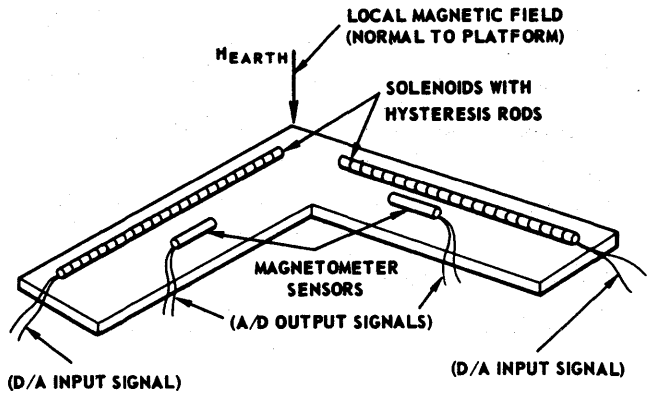


Figure 6—Hysteresis rod setup for computer

OSCAR and TRIAD simulations

OSCAR and TRIAD are gravity-gradient stabilized magnetically damped spacecraft. Both utilize an extended structure such that a large difference in the body moments of inertia give rise to gravity-gradient torques tending to orient the extended axis earthward. TRIAD includes, in addition, a high speed momentum wheel which orients a second axis along the orbit normal. The TRIAD configuration is shown in Figure 5. Both satellite systems are in relatively near earth orbit ranging from 890 to 1100 km altitude.

Spacecraft librations (motion of the extended axis about the local vertical) are damped by the interaction between long rods of a material which exhibits magnetic hysteresis and earth's magnetic field. These rods are installed within the solar panels. In the past, prediction of the attitude behavior of these satellites was based on highly complex digital models of the hysteresis function. The digital models were extremely expensive to run, often conflicted with each other and provided only general attitude patterns.

In the SIMCON hybrid simulation, an actual set of magnetic hysteresis rods, operating in their true analog environment, are linked to a digital simulation. The two rods are attached to an "L" shaped platform in the same relative position as on the extended solar panels (see Figure 6). The entire platform is positioned normal to the earth's local magnetic field. Solenoids, surrounding each rod, are driven by the computer (via D/A converters) and produce a magnetic field equal to that which would be observed by each rod if it were in flight. Magnetometer sensors adjacent to each rod sense the rod magnetization and provide a signal to the computer, via an A/D converter, proportional to the dipole moment of each rod. It is noted that the

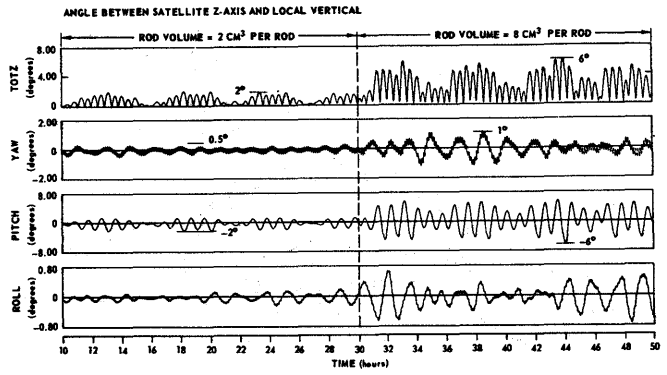


Figure 7—TRIAD steady-state dynamics computer results

magnetometer sensors are positioned such that there is no crosstalk between a magnetometer and an orthogonal rod.

Operating in the SIMCON environment, we are able to check the alignment and calibration of the hysteresis rod setup as an initialization phase of digital simulation. In addition, during execution, the simulation can be put into a "hold" mode while simulation study can be performed in one sitting.

Figure 7 is a representative example of the graphical output generated during a TRIAD simulation. In this case, the satellite's steady-state dynamics as a function of two different hysteresis rod volumes were examined. Using SIMCON, the effective rod volume was increased, from 2 cm^3 to 8 cm^3, after simulated hours. The difference in the TRIAD steady-state attitude angles, resulting from the rod modification (a capability also built into the real system), is displayed in Figure 6.

The execution cost of TRIAD and OSCAR hybrid simulations have averaged \$1.50 per simulated orbit or \$20 per simulated day. There are twelve variables of integration with magnetic field computation and D/A and A/D conversions performed every 10 seconds.

Radio astronomy explorer satellite simulation

The RAE hybrid simulation was similar in execution to the OSCAR and TRIAD with the exception that the analog portion consisted of a complete flight attitude control system electronic package. The RAE attitude control system, as described in Reference 9, consisted of a set of magnetometers, electromagnets and a signal processing package which included a hysteresis function generator. Hysteresis rods could not be used because of the exceedingly weak magnetic field at the RAE altitude (6000 km). The hysteresis function generator along with the electromagnets provided a damping system equivalent to an extremely large set of hysteresis rods. The flight hardware also included a command link for changing operating modes of the system and the gain of the hysteresis function generator.

This RAE simulation demonstrated the following capabilities of hybrid simulation via SIMCON:

1. the magnetic control system hardware was exercised in the same fashion as it would be operated in orbit;
2. all the digital facilities of the IBM 360/91 in a time-sharing multi-variable task environment were available;
3. execution of program control via man-in-the-loop interaction was provided by the SIMCON terminal; and
4. concurrent analog display of control system performance was provided by analog peripherals, including the eight channel incremental recorder.

SUMMARY

SIMCON has been presented as a significant advancement in the simulation of physical systems. It includes a digital simulation language for describing the dynamics of continuous systems and an interactive terminal for continuous program monitoring and hybrid capability.

DSL/91, the digital simulation language adapted to the IBM 360/91, utilizes the building block approach of digital-analog simulators but retains the power of logical and algebraic equation notation. DSL and FORTRAN statements may be intermixed allowing existing FORTRAN subroutines to become DSL function blocks.

The SIMCON control console allows on-line monitoring of problem execution, modification of data, early termination of a run or execution of sequential runs. The console, in conjunction with DSL/91, can thus test real physical system components in a closed-loop interactive environment as illustrated by the RAE and TRIAD simulations.

REFERENCES

1 N K BROWN
DSL/91 programmer's guide
JHU/APL Report BCE-T-0142 April 1 1970
2 R L McCUTCHEON
Simulation console operator's guide
JHU/APL Report BCP-461 (BCE-T-0143) November 1969

3 L LEVINE
A new digital computer for solving differential equations
Simulation April 1965 (DES-1)

4 S M SYN D G WYMAN
An IBM 360 Model 44 program for the simulation of process dynamics
(DSL/44) Contributed program No 360D 43.1.002

5 R BLOOM
Process operator's console support 1
(POCS1) Contributed Program No. 360D 16.8.001

6 B E TOSSMAN
RAE-B hybrid simulation and improved magnetic stabilization system
APL Internal Memorandum S2P-2-238 April 7 1969

7 C E WILLIAMS B E TOSSMAN N K BROWN
Interactive hybrid computer simulations of magnetically damped spacecraft
To be presented at AIAA Guidance Control and Flight Mechanics Conference Santa Barbara California August 17-19 1970

8 A WONG R GLUCK
Inline hybrid computer for simulation of passively stabilized satellites
Journal of Spacecraft and Rockets Vol 6 No 7 July 1969

9 B E TOSSMAN
Magnetic attitude control system for the radio astronomy explorer-A satellite
Journal of Spacecraft and Rockets Volume 6 No 3 March 1969

COMSL—A communication system simulation language

by R. L. GRANGER and G. S. ROBINSON

Communications Satellite Corporation
Washington, D.C.

INTRODUCTION

In recent years, computer simulation has come to play an important role in the design of communication systems.[1,2,3] Such systems frequently cannot be classified as either purely continuous or as purely discrete but are, instead, a combination of both continuous and discrete subsystems. The simulation of such hybrid systems cannot readily be carried out either in a discrete simulation language such as GPSS or in a continuous simulation language such as CSMP. As a consequence, most simulation models of communication systems are written in some general purpose higher level language, e.g., FORTRAN, ALGOL or PL-1. A notable exception to this general rule is BLODI and BLODIB described in Reference 4. The simulation of even moderately complicated communication systems in a general purpose language poses several major problems. First of all, the simulation model tends to be time consuming to write and debug, difficult to modify once written, and, unless considerable care is taken, requires an inordinate amount of time to execute. However, the most important weakness of a communication system simulation model written in a higher level language is that the model usually is written by someone with a limited knowledge of the actual system being simulated. As a consequence, there frequently exists the possibility that the simulation model differs in some significant, but unnoticed, respect from the actual system. Primarily to avoid this problem, a new simulation language, COMSL, is described which facilitates the simulation of a wide variety of communication systems.

COMSL, a Communication System Simulation Language, is designed for use by the typical communication systems engineer who frequently has a modest knowledge of some higher level language but very little experience in the use of that language. COMSL is a block diagram oriented language which is relatively easy to learn and to use. Given a block diagram description of a communication system, a valid COMSL model can be written in a few hours whereas a similar model written in a general purpose language, e.g., Fortran, might take a capable programmer several weeks or even months.

A preliminary version of a translator which converts valid COMSL statements into Fortran code has been written. Care has been taken to ensure that the Fortran code generated by the translator is as efficient as possible. For example, few subroutine or function calls are generated by the translator except to compute certain initial system parameters. Since the simulation of most voice communication systems requires a sampling rate of at least 8 KHz, a significant execution time savings is achieved by avoiding numerous repetitive subroutine calls. The translator is embedded in the operating system in such a way that its presence is unnoticed by the COMSL programmer. A few simple control statements together with the COMSL system description is all that is required to obtain a voice tape "produced" by the simulated system. For video communication systems a digital tape is produced from which one or more "frames" can be obtained using a flying-spot scanner.

FUNCTIONAL CHARACTERISTICS OF COMSL

A simulation program written in COMSL consists of the following four types of statements:

1. Block definition statements which define the attributes of the various system blocks.
2. System definition statements which describe the interconnection of the various blocks.
3. Signal analysis statements which permit the computation of certain common statistics for any arbitrary signal, e.g., first and second moments, signal-to-noise ratio and power spectrum estimates.
4. Control statements which specify options re-

lating to the translation, execution, and output phases of a program.
5. User defined blocks written as standard Fortran subprograms.

Block definition statements

In order to simulate a given communication system, the user examines the block diagram description of the system and writes a single block definition statement for each block that appears on the block diagram. The block definition statements, which may be listed in any order, are written in the following format:

BLKTYPE(BLKNUM, ATTRIBUTES). (1)

BLKTYPE refers to one of the block types shown in Table I. BLKNUM is an arbitrary, but unique, number assigned to a particular block of this type. For example, if there are six filter blocks in a particular system, BLKNUM would take on values 1, 2, ... 6. ATTRIBUTES refers to a list of attributes which define a particular block of type BLKTYPE. The following is an example of a typical block definition statement:

FILTR(3, 1, 16, 250., 750., .5,). (2)

By comparing the attributes listed above with those listed under block type FILTR in Table 1, we see that statement (2) defines a 16th order band-pass Chebyshev filter with cutoff frequencies of 250 and 750 Hz and a pass-band ripple of .5 db. The filter has been assigned an arbitrary BLKNUM of 3.

System definition statements

Having described the attributes of all blocks in the block diagram using block definition statements described above, the user next describes the interconnection of the various blocks using so-called "system definition statements." These are written in the following format:

Y=BLKTYPE'BLKNUM'(X). (3)

Y is an arbitrary name, subject to certain restrictions mentioned later, assigned to the output of a particular block. X is the input to the block and BLKNUM is the arbitrary block number given to the block in the corresponding block definition statement. It is important to note the one-to-one correspondence between block definition statements and system definition statements. While the two statement types could have been combined into a single statement type, it is felt that this separation of assignment of block attributes and the definition of system structure yields a more natural, if somewhat more lengthy, description of a given communication system.

As an example of a typical "system definition statement" consider the following:

$X2$=FILTR3 ($X0$). (4)

The above statement simply states that the input to filter 3 is $X0$ and its output is $X2$. The attributes of filter 3 have, of course, been defined in a previous block definition statement.

In order to avoid defining an unnecessarily large number of trivial blocks to perform simple arithmetic functions, the following basic operators are used:

Symbol	*Function*
+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Signal analysis statements

In the simulation of any system it is frequently desirable to ascertain characteristics of a given signal at an arbitrary point in the system. For example, one may wish to know the signal-to-noise ratio at a particular point in the system. Alternatively, one may wish to know the spectrum of an arbitrary signal. This is accomplished in COMSL through the use of so-called "signal analysis statements" which take the following form:

COMPUTE CTYPE(S1, S2). (5)

CTYPE is the pseudonym for the characteristic to be computed, S1 is the signal to be analyzed and S2 is a reference signal which may or may not be needed. As an example, assume that one wishes to compute the signal-to-noise ratio at the output of a subsystem where the input signal is XIN and the output signal is XOUT. The following statement accomplishes this:

COMPUTE SNR(XIN, XOUT, 1). (6)

At the termination of the simulation, the above statement causes the following to be printed once each second for the duration of the simulation run.

SIGNAL-TO-NOISE RATIO=_____db FOR XIN, XOUT at time_____

where

$$\mathrm{SNR}=10\log_{10}\left[\left[\sum_{i=1}^{N}(X\mathrm{OUT}_i-X\mathrm{IN}_i)^2/X\mathrm{IN}_i^2\right]\Big/N\right] \quad (7)$$

TABLE I—Available Function Blocks

BLOCK TYPE	FUNCTION	GENERAL FORM[1]	BLOCK ATTRIBUTES
FILTR	analog filter	FILTR (I, J, K, A, B, C, D)	J = filter type: 0—Butterworth, 1—Chebyshev, 2—Elliptical K = the order of the filter A = lower cut-off frequency B = upper cut-off frequency C = pass-band ripple (J = 1, or 2) D = pass band/stop band ratio (J = 2)
CXFRM	user specified continuous transfer function	CXFRM (I, J, K, A, B) Y = CXFRM'I'(X)	J = # of zeros of H(s) K = # of poles of H(s) A = array containing complex poles B = array containing complex zeros
DXFRM	user specified discrete transfer function	DXFRM (I, J, K, A, B) Y = DXFRM'I'(X)	J = length of array A K = length of array B A, B } coefficients describing H(z)
CMPRS	μ-law compressor	CMPRS (I, A, B) Y = CMPRS'I'(X)	A = μ B = range of input to block
EXPND	μ-law expander	EXPND (I, A, B) Y = EXPND'I'(X)	A = μ B = range of input to block
QUANT	quantizer	QUANT (I, J, A) Y = QUANT'I'(X)	J = # of bits, i.e., # of distinct quantization levels = $2^j - 1$. A = range of input to block
DELAY	arbitrary length delay	DELAY (I) Y = DELAY'I'(J)	J = duration of delay
ZOH	zero-order hold	ZOH (I, A) Y = ZOH'I'(X)	A = incremental hold, i.e., $Y_i = X_i$ if $\mid X_i - X_{i-1} \mid \geq A$ $Y_i = X_{i-1}$ if $\mid X_i - X_{i-1} \mid < A$
DERIV	derivative	DERIV (I) Y = DERIV'I'(X)	
INTG	integral	INTG (I, A) Y = INTG'I'(X)	A = initial value
LIMTR	limiter	LIMTR (I, A, B) Y = LIMTR'I'(X)	A = upper limit, B = lower limit $Y_i = X_i$ if $A \leq X \leq B$
SPCNV	serial-to-parallel converter	SPCNV (I, J) Y = SPCNV'I'(X)	J = # of parallel paths out of the converter
PSCNV	parallel-to-serial converter	PSCNV (I, J) Y = PSCNV'I'(X)	J = # of parallel paths into the converter
FFT	fast Fourier transform	FFT (I, J) Y = FFT'I'(X)	J = dimension of the transform note: X must be the output of an SPCNV block

TABLE I—Available Function Blocks (Continued)

BLOCK TYPE	FUNCTION	GENERAL FORM[1]	BLOCK ATTRIBUTES
IFFT	inverse fast Fourier transform	IFFT (I, J) Y=IFFT'I'(X)	J=dimension of the transform
FHT	fast Hadamard transform	FHT (I, J) Y=HT'I'(X)	J=dimension of the transform
FWR	full wave rectifier	FWR (I) Y=FWR'I'(X)	
HWR	half wave rectifier	HWR (I) Y=HWR'I'(X)	
CNOISE	generates a specified channel error rate	CNOISE (I, J, A, B)	J=# of bits of quantization A=maximum amplitude of signal B=desired error rate
URAN	"noise" generator with a uniform distribution	URAN (I, J) Y=URAN'I'	J=any odd integer
GRAN	"noise" generator with a Gaussian distribution	GRAN (I, J, A)	J=any odd integer A=RMS noise level
CLOCK[2]	timing signal generator	CLOCK (I, J_1, J_2, J_3, . . . J_n) ———	note: generates a timing signal ITZ'I'=J_n where J_n=0 or 1
AND[3]	logical AND	——— Y=AND'I'(X)	note: Y=X if ITZ'I'=1 Y=0 if ITZ'I'=0
NAND[3]	logical NAND	——— Y=NAND'I'(X)	note: Y=0 if ITZ'I'=1 Y=X if ITZ'I'=0
INPUT	inputs data	——— Y=INPUT (FILE 'N')	note: N=1, 2, 3, or 4, depending on which of four input files is to be read
OUTPUT	outputs data	OUTPUT'I' (FILE 'P')=X	note: P=5, 6, 7, or 8, depending on which of four output files is to be written

Note: 1—The block definition statement and corresponding system definition statement are listed for each block. If one or the other statement type is not required, its absence is denoted by ———.
2—For example, CLOCK(3, 1.0) causes timing signal ITZ3 to take on values 10101010 . . . .
3—The corresponding timing signal must be generated by the presence of a CLOCK definition statement.

As a further example, if one wishes to know the power spectrum of an arbitrary signal, say $Y1$, then the following statement is coded:

COMPUTE PWRSP'I'($Y1$, N) (8)

where I is either 0 or 1 depending on which of two algorithms is to be used.[5,6] At the termination of the simulation an estimate of the relative power density of signal $Y1$ is printed at $N+1$ equally spaced frequencies between 0 and the Nyquist frequency.

TABLE II—Signal Analysis Statements

NAME	FUNCTION PERFORMED	GENERAL FORM	COMMENTS
STAT	computes first moment, second moment, and standard deviation	COMPUTE STAT (X, INCR)	computes and prints the first and second moments and standard deviation of X every 'INCR' seconds
PDF	computes the probability density function	COMPUTE PDF (X, XMAX, XMIN, NPOINTS)	computes and prints the relative occurrence of X over the interval XMAX to XMIN at intervals $\frac{(XMAX-XMIN)}{NPOINTS}$
SNR	computes the signal-to-noise ratio	COMPUTE SNR (X1, X2, INCR)	computes and prints the signal-to-noise ratio every 'INCR' seconds. X1 is the 'input' signal and X2 is the 'output' signal.
PWRSP	computes an estimate of the power spectrum	COMPUTE PWRSP'I'(X, N)	computes an estimate of the power spectrum of X at N equally spaced frequencies between DC and the Nyquist frequency using one of two different algorithms (I=0 or 1)

Control statements

A maximum of six control statements are needed to execute a COMSL program. These control statements are typically written in the following order:

INPUT ON FILE *'N'*

OUTPUT ON FILE *'P'*

(or, OUTPUT ON NEWFILE)

RUN TIME=_____SECONDS

(or, RUN TIME=____ FRAMES)

BLOCK DEFINITION

{Block definition statements}

SYSTEM DEFINITION

{System definition and signal analysis statements}

END SIMULATION

The nature of the control statements is, hopefully, more or less self-explanatory. The first statement simply specifies which of the available files are to be used in the simulation. The OUTPUT statement specifies which file the result of the simulation is to be written on. The RUN TIME statement specifies the length of the simulation in seconds for voice systems or in frames for video systems. THE BLOCK DEFINITION, SYSTEM DEFINITION and END SIMULATION statements serve only to separate block definition from system definition statements. Only the END SIMULATION statement need be present if the program contains only COMSL statements, i.e., if no user defined subroutine or function subprograms are present.

User defined blocks

One important feature of the language is the capability for defining special purpose blocks in the form of standard Fortran subprograms. In addition, any legitimate sequence of Fortran statements can be incorporated in the text of a COMSL program, subject to the following restrictions which somewhat simplify the translator design:

1. All TYPE, DIMENSION, COMMON and DATA statements must precede the SYSTEM DEFINITION statement.
2. The letter *Z* is not a permissible Fortran character.
3. Fortran statement numbers greater than 800 are prohibited.

PRINCIPAL FEATURES OF THE LANGUAGE

The systems which COMSL can simulate are combinations of sampled-data and bandlimited continuous subsystems. Incorporated in the language are functional blocks which create efficient sampled-data approximations to arbitrary continuous transfer functions using either the Z-transform, the bilinear Z-transform or the matched Z-transform. The user of the language need not concern himself with the details of these approximations assuming the input signal is bandlimited, a reasonable assumption for most voice and video communication systems. The functional block FILTR automatically generates a sampled-data approximation to an analogue filter of standard form. The user need only specify the conventional filter parameters in Table I. One of three sampled-data transformations is then automatically selected depending on the filter type, i.e., whether it is a low-pass, band-pass, high-pass, or band-stop filter.

In addition to creating sampled-data approximations to continuous filters, COMSL permits the user to define in a straightforward way any arbitrary continuous system transfer function. A sampled-data approximation to that transfer function is automatically generated using the functional block CXFRM described below.

In a similar fashion, the user can define an arbitrary sampled-data transfer function in a straightforward way using the functional block DXFRM.

Specification of an analogue filter, using functional block FILTR

The functional block FILTR creates a sampled-data approximation to an analogue filter of standard type. It generates a set of difference equation coefficients which define a Z-transfer function of the following form:

$$H(z) = \sum_{i=1}^{P} (A_{2i-1}+A_{2i}z^{-1})/(B_{2i}z^{-2}+B_{2i-1}z^{-1}+1) \quad (9)$$

where $z^{-1} = \exp(-sT) =$ the unit delay operator and $T =$ unit sampling interval.

In order to obtain coefficients $\{A\}$ and $\{B\}$ above the following computational sequence is carried out:

Step 1: The poles and zeros of the low-pass continuous filter

$$\text{prototype} \begin{Bmatrix} \text{Butterworth} \\ \text{Chebyshev} \\ \text{Elliptic} \end{Bmatrix} \text{ are determined.}$$

Step 2: One of four standard band transformations is applied to the low pass prototype in order to obtain the desired continuous system transfer function, $H(s)$.

Step 3: The continuous transfer function is then transformed using one of three sampled data transformations, i.e., the Z-transform, the bilinear Z-transform, or the matched Z-transform.

The complete algorithm, outlined above, used to compute coefficients $\{A\}$ and $\{B\}$ will be published later. To some extent it parallels Reference 7.

Having determined the appropriate $H(z)$ given by Equation (9) the corresponding time domain difference equations can be written in terms of coefficients $\{A\}$ and $\{B\}$:

$$Y(nT) = A_{2i-1}f_i(nT) - A_{2i}f_i(n-1) \quad (10)$$

where

$$f_i(nT) = X(nT) - B_{2i-1}f_i(n-1)T - B_{2i}f_i(n-2)T \quad (11)$$

$$(i=1, 2, \ldots, P).$$

$X(nT)$ is the input to the filter at time nT and $Y(nT)$ is the output at time nT.

It may be worthwhile to note at this time that the so-called "parallel" realization of an arbitrary sampled data transfer function of the type given by Equation (9) is not the computationally most efficient realization. In particular, an equivalent transfer function can be obtained which yields an expression for $Y(nT)$ in terms of a single high order difference equation. This simpler or so-called "direct" realization tends, however, to be numerically unstable if the order of the difference equation is greater than about 3. An excellent discussion of the merits of apparently "equivalent" realizations of an arbitrary $H(z)$ can be found in Reference 8.

An important feature of COMSL is that it prints (or plots) both the time and frequency domain response of the sampled-data approximation to the desired analogue filter. In theory, it is not possible to match exactly both the time and frequency domain response of the desired analogue filter. In practice, however, it has been found that differences are almost invariably insignificant. For example, if a 16th order Chebyshev filter with a pass-band ripple of 0.5 db is desired, the sampled data approximation will have a pass-band ripple within $\pm.02$ db of its desired value. In addition, the insertion loss at the specified cut-off frequencies will be within $\pm.02$ db of the nominal value and the fall-off of the "skirts" of the pass-band will closely approximate the nominal value except near 0 and f_{Nyquist}.

Specification of an arbitrary continuous transfer function, $H(S)$, using functional block CXFRM

An arbitrary continuous system transfer function, $H(s)$, can be specified in COMSL through the use of the functional block CXFRM. The user specifies the desired transfer function, $H(s)$, in terms of its poles and zeros., i.e.,

$$H(s) = \left[\prod_{i=1}^{M}(s-z_i)\right] \Big/ \left[\prod_{j=1}^{N}(s-p_j)\right], \quad (M<N) \quad (12)$$

A sampled data approximation to $H(s)$ is then obtained by applying one of three arbitrary sampled data transforms. The user may specify the transformation to be used or he may permit the system to select the "best" of the three transformations. It accomplishes this by examining the frequency response of the desired $H(s)$ and choosing that sampled-data transform which minimizes the RMS difference between the desired frequency response and the response of the sampled data approximation. As a normal output, a table (or plot) of the time and frequency response of both the specified H(s) and the resultant $H(z)$ is generated.

Specification of an arbitrary sampled data transfer function, H(z), using functional block DXFRM

An arbitrary discrete, i.e., sampled-data, transfer function can be specified in COMSL through the use of the functional block DXFRM. The user specifies the desired transfer function, $H(z)$, in one of the following ways:

Form 1:

$$H(z) = \left[\prod_{i=1}^{M}(z-z_i)\right] \Big/ \left[z\prod_{j=1}^{N}(z-p_j)\right] \quad (13)$$

Form 2:

$$H(z) = \sum_{i=1}^{N}(A_{2i-1}+A_{2i}z^{-1})/(B_{2i}z^{-2}+B_{2i-1}z^{-1}+1) \quad (14)$$

An equivalent set of difference equations corresponding to (13) or (14) above is generated by DXFRM. As a normal output, the time and frequency response of the specified discrete transfer function is printed (or plotted).

EXAMPLES ILLUSTRATING THE USE OF THE LANGUAGE

In order to illustrate the use of the language a relatively simple voice PCM system is shown in Figure 1

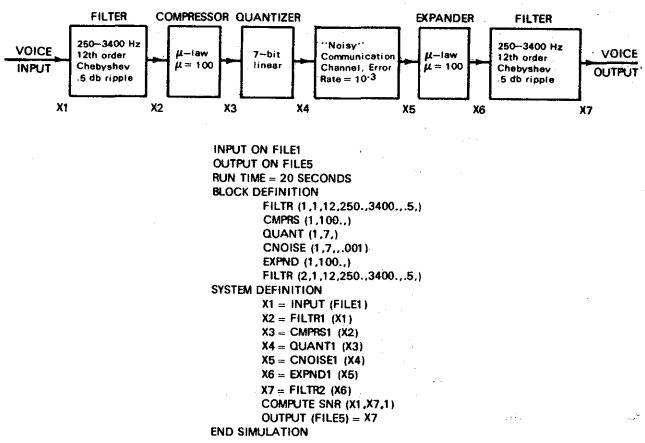


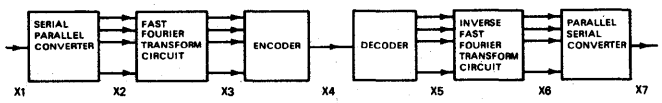
Figure 1—Simple voice PCM system

together with its COMSL description. The correspondence between the block attributes shown in the COMSL program and those specified in the block diagram can be seen by referring to Table I. It should be noted that the input files normally used in the simulation of voice communication systems contain speech data sampled at $8K$ Hz and quantized to 15 bits, i.e., max $|XIN| = 16384$. In this example the amplitude attributes of CMPRS1, QUANT1, CNOISE1, and EXPND1 are assigned a default value of 16384. The output of the simulation consists of a 20-second voice sample "processed" by the simulated system together with a tabulation of the signal-to-noise ratio computed on a second-by-second basis. In addition, the impulse and frequency response of the two filters in the system are printed.

A second example illustrating the use of the language is shown in Figure 2. The system shown is an experimental voice data rate compression system described in reference 2. This example illustrates the ease with which user defined blocks can be added to augment the basic set of function blocks. In this example, ENCODE and DECODE are standard Fortran subroutines. Since the input to the encoder consists of 64 parallel paths, the variable $X3$, as well as $X4$ and $X5$, must be dimensioned accordingly within the Fortran subroutines ENCODE and DECODE. The output from this simulation consists of a 30-second voice sample "processed" by the system as well as an 129 point estimate of the power spectrum of the "error" voltage.

IMPLEMENTATION OF THE LANGUAGE

A modular translator has been designed and a preliminary working version of it implemented under




```
INPUT ON FILE2
OUTPUT ON NEWFILE
RUN TIME = 30 SECONDS
BLOCK DEFINITION
        SPCNV (1, 64)
        FFT (1, 64)
        IFFT (1, 64)
        PSCNV (1, 64)
SYSTEM DEFINITION
        X1 = INPUT (FILE2)
        X2 = SPCNV1 (X1)
        X3 = FFT1 (X2)
        CALL ENCODE (X3, X4)
        CALL DECODE (X4, X5)
        X6 = IFFT1 (X5)
        X7 = PSCNV1 (X6)
        COMPUTE PWRSP2 (X7–X1, 128)
        OUTPUT (NEWFILE) = X7
END SIMULATION
        User defined subroutines ENCODE and DECODE
```

Figure 2—An experimental voice data rate compression system

OS/360 on an IBM 360/65. As of April, 1970, approximately two-thirds of the system blocks shown in Table I have been implemented, including FILTR, and preliminary versions of CXFRM and DXFRM.

The existing COMSL translator was written in IBM System/360 Fortran IV and compiled using the *G* compiler. It accepts a valid COMSL program and generates a corresponding Fortran program which is then processed in the normal fashion by the Fortran compiler. While the translator is to some extent machine dependent, care was taken to minimize that dependence. The implemented translator is approximately 60 K bytes in length. The Fortran code generated by the translator is roughly as efficient as that which could be generated by a careful Fortran programmer. As was mentioned earlier, the use of function or subroutine calls in the code generated by the translator has been avoided almost entirely except for certain initial computations. This was necessary because the relatively high sampling rate, 8kHz, for voice communication system simulation results in fairly lengthy execution times for even moderately complicated systems. As a general rule, subprogram calls have been avoided if the linkage time would represent more than five percent of the total computation time for a particular block.

REFERENCES

1 W K PRATT T KANE H C ANDREWS
Hadamard transform image coding
Proceedings of the IEEE Volume 57 pp 58-68 January 1969

2 G S ROBINSON R L GRANGER
Fast Fourier transform speech compression
Proceedings of the 1970 International Conference on Communications (ICC 70) San Francisco June 8-10 1970

3 A V OPPENHEIM R W SCHAFER
T G STOCKHAM
Nonlinear filtering of multiplied and convolved signals
Proceedings of the IEEE Volume 56 Number 8 August 1968 pp 1264-1291

4 B J KARAFIN
A sampled-data system simulation language
System Analysis by Digital Computer edited by F F Kuo and J F Kaiser John Wiley 1966 pp 286-314

5 R B BLACKMAN J W TUKEY
The measurement of power spectra
Dover Publications Inc New York 1958 pp 52-54

6 P D WELCH
The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms
IEEE Transactions on Audio and Electroacoustics Volume AU-15 Number 2 June 1967 pp 70-73

7 R M GOLDEN
Digital filter synthesis by sampled-data transformation
IEEE Transactions on Audio and Electroacoustics Volume AU-16 Number 3 September 1968 pp 321-329

8 C M RADER
On digital filtering
IEEE Transactions on Audio and Electroacoustics Volume AU-16 Number 3 September 1968 pp 303-314

CyberLogic—A new system for computer control

by G. R. TRIMBLE, JR.

Penta Computer Associates, Inc.
New York, New York

and

D. A. BAVLY

Penta Computer Associates, Inc.
Dallas, Texas

INTRODUCTION

The rapid advance of technology during recent years has made possible the development and use of very complex manufacturing processes. A prime example of such a process is a modern oil refinery in which there are actually many processes operating concurrently. The control of such a process requires the monitoring of pressures, temperatures, flow rates, and other such variables at many points within the refinery. Based on an analysis of the variables sampled, the process is controlled by adjusting flow rates, opening and closing valves, and recording the actions taken for further analysis and evaluation of the results of such actions.

A more restricted example of such a process is an experiment performed in a research laboratory in which a limited number of variables must be monitored and only a few valves, switches, and other parameters must be controlled. A bio-medical analysis or monitoring system is another example of a relatively simple process.

In all of these processes, control is based on an analysis of the variables monitored. Actions are taken to adjust the parameters of the system to make its performance meet the required objectives. The loop can be closed by having people monitor the variables and manually adjust the parameters based on the values of the variables, or the loop can be closed by having a computer monitor the variables and automatically adjust parameters based on its analysis of these values.

In a laboratory experiment, it is common to have a technician monitor the experiment continuously. When some variable exceeds a prescribed limit, he must take action to adjust the system to bring the variable back within the specified range of permissible values. In order to evaluate the experiment later, it is necessary that the technician record the status of the system at the time that he took the action and indicate what action he took as well as the time the action was taken.

CYBERLOGIC OBJECTIVES

CyberLogic is designed to provide closed-loop control and eliminate the need for a man to continuously monitor the process being controlled. A computer is used by CyberLogic to monitor the process, analyze values of the variables, and determine whether some action should be taken. The conditions existing within the process being controlled are continuously analyzed and action taken as a result of this analysis to adjust the process variables and maintain them within the required ranges.

The man is not excluded from the control of the process, however, and CyberLogic includes facilities through which he can exercise control over the process by making modifications to the specifications defining the system as well as being apprised of the status of the system when particular events occur.

On-line control of a process by CyberLogic provides many advantages which are not possible with manual control. One of these is repeatability. With manual control, the performance of the system is dependent upon the reaction time of the human being, who must recognize the conditions having reached a point at which some actions is required, then deciding what action is necessary, and finally taking the action. If the same conditions arise twice, the reaction of the

human being may be such that the second occurrence is not exactly a duplicate of the first. With computer control, using CyberLogic, repeatability is obtained with a high degree of precision, since the system will recognize the conditions immediately and provide the response necessary for the situation.

A secondary by-product of the use of CyberLogic is the rapid response of the computer to changes in the system. The manual system depends on human reaction time, and CyberLogic can respond much more rapidly in most instances than the human being can respond.

The analysis of a process and evaluation of what has happened is dependent upon an accurate record of exactly what happened. CyberLogic provides an unbiased and accurate record of the system conditions and actions taken. It does not depend upon a human operator to write down the time at which some event occurred and the action taken when it occurred, but rather CyberLogic will always record the time as well as the conditions and the actions which are associated with the event. As a result of having accurate records, it is possible to do a comprehensive analysis of what happened during the execution of a process and obtain a more accurate analysis as a result.

CyberLogic is based on using decision tables to define the process to be controlled. These decision tables define precisely every condition which can occur in the system and specify what action is to be performed when these specific conditions are met. The decision table language is oriented towards the types of functions which are performed in a simple process control system and permit a technician who is not a computer programmer to define the process in his own terminology. Thus, this simple language is oriented towards the user and he can express himself in terms which are natural to him in defining the process.

One consequence of using decision tables is that it forces a complete and comprehensive analysis of the process and a definition of all of the possible conditions which can occur. As a result, a much better understanding of the process is derived, as well as avoiding the possibility of overlooking conditions which normal analysis and programming procedures would possibly miss.

Many process control systems are not fixed, but vary as new instruments and modifications to the process are introduced. It is desirable that the process be defined and set up in a minimum amount of time. In addition, modifications to the process, once it has been defined, should be easily and quickly made.

For example, the use of a computer controlling a laboratory experiment requires that the experiment be defined and the experiment be set up and run. Using standard assembly language or FORTRAN language programming, this can be a very time-consuming and exasperating job. The decision table language used with CyberLogic is oriented specifically towards the types of functions performed in a process control system and will greatly facilitate the set-up of the experiment.

After an experiment has been running for a few days or a few weeks, it is desirable to use the computer for control of another different experiment. It is relatively easy to define and check out a new experiment using the decision table language with the CyberLogic system, so that conversion of the use of the computer to the new experiment can be quickly and easily accomplished. Thus, the conversion to a new experiment is not a traumatic process, but one which can be performed with great ease and minimize the wasted time of the computer between experiments.

Once a process or a laboratory experiment has been defined by the decision table language within CyberLogic, and the system set up and the process running, changes will frequently be made to the system. CyberLogic includes an on-line language which permits the user of the system to make modifications to the system and redefine the process for controlling the system to a limited extent. This eliminates the need for making up a new set of decision tables and compiling them to generate a new control system. Once these modifications have been made through use of the on-line language and the system checked out, the user can go back to the original source decision table language and redefine the system completely.

In general the objectives of the CyberLogic system are to provide a flexible, easy-to-use system for controlling a wide variety of processes, which is easy to set up and capable of being modified once the process has been established and is operating. The decision table approach has made it possible to meet these objectives and provide a system which will find use in a wide variety of applications.

CYBERLOGIC OVERVIEW

CyberLogic is a general purpose hardware/software system which can be adapted to automate a wide variety of closed loop control systems. The hardware consists of a Redcor RC-70 computer, various peripheral equipment including a teletype and high-speed photoreader, and a flexible acquisition IO system which can be easily connected to various types of sensors, tranducers, A D converters, and switches. The software consists of an executive routine which provides overall supervisory control of the system and two language translators. The first language, with both logical and arithmetic capabilities, permits the user to describe the

control procedures to be performed by the system. The second language operates on-line and permits the user to change parameters and monitor the control procedures during the course of operations.

Specification of the control procedures to be performed by the computer is accomplished through a descriptive language in which control variables are defined, conditions to be checked are specified, and actions to be performed when these conditions occur are described. The language provides facilities to define sources and destinations of input and output quantities, associate interrupt processing routines with hardware levels, specify sampling sequences and rates, define time dependent events, and specify standard subroutines to incorporate into the system. A translator program converts these language statements into a compact tabular format. These compact tables provide the data base for interpretive execution of the control procedures described by the user.

While the control procedures are being executed, the user has a facility to temporarily shutdown the system, make limited modifications to the control procedures previously defined, and restart the control procedures without performing retranslation. This facility is provided by an on-line language which resides in memory with the executive. The on-line language permits the user to look at or modify the values of control variables, to delete or change the order of condition or action statements, or to substitute the occurrence of one control variable for another in any condition or action. This facility is useful in debugging control procedures and in varying parameters of the procedure within a very short response time.

The major features provided by the CyberLogic System are:

1. A computer-independent control-oriented language which describes the sequence and timing of control procedures to be applied to a physical system.
2. The representation of such procedures by decision tables.
3. The translation of decision tables to a canonical form to be interpreted by a real-time control system which drives the physical process.
4. The ability to schedule the interpretation of a decision table, or a sequence of decision tables, at precise intervals of time, either once or periodically.
5. The ability to specify continuous sampling of variables attached to an Analog Multiplexor and Converter asynchronous to the decision table interpretation mechanism.
6. The ability to make available the latest values of both analog and digital input variables at the beginning of the interpretation of a decision table, or sequence of tables, and to keep a consistent set of values for these variables until interpretation is completed.
7. The ability to send out a digital signal (either of two voltage levels), or to generate a square waveform which alternates between the two voltage levels at periodic intervals.
8. The use of two separate time units; a fine resolution for updating the system clock and transmitting output waveforms, and a coarser resolution, expressed as a multiple of the fine resolution, for execution of decision tables.
9. The ability to associate critical decision tables with hardware interrupts, and the ability to either return to the interrupted table, or to initiate a new control path upon completion of the critical procedure.
10. The ability to specify a decision table for a master shutdown procedure which may be initiated by the normal scheduling mechanism, the detection of an inconsistent situation by the system software, or by the initiation of the "panic" interrupt.
11. The ability to make limited modifications of a commonly occurring nature to the tables by on-line commands, without recourse to complete retranslation.
12. The ability to specify decision tables to bring the control system to a quiescent state prior to the specifications of on-line language commands, and the ability to specify a decision table to bring the system back to an operational state following such specifications.

The flow of the CyberLogic system showing the relationship between the various hardware and software components is illustrated in Figure one.

CYBERLOGIC LANGUAGE

CyberLogic provides a higher order language approach to the problems of computer control. It enables the control system engineer to design his control hardware and software simultaneously. In order to describe the control software, he needs to define all his control variables and specify the effects of their values upon the rest of the system. This is done in the form of decision tables by specifying the conditions to be satisfied by variables either singly or in relation to one another, and actions to be performed when various combinations of conditions are satisfied. The statements

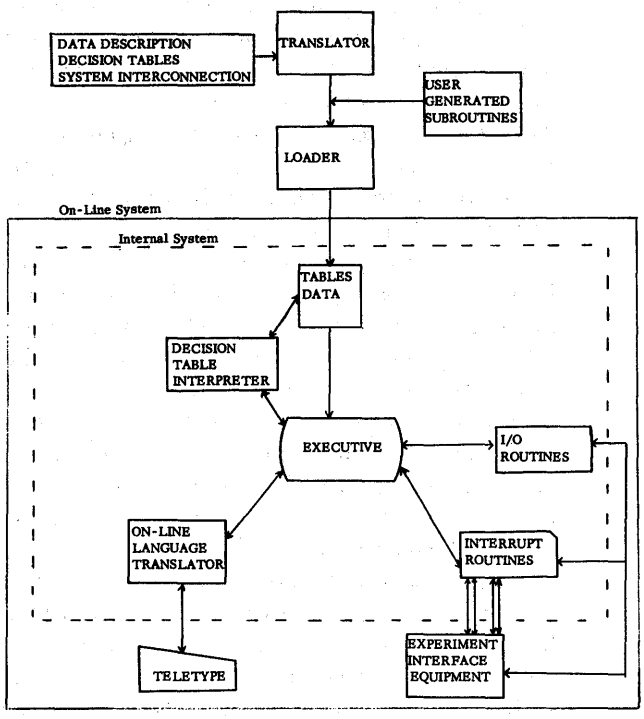


Figure 1—Component overview

which make up this language are prepared on a form and then are punched on paper tape. The paper tape thus produced is fed into a translator program which converts these language specifications to compact lists of binary information which will be used to drive the control process. These lists are interpreted by a computer program which operates in an On-Line Real Time environment and, under control of these lists, samples variables, uses the values of the variables to evaluate conditions, and the outcome of the conditions to determine which actions need to be performed at the present time, and causes the computer to perform the necessary actions to control the system.

Some of the more significant features that are in the language are the following:

1. The ability to define control variables as either input, output or working variables and from then on refer to them by a symbolic name.
2. The ability to perform arithmetic calculations involving control variables and constants.
3. The ability to dynamically select the analog inputs to be sampled at any time.
4. The ability to display the current value of any variable on the teletypewriter.
5. The ability to interpret more than one table using consistent values of a set of variables.
6. The ability to schedule another table for interpretation at a later time using the values of control variables at the later time. Furthermore, this procedure can be repeated a given number of times with a specified frequency.
7. The ability to generate a digital representation of a square pulse which raises and lowers the voltage level at specified frequencies.
8. The ability to escape to FORTRAN written sub-routines when requirements beyond the calculational facility of CyberLogic are required.
9. The ability to cancel pending requests for execution of tables that have built up in the system.

The descriptive language required to describe control procedures is divided into three parts. The first part of the language describes the control variables associated with the process. The second part of the language describes the conditions to be checked and the actions to be performed when these conditions are satisfied. These conditions and actions are represented in the form of decision tables. The third part of the language describes the connection of hardware interrupt levels to particular decision tables that have already been defined in the second part.

DATA DESCRIPTORS

A "variable" is the basic processable unit of information within the system. A variable may be either an analog input with up to 16 bits of precision, a single bit discrete variable, or a two-word floating point internal variable. It is referred to using a name consisting of up to four alphanumerical characters, and is converted to a two-word floating point quantity for uniformity of processing.

Data Description Statements define the variable names and the characteristics of the variables. Thus, once having described a variable through a Data Description Statement, the user need no longer concern himself with its characteristics, but uses the symbolic variables name to define the testing, processing, and input or output functions associated with the variable. The format of a Data Description Statement is

NAME, LENGTH, TYPE

for an internal variable;

NAME, LENGTH, TYPE, I/O UNIT, CHANNEL

for an analog input variable; or

NAME, LENGTH, TYPE, I/O UNIT, BIT #

for a digital input or output variable.

In summary, a NAME is used to refer to a variable which is a single bit within a digital input word, a single bit within a digital output word, a single word on an analog multiplexor channel, or an internal variable. The size and physical location of the variable are specified in the Data Description and the NAME is used to refer to the variable in the various processing statements.

DECISION TABLES

The conditions to be checked and the actions to be performed when conditions are satisfied are represented by decision tables. The basic form of a decision table is indicated in Figure 2. The upper left portion of the table is called the Condition Stub, the lower left portion is called the Action Stub, the upper right is called the Condition Entry and the lower right is called the Action Entry. Each column in the right side of the table is called a Rule.

The various conditions to be examined are defined in the Condition Stub. For each condition, the entries made in the Rules Columns for the corresponding condition are Y, N, and -, to denote "Condition True", "Condition Not True" or "Condition Irrelevant" respectively. Each distinct set of condition values (condition true or condition false) for the various conditions in the table results in the selection of a rule. For each rule, an action, or set of actions, is to be performed. An X in the box for a particular Rule/Action combination indicates that the action is to be performed when the conditions defining that rule are met. A zero in the box for a particular Rule/Action indicates that the action is not to be performed.

When a particular decision table is being interpreted, the conditions are evaluated to determine which rule, or rules, are true. If more than one rule is true, the leftmost one in the table will be selected. The ordering of the conditions in the table is irrelevant. When a rule is found for which the various combinations of conditions are met, the actions which are to be performed, as indicated by X's, are executed in sequence starting with the top-most action having an X and proceeding downward to the last action having an X.

TABLE NAMES

Decision tables are given unique names by prefacing the description of each table by a table name statement. The format of this statement is

TABLE table-name, parameter

where table-name is a string of four or less characters

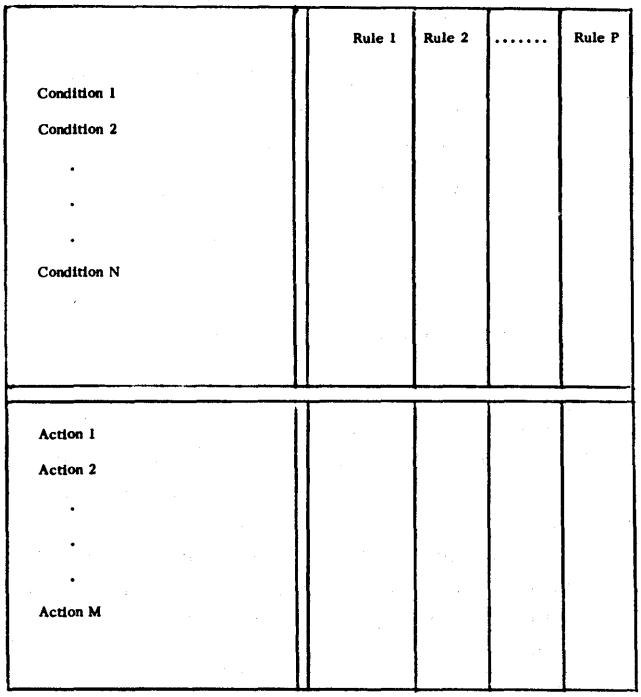


Figure 2—Decision table format

used to identify a decision table. Because of requirements of other language statements, certain combinations of characters must be reserved and cannot be used as table names.

CONDITIONS

The conditions which can appear in the Condition Stub may have one of four forms:

NAME1 R CONSTANT

NAME1 R NAME3

NAME1 ± NAME2 R CONSTANT

NAME1 ± NAME2 R NAME3

Where NAME 1, NAME 2 and NAME 3 are the symbolic names of variables, CONSTANT is a constant value and R is one of the six standard relational operators (GT, GE, LT, LE, EQ, WE).

ACTIONS

Several types of elements may be used in the Action Stub. The following descriptions of the Action Stubs

indicate the various actions which can be initiated when the conditions defining a rule are met.

1. *Write Action*: the WRITE Action causes a line of output to be sent to the ASR-33. The format of the WRITE stub is

   WRITE literal-1 name literal-2

2. *Set Action*: the format of the SET Action is

   SET name TO expression

   The general form of an expression is:

   X1 OP1 fn1 (name-1) OP2 X2 OP3 fn2 (name-2)

   where X1 and X2 are either constants or the names of variables, OP1 and OP3 are one of the operators * (multiply) or / (divide), name-1 and name-2 are the names of variables, OP2 is one of the operators + (add) or — (subtract), and fn1 and fn2 are one of the functions SIN (sine), COS (cosine), TAN (tangent), SQT (square root), LOG (logarithm), or EXP (exponential).
3. *Count Action*: The format of the COUNT Action is

   COUNT name

   Each time the COUNT Action is executed, a one is added to the variable. This action provides the capability of counting the number of occurrences of some event.
4. *Go To Action*: The GO TO Action provides the capability of directing the Table Interpreter to give control immediately to another table. The format of the GO TO stub is

   GO TO table-name

   where table-name is the name of the table to which control is given.
5. *Call Action*: The CALL Action provides a means for calling subroutines which have been included in the system. Subroutines may be written in symbolic assembly or Fortran language, compiled independently, and then incorporated into the control system. The form of the CALL stub is

   CALL subname (name-1, name-2, . . ., name-6)

6. *Execute Action*: The EXECUTE Action is the means of initiating the execution of other tables asynchronously with the current table. The format of the EXECUTE stub is

   EXECUTE table-name TIMES nn
   PERIOD time-1 DELAYED time-2

   where table-name is the name of the table to be executed, nn is the number of times the table is to be executed, time-1 specifies the period at which the table is to be executed, and time-2 is the time interval to delay before the first execution of the table.
7. *Output Action*: The OUTPUT Action causes the immediate output of a digital variable. The format of the OUTPUT stub is:

   OUTPUT name

8. *Pulse Action*: The PULSE Action provides the capability of periodically generating a digital square wave. The format of the PULSE stub is:

   PULSE name TIMES nn PERIOD time-1
   WIDTH time-2 DELAYED time-3

   where name is the name of the variable to be read out, nn is the number of times the pulse is to be generated, time-1 specifies the period of the pulse, time-2 specifies the pulse width, and time-3 is the time interval to delay before generating the first pulse.
9. *Scan On Action*: The SCAN ON Action provides the ability to activate sampling of the analog inputs named. The format of the SCAN ON stub is:

   SCAN ON name-1, name-2, . . ., name-6

10. *Scan Off Action*: The SCAN OFF Action provides the ability to inhibit samplings of the analog inputs named. The format of the SCAN OFF stub is:

    SCAN OFF name-1, name-2, . . ., name-6

11. *Erase Action*: The ERASE Action provides the ability to cancel all pending EXECUTE and PULSE actions. Upon recognition of the ERASE Action, the system will erase from its memory all requests to interpret tables that were generated as a result of previously encountered EXECUTE Actions, and all requests to pulse digital output that were generated by previously encountered PULSE actions. The format of the ERASE stub is:

    ERASE

SYSTEM INTERCONNECTION

The purpose of system interconnection is to trigger the execution of a decision table based upon the occurrence of a particular hardware interrupt. The

system interconnection statement associates a hardware interrupt level with a decision table, such that whenever that interrupt occurs, control is assigned to the Decision Table Interpreter, which causes the associated decision table, or string of decision tables joined by GO TO Actions, to be executed. The execution of the corresponding table occurs immediately, interrupting whatever processing is currently in progress, with one exception. If the current processing is due to another interrupt, it is allowed to finish, and the table connected to the interrupt is serviced immediately thereafter. The format of a system interconnection statement is:

table-name, level

where table-name is the name of the table to be executed when an interrupt is received on a particular hardware level.

ON-LINE LANGUAGE

While the system is running, the user may wish to modify some parameters without going through the entire process of translating from the source language (decision tables and other specifications) to the tables used to perform control of the system. The On-Line Language provides the capability of performing such modifications.

In order to perform these modifications without causing instabilities in the control process introduced by altering tables which may be executing or altering variables which may be used by currently executing tables, the user must bring the system to a quiescent state before entering On-Line Language statements. The user must preface his On-Line Language statements with an EXECUTE command, which causes the specified decision table to be executed immediately. The purpose of this table is to bring the system into the quiescent state. Similarly, the user must follow his On-Line Language statements with another EXECUTE command, which specifies a table whose purpose is to restore the system to an operational state. The format of the EXECUTE command is:

EXECUTE table-name

The following descriptions of the On-Line Language Statements show the modifications that may be made through the On-Line Language.

1. *Set Statement*: The SET Statement alters the value of an internal variable to a constant specified in the on-line command. The format of the SET Statement is:

   SET name TO constant

2. *Display Statement*: The DISPLAY statement causes the current value of any variable to be typed out on the ASR-33. The format of the DISPLAY statement is:

   DISPLAY name

3. *Delete Statement*: The DELETE statement causes a single condition, action, or rule to be removed from the specified table. The format of the DELETE Statement is:

   DELETE CONDITION nn IN table-name

   or

   DELETE ACTION nn IN table-name

   or

   DELETE RULE nn IN table-name

4. *Move Statement*: The MOVE Statement causes a rule or an action to be moved from its present position in a decision table, and be placed directly before another rule, or action, in the same decision table. All rules (or actions) between the specified ones are adjusted right or left, as the case may be, by one position. The format of the MOVE Statement is

   MOVE RULE ii BEFORE jj IN table-name

   or

   MOVE ACTION ii BEFORE jj IN table-name

5. *Swap Statement*: The SWAP Statement causes a rule or an action to be interchanged with another rule, or action, in the same decision table. All other rules (or actions) besides these two are unaffected. The format of the SWAP Statement is:

   SWAP RULE ii WITH jj IN table-name

   or

   SWAP ACTION ii WITH jj IN table-name

6. *Change Statement*: The CHANGE Statement causes the name of a variable, used in a condition or action stub, to be changed to another. The format of the CHANGE Statement is

   CHANGE name-1 IN CONDITION
   nn OF table-name TO name-2

   or

   CHANGE name-1 IN ACTION
   nn OF table-name TO name-2

7. *Replace Statement*: The REPLACE Statement causes the value of a condition or action entry to be changed to a specified value. The format of the REPLACE Statement is:

    REPLACE CONDITION ii RULE jj
    IN table-name BY symbol-1

    or

    REPLACE ACTION ii RULE jj IN
    table-name BY symbol-2

    In the case of a condition, the value is set to symbol-1, which may be Y, N or—(dash). In the case of an action, the value is set to symbol-2, which may be X or 0 (zero).

ACKNOWLEDGMENTS

The system described in this paper was developed with the cooperation of the Mobil Research and Development Corporation. We are particularly indebted to the assistance and support provided by Drs. John P. Heller and Steven J. Bomba of Mobil Research and Development Corporation.

REFERENCES

1 G W OERTER
A new implementation of decision tables for a process control language
IEEE Trans on Ind Elec and Control Inst Vol IECI-15 No 2 December 1968 pp 57-61

2 E WEISS
Programming for better control
Control Engineering December 1968 p 84

A model for traffic simulation and a simulation language for the general transportation problem*

by ROGER S. WALKER, BAXTER F. WOMACK, and C. E. LEE

The University of Texas at Austin
Austin, Texas

INTRODUCTION

Widespread interest in better traffic control techniques has resulted in the past few years because of the rapidly increasing numbers of vehicles in American cities. This paper describes a modeling technique which promises to be useful for traffic simulation and, thereby, aiding development of more advanced on-line signal control systems. For this method, simulation is accomplished by stepping individual vehicles through a traffic network system in accordance with driver responses to changing traffic and environmental conditions.[1,2] During each time increment, a driver's response such as speed up, slow down, etc., is computed from various input conditions normally available to a driver. These inputs might include "is vehicle ahead slowing down," "is vehicle ahead stopped," etc. The vehicle's new positions, velocities, etc., are then computed and the next vehicle considered. Thus, the driver's responses to various input conditions are preprogrammed for the various driver types and intersection conditions. Vehicle arrival rates, driver characteristics, and vehicle flow paths may be functions of various statistical distributions. However, by also allowing preprogrammed driver actions, a desirable mixture between statistical and heuristic operations is obtained. Since the simulation can be used to predict traffic conditions, on-line simulation can then possibly be used as criteria for the selections of traffic signal control patterns or other control techniques.

The modeling method described in this paper can be used for implementing the step-through simulation technique. To implement this method, Boolean Algebra and modern system programming techniques are used. Traffic flow through a hypothetical intersection controlled by four-way stop signs is used to illustrate the method and to indicate the relative efficiency of a computer program written on the CDC 6600 to implement this method.

A simulation language, Traffic Network Simulator (TRANS), is described.[3] TRANS provides the needed programming flexibility to develop the general k intersection, n lane network system. The program described in this paper for modeling the four-way stop intersection is not written in this language. However, the programming concepts used in writing this program were incorporated in developing TRANS.

The majority of traffic simulation models proposed up to now have been developed around probability and queueing theory. Descriptions of many of these models are available in the literature. The model described in this paper deviates somewhat from this standard queueing theory approach and can be implemented for various and multiple intersection types with minimum memory and computer time requirements.

The first part of this paper describes the step-through modeling technique followed by a discussion of some results using the model. Finally, the simulation language TRANS which is currently being used for modeling a general k intersection, n lane network, system, is briefly described.

TRAFFIC NETWORK MODEL

As briefly discussed above the modeling technique described in this paper attempts to simulate or model the actions taken by the driver of each vehicle in a system as the vehicles move within a street network. This is accomplished by first prespecifying what the driver's actions will be in response to several sets of inputs that are available to him concerning the roadway and traffic situation. The driver's actions are then determined for each particular combination and for each increment of time. Thus, by stepping each driver

* This research was supported in part by the Joint Services Electronics Program and Department of Civil Engineering at the University of Texas at Austin.

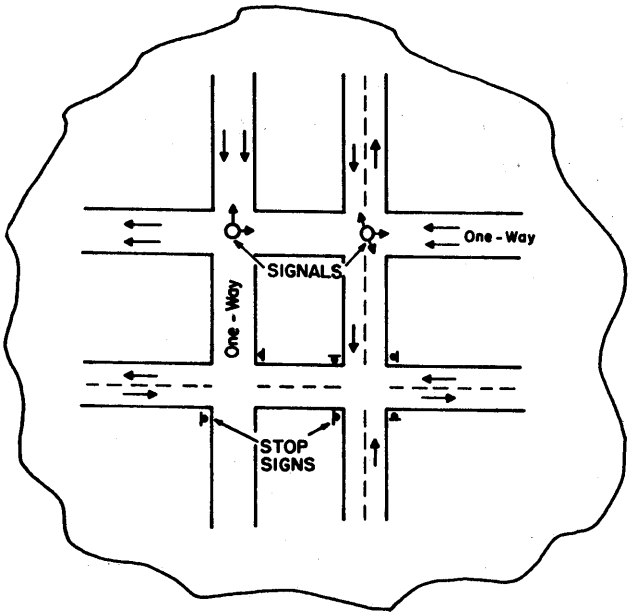


Figure 1—Traffic network system

through the traffic network system, network traffic flow can be realized.

To formulate this technique the following approach was taken: first, a traffic network system was defined as a network of various types of traffic intersections (i.e., four-way stops, two-way stops, signal-controlled, etc.), each with one or more lanes per approach (see Figure 1). Next, the set of n inputs required by the driver while stepping through the system and the set of m driver responses were defined as the driver input set and driver response set, respectively. The set of decision response functions was defined as that set of functions which associates with each of the possible 2^n evaluations of the input set a value from the driver response set; see Figure 2. There are 2^n possible evaluations of the input set, as each input variable of the input set has only two possible values, i.e., yes or no. A second set of functions denoted as the intersection functions associates a particular set of driver inputs and driver responses to the physical characteristics of an intersection type. A set of intersection types completely describes a network system.

Some typical vehicle inputs might be: present speed, desired speed, and questions such as "Is the vehicle at the intersection," or "Is the vehicle stopped." Some typical driver responses are: stop vehicle, increase speed, decrease speed, turn right, turn left, etc. Since a particular driver response is determined by a set of yes/no driver input combinations, the response function may be expressed in a standard computer flow chart form. The network system could thus be expressed by all such intersection flow charts. Because of the way the model variables were structured, the intersection and system network functions form a Two-valued Boolean Algebra. Thus, for example, each intersection type may be expressed as a sum of products (or its dual, a product of sums) of the driver input variables. A typical decision response function might be:

$$A = \bar{x}_1\bar{x}_2 + x_3x_4$$

where

x_1 = no vehicle is in close proximity

x_2 = not at desired speed

x_3 = vehicle at intersection

x_4 = first vehicle at intersection

A = accelerate vehicle

By expressing these functions in Boolean Algebra, the standard rules regarding Boolean simplification apply.

Figure 3 depicts symbolically the traffic network model. As may be noted in this figure, vehicles enter the network system at a specified speed, lane approach, and time. These dynamic vehicle characteristics are generated from random deviates from prespecified distributions. The appropriate response is then determined; that is, accelerate, decelerate, etc., via the driver response functions. Once the proper responses have been determined, the vehicle's new position, speed, etc., are found. This operation continues as the vehicle is stepped through the network in accordance with the prespecified or statistically generated flow. When the vehicle reaches the edge of the network system, it is logged out of the model. By observing this process for selected vehicle speeds, directions, and time combinations, traffic flow through the network model is realized.

In the section which follows, the four-way stop, single intersection network will be considered.

Four-way stop single intersection network

Consider a four-way stop, single lane, single intersection network as depicted in Figure 4. For the

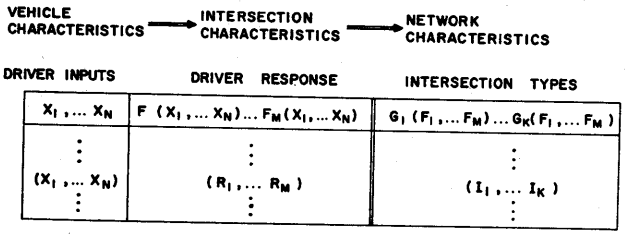


Figure 2—Table of model combinations

particular system shown, vehicles enter the system 450 feet before reaching the near side of the intersection and exit 450 feet beyond this point. These dimensions are, of course, completely arbitrary and may be selected to describe a particular intersection of interest. The intersection accommodates one lane for each of the four directions—North, South, East, West—with stop signs on each approach lane at the intersection.

The four-way stop intersection, as the name implies, requires that all vehicles stop before entering the intersection. The decision to proceed into the intersection is then based on the particular traffic laws of the city or state where the intersection is located. The rules or laws selected for the four-way stop model being described are those recommended by the National Committee on Uniform Traffic Laws and Ordinances, and are similar to those applicable in the state of Texas. These rules may be stated as follows:[4]

a. "When two vehicles approach or enter an intersection from different highways at approximately the same time, the driver of the vehicle on the left shall yield the right-of-way to the vehicle on the right."
b. "The driver of a vehicle intending to turn left within an intersection or into an alley, private road or driveway shall yield the right-of-way to any vehicle approaching from the opposite direction which is within the intersection or so close thereto as to constitute an immediate hazard."
c. "Except when directed to proceed by a police officer or traffic-control signal, every driver of a vehicle approaching a stop intersection indicated by a stop sign shall stop at a clearly marked stop line, but if none, before entering the crosswalk on the near side of the intersection or, if none, then at the point nearest the intersecting roadway where the driver has a view of approaching traffic on the intersecting roadway before entering the intersection. After having stopped, the driver shall yield the right-of-way to any vehicle which has entered the intersection from another highway or which is approaching so closely on said highway as to constitute an immediate hazard during the time when such driver is moving across or within the intersection."

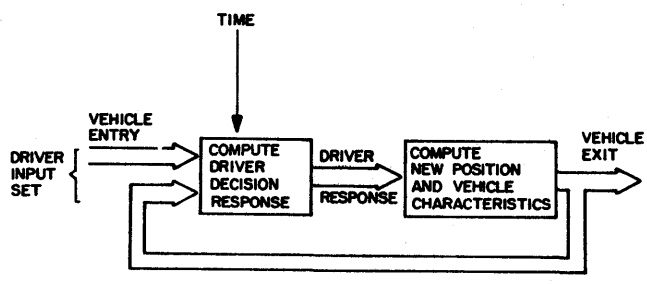


Figure 3—Traffic network model

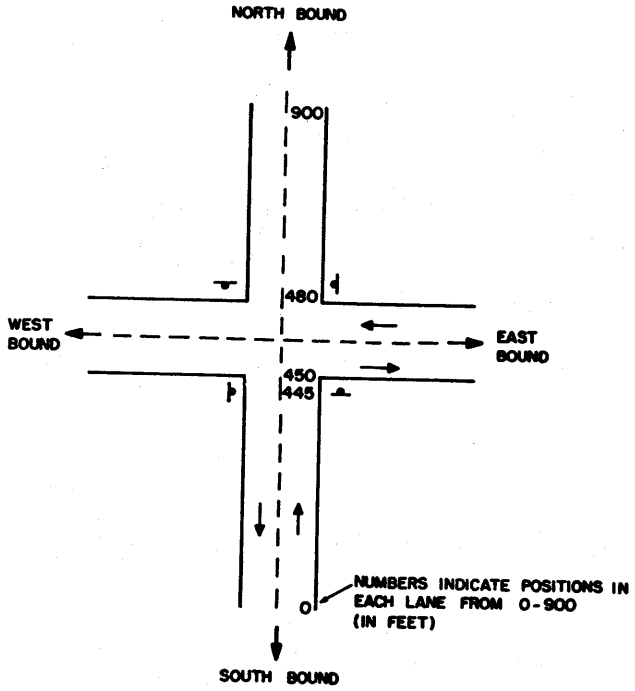


Figure 4—Four-way stop intersection

These rules are used in the model to govern the driver as he steps through the intersection. By flow-charting the driver's actions as he transverses the network system, the decision response flow chart is generated. In developing this chart, the vehicle input and response sets are defined. Figure 5 depicts the decision response flow chart for the four-way stop. Table I lists the necessary driver inputs and driver responses used by the driver as he transverses the network system. As noted, only three responses are actually required by the driver, namely, increase, decrease, or maintain speed.

To stop the vehicle, the vehicle speed is simply decreased until its speed is zero. Similarly, when the vehicle is stationary, movement is initiated by increasing velocity.

Turning movements are not considered responses by the model but part of the predetermined vehicle path. That is, when a vehicle enters the system, the driver destination has been prespecified as a left turn, right turn, or straight through movement. Then, as the vehicle is stepped through the network, if a turn is

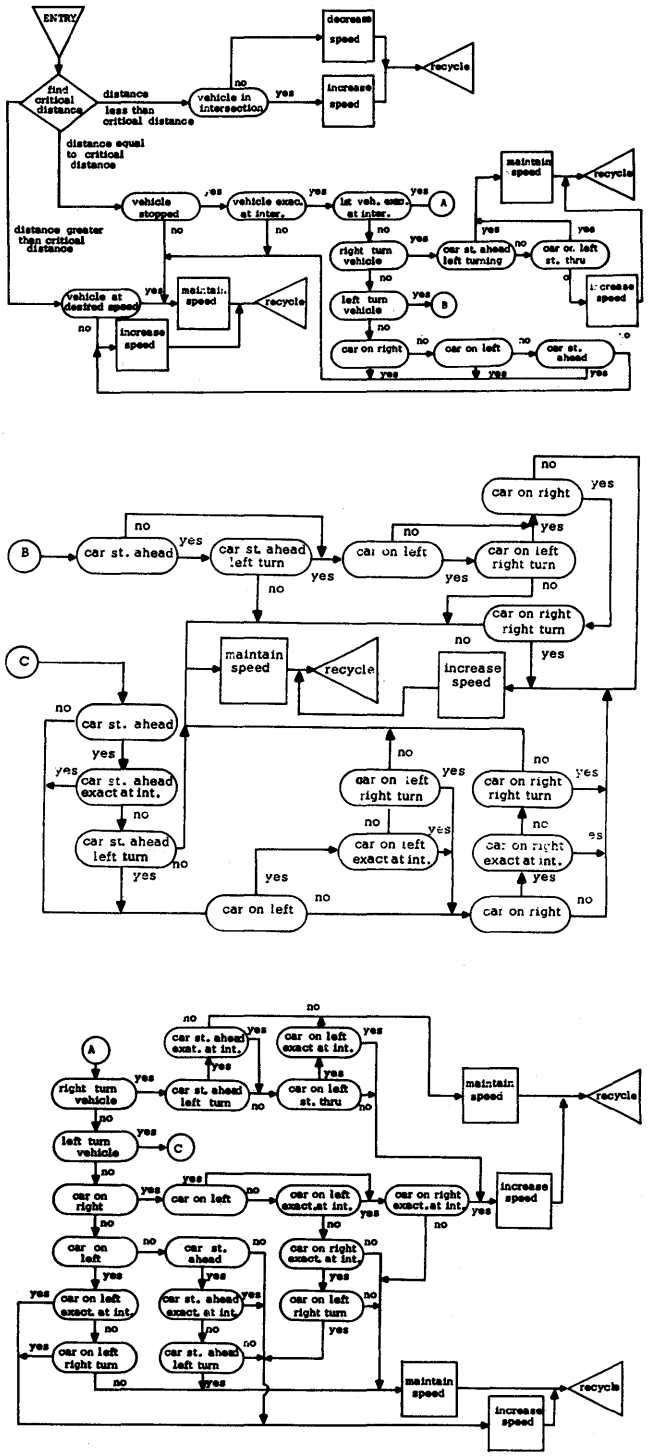


Figure 5—Four-way stop decision logic

specified for that vehicle, it is automatically initiated as he follows the appropriate movement flow path.

Also indicated in both the vehicle input set and the decision response flow chart are the inputs, vehicle less than, greater than, or equal to critical distance. The critical distance criterion is perhaps the most sensitive of all vehicle characteristics and certainly one of the more difficult ones to model realistically. The term critical distance as used in the model is defined as that distance which is required to stop or slow the vehicle sufficiently to prevent it from colliding with an adjacent vehicle or object. Thus, when the input, vehicle less than critical distance, is true, the model should slow the vehicle. The amount of slowing or decelerating is important since insufficient deceleration will result in a collision, and too much might result in oscillation about the proper critical distance as the driver tends to over-correct.

Briefly, the critical distance function was obtained by first establishing a relation between acceptable deceleration rates and vehicle velocity. Once this function was found, it was then used to express the distance required to stop the vehicle as a function of the vehicle velocity.

Vehicle acceleration is also important although not as critical as deceleration since the risk of a collision is very small for reasonable rates of acceleration.

The model operation is as depicted symbolically in Figure 3. Descriptive characteristics for a set of vehicles is first generated by specifying for each vehicle

TABLE I—Vehicle Input and Response for Four-Way Stop

Driver Inputs	Driver Responses
Distance less than critical distance	Increase speed
Distance equal to critical distance	Decrease speed
Distance greater than critical distance	Maintain present speed
Vehicle in intersection	
Vehicle exactly at intersection	
First vehicle exactly at intersection	
Right turning vehicle	
Left turning vehicle	
Straight thru vehicle	
Vehicle at desired speed	
Vehicle less than desired speed	
Vehicle stopped	
Car on right	
Car on left	
Car straight ahead	
Car on right exactly at intersection	
Car on left exactly at intersection	
Car straight ahead exactly at intersection	
Car on right right turning	
Car on right left turning	
Car on right straight thru	
Car on left right turning	
Car on left left turning	
Car on left straight thru	
Car straight ahead right tuning	
Car straight ahead left turning	
Car straight ahead straight thru	

an entry time, speed, lane approach, and vehicle flow path. This set may be determined either statistically with characteristics such as turning direction or lanes biased according to some observed or preestablished probabilities or distributions, or by other such means. Then at the appropriate time each vechicle is queued into the system model. Upon entry, the vehicle joins the other vehicles already in the system and the proper vehicle input set is generated. For each time increment, each vehicle is sequentially examined, first determining the appropriate decision response from the vehicle input set in accordance with the decision response flow chart of Figure 5, and then, as directed by this response, updating the vehicle input set.

Multiple intersections

The network system described in the preceding section was for the four-way stop single intersection network. The procedure used to model the multiple intersection network of k intersection types is similar to that just described; that is, a set of k decisoin flow charts and the respective vehicle input and response sets are developed. Vehicle entry information, however, must include, in addition to the entry information previously described, the entry intersection and an expanded data flow path which specifies the intersection combination sequence. Then for the model operations, the vehicle is simply stepped from one intersection to the next as prescribed by the data flow path until it leaves the network system. Hence, the multiple intersection model is simply composed of multiple single intersection models or subsystems.

SIMULATION RESULTS

This section describes the use of the program for traffic flow simulation through a four-way stop intersection in which 100 randomly generated vehicles are entered into the system over a 23 minute time period. Of course, any desired vehicle input combination can be specified for traffic simulation studies. As may be noted in the summary information discussed later, the direction selection was biased so that more vehicles were selected for the northbound approach than for the other directions. This is realistic as in many cases one approach often accommodates more vehicles than the others. The 100 vehicles were entered into the system over a 23 minute time period. The first vehicle with an Ident of 1 entered the system at a time of 12.3 seconds on a westbound approach at a velocity of 6 miles per hour. Similarly, the last vehicle with an Ident of 100 entered the system at 1302.4 seconds on a northbound approach at 27 miles per hour. Table II provides some selected results of the simulation run. As noted, the first vehicle left the system at 54 seconds and was under 10 miles per hour for 13 seconds. Similarly, the last vehicle left the system at 1336 seconds. The model terminated at 1403 seconds, approximately 100 seconds after the last vehicle entered the system (as specified in the data cards). The last vehicle traveled under 10 miles per hour for only 9 seconds, because of its faster entry velocity. It exited on the westbound lane after turning left off the northbound approach at the intersection. The grand total printout provides a summary of the number of vehicles counted into the system, the average stop time delay, the average delay below 10 miles per hour, and average time the vehicles were in the system. This information, as may be noted, is given by lane approach and turning movements. The actual time the system was used, 1276 seconds, is also provided for intersection study purposes. The minimum stop time delay is for a one second period. This is program selectable, however, in accordance with the desired driver response time.

The real-time-to-computer-time ratio for this particular problem was approximately 60 to 1. By the nature of the program technique, computer time requirements are primarily dependent on the cycle time selected (0.5 second for this example) and the number of vehicles in the system at a given time. It is relatively insensitive to the number of intersections or to the intersection type. Program memory requirements are less than 8K.

TRANS, A TRAFFIC NETWORK SIMULATION LANGUAGE

This section will briefly describe the simulation language currently being used for modeling the k intersection, n lane network system. It should be noted that the use of this language is not limited to the traffic network problem but could be used in a variety of other simulation and control situations where particular actions are dependent on sets of logical decisions. Other such examples might include its use for modeling traffic parking problems, airport runway congestions, warehouse or goods distribution, waiting line problems, or missile range control countdown simulation for range scheduling purposes. TRANS is compiled under FORTRAN and permits the use of FORTRAN statements and subroutines.

Program variables and parameters used in TRANS are categorized according to the following classification:

Input Variables (driver input set)
Response Variables (driver response set)
Characteristic Parameters

The input variables are the logical two-valued variables used by the Boolean decision response functions for generating the response variables. The response variables are the two-valued variables that specify a particular driver response. The vehicle characteristic parameters describe the vehicle characteristic information needed for computing new position, speed, direction, etc.

The basic program structure of TRANS consists of five major subparts: the stack, the real-time cycle processor, the response function processor, the extract/repack processor, and the general utility processor.

The stack is a storage array used by TRANS for maintaining vehicle characteristics; that is, it contains the input variables and characteristic parameters for each vehicle. Since much of this information requires only a few bits (e.g., only two bits are needed for each logical input variable—yes, no, and don't care), this information is packed into each stack word. Then during simulation, the vehicle characteristics are extracted from the stack for each vehicle at the appropriate time.

The real-time processor subpart acts as the executive of the simulation process transferring control to the appropriate system and model subroutines. For example, this routine calls on the response function processor, the extract/repack processor, and the traffic model sub-

```
VEHICLE LEFT SYSTEM     IDENT=    1 ISI=   13 TURN TYPE= 0 DIRECTION= 3 TIME=  54.00000
TIME IN SYSTEM=  42.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=  13.00000

VEHICLE LEFT SYSTEM     IDENT=    2 ISI=   14 TURN TYPE= 1 DIRECTION= 0 TIME= 117.00000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=    3 ISI=   15 TURN TYPE= 1 DIRECTION= 3 TIME= 136.50000
TIME IN SYSTEM=  37.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM     IDENT=    4 ISI=   13 TURN TYPE= 0 DIRECTION= 0 TIME= 162.50000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=    5 ISI=   16 TURN TYPE= 0 DIRECTION= 2 TIME= 179.00000
TIME IN SYSTEM=  36.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM     IDENT=    6 ISI=   14 TURN TYPE= 3 DIRECTION= 1 TIME= 196.50000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=    7 ISI=   17 TURN TYPE= 1 DIRECTION= 0 TIME= 210.00000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=    8 ISI=   18 TURN TYPE= 0 DIRECTION= 0 TIME= 220.50000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=    9 ISI=   13 TURN TYPE= 0 DIRECTION= 3 TIME= 227.50000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=   10 ISI=   19 TURN TYPE= 0 DIRECTION= 2 TIME= 237.50000
TIME IN SYSTEM=  35.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=   11 ISI=   15 TURN TYPE= 0 DIRECTION= 2 TIME= 254.00000
TIME IN SYSTEM=  35.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM     IDENT=   12 ISI=   16 TURN TYPE= 0 DIRECTION= 2 TIME= 272.50000
TIME IN SYSTEM=  35.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM     IDENT=   13 ISI=   17 TURN TYPE= 3 DIRECTION= 3 TIME= 283.00000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM     IDENT=   14 ISI=   15 TURN TYPE= 1 DIRECTION= 2 TIME= 299.50000
TIME IN SYSTEM=  39.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=  11.00000

VEHICLE LEFT SYSTEM     IDENT=   15 ISI=   13 TURN TYPE= 1 DIRECTION= 0 TIME= 318.50000
TIME IN SYSTEM=  33.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000
```

TABLE II—Results of Computer Run, 100 Vehicles

```
VEHICLE LEFT SYSTEM    IDENT=   93 ISI=   14 TURN TYPE= 0 DIRECTION= 3 TIME=1269.00000
TIME IN SYSTEM=  36.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM    IDENT=   96 ISI=   17 TURN TYPE= 0 DIRECTION= 2 TIME=1275.00000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   6.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=  13.00000

VEHICLE LEFT SYSTEM    IDENT=   95 ISI=   19 TURN TYPE= 0 DIRECTION= 2 TIME=1279.50000
TIME IN SYSTEM=  33.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM    IDENT=   96 ISI=   15 TURN TYPE= 0 DIRECTION= 0 TIME=1291.50000
TIME IN SYSTEM=  35.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM    IDENT=   97 ISI=   18 TURN TYPE= 1 DIRECTION= 0 TIME=1296.50000
TIME IN SYSTEM=  37.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   8.00000

VEHICLE LEFT SYSTEM    IDENT=   98 ISI=   16 TURN TYPE= 3 DIRECTION= 3 TIME=1313.00000
TIME IN SYSTEM=  40.00000 STOP TIME DELAY=   1.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=  11.00000

VEHICLE LEFT SYSTEM    IDENT=   99 ISI=   17 TURN TYPE= 1 DIRECTION= 3 TIME=1324.00000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000

VEHICLE LEFT SYSTEM    IDENT=  100 ISI=   14 TURN TYPE= 3 DIRECTION= 3 TIME=1336.50000
TIME IN SYSTEM=  34.00000 STOP TIME DELAY=   2.00000  TIME DELAY FOR VEHICLES UNDER  10.00000MPH=   9.00000


 GRAND TOTAL SUMMARY INFORMATION FOR TIME =1403.00000
SYSTEM USE TIME=1276.00000


NORTHBOUND APPROACH
 STOP TIME DELAY...RT TURN=       1.4  LEFT TURN=       1.5  ST THRU=       1.8
TIME DELAY FOR    10.0  MPH=.... RT TURN=       9.8  LEFT TURN=       8.7  ST THRU=       9.2
TOTAL VEHICLES COUNTED=....RT TURN=    8  LEFT TURN=   15  ST THRU=   20  AVERAGE TIME IN SYSTEM=      35.4


EAST BOUND APPROACH
 STOP TIME DELAY...RT TURN=       1.0  LEFT TURN=       8.0  ST THRU=       1.5
TIME DELAY FOR    10.0  MPH=.... RT TURN=      11.0  LEFT TURN=      17.0  ST THRU=       8.5
TOTAL VEHICLES COUNTED=....RT TURN=    1  LEFT TURN=    1  ST THRU=    2  AVERAGE TIME IN SYSTEM=      38.2


SOUTHBOUND APPROACH
 STOP TIME DELAY...RT TURN=       1.4  LEFT TURN=       1.3  ST THRU=       2.1
TIME DELAY FOR    10.0  MPH=.... RT TURN=       8.6  LEFT TURN=       8.3  ST THRU=       9.1
TOTAL VEHICLES COUNTED=....RT TURN=    5  LEFT TURN=    3  ST THRU=   15  AVERAGE TIME IN SYSTEM=      35.3


WESTBOUND APPROACH
 STOP TIME DELAY...RT TURN=       1.5  LEFT TURN=       3.0  ST THRU=       2.1
TIME DELAY FOR    10.0  MPH=.... RT TURN=       9.7  LEFT TURN=      12.0  ST THRU=       9.9
TOTAL VEHICLES COUNTED=....RT TURN=   11  LEFT TURN=    5  ST THRU=   14  AVERAGE TIME IN SYSTEM=      36.7
```

TABLE II—Results of Computer Run, 100 Vehicles

routines after each stack entry during each real-time cycle.

The response function processor is used to compute the proper driver responses from the Boolean decision response functions and the input variables; thus, the processor examines all input combinations associated with a particular response, and if the proper combinations of inputs are true, the associated response is indicated. Since each decision response function is a Boolean function, the following technique is employed to process this response:

Each driver response is first expressed in a sum product form. The Boolean sum product functions are generated by TRANS from the decision response control statements used in describing each intersection. A set of computer words are then used to contain each product term where the bit positions of each variable are identical with those of the input variable words in

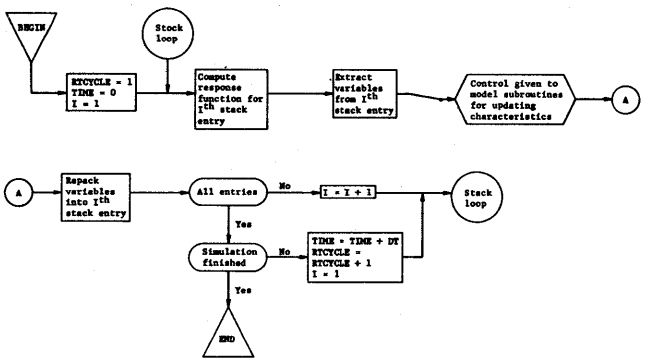


Figure 6—TRANS program flow

the stack. The set of all such word product terms then describes the response function and all such sets describe the intersection type. Each single product term or computer word group is ANDed with the input set. If the resulting ANDed word group is identical to the product term, the response is set true. Accordingly, if all terms are not true, the response is set false. This operation continues until all response functions have been examined.

The extract/repack processor is the processing routine used to extract each variable and parameter word from the stack as required by the traffic simulation subroutines. Once these variables and parameter words have been operated on and updated, the repack portion of this processor is called on to repack these words into the appropriate stack entries.

The general utility processor is the routine called on during simulation for processing the STACK entry assignment, stack entry releases, and variable request/replace control statements.

TRANS program flow is as indicated in Figure 6; as noted from this figure, the time is initially zero, and the real-time cycle set to one. Then for each time cycle all stack entries sets corresponding to vehicles are processed as follows:

The response function processor first computes the appropriate responses. The vehicle characteristic parameters and variables are then extracted for the simulation or model subroutines. Once these routines are completed, the characteristic variables are repacked into the stack entry and the next vehicle examined. This process continues until all stack entries have been examined at which time the real-time cycle is incremented and the process repeated.

Model subroutines have the options to be processed only once during model initializations (zero cycle tasks), once at the beginning of each real-time cycle (one cycle task), or for each stack entry during each real-time cycle (n cycle tasks).

Some of the more pertinent control statements in addition to those permitted by CDC 6600 FORTRAN are listed below.

Network Configuration Statements

INTER—defines intersection configuration
ILNK—defines network linkage

Vehicle Characteristic Statements

INPUT—declares input variables
RES—declares response variables
PAR—declares characteristic parameters
FUN—defines decision response functions
FUN INTER—relates response functions to intersection type

Model Subroutine Statements

SUB—model task subroutine declaration
ENDSUB—model task subroutine terminator
REQS—request stack entry assignment
RELQS—release stack entry assignment
REQUEST—request stack entry variable
REPLACE—replace stack entry variable
VEH—vehicle input declaration

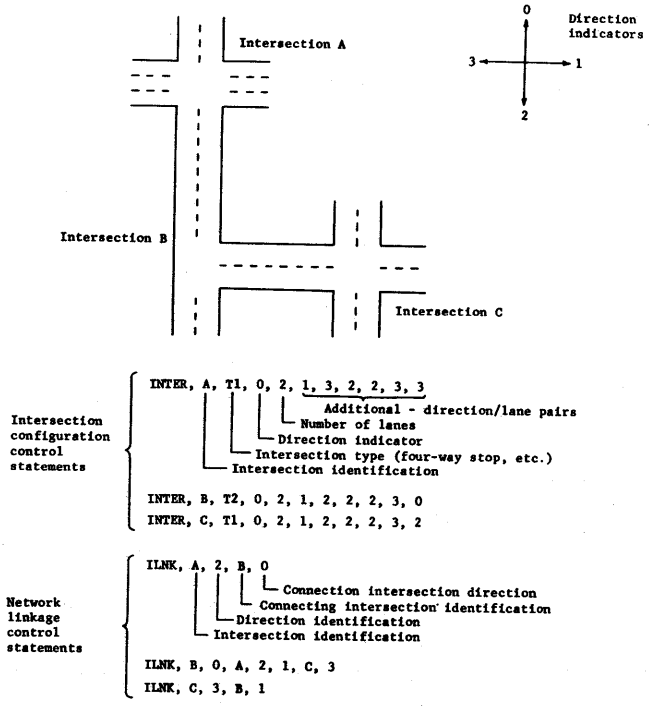


Figure 7(a)—Network configuration control statments

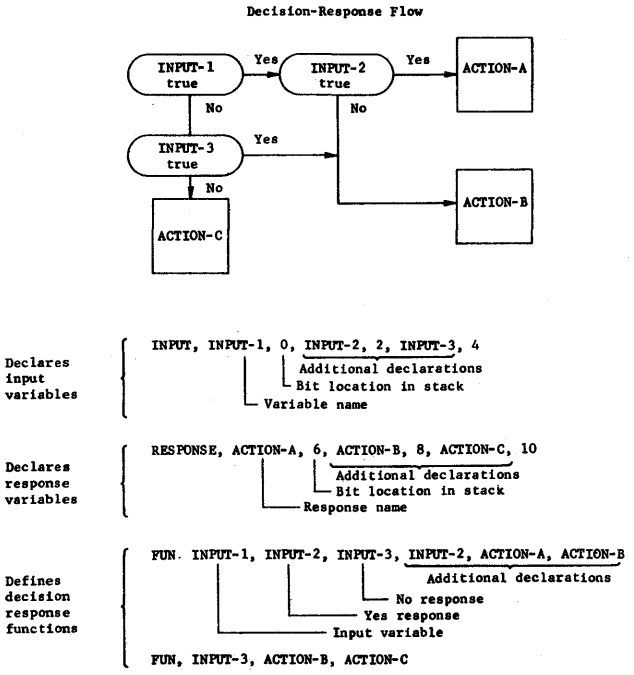


Figure 7(b)—Vehicle characteristic control statements

PATH—vehicle path declaration
SIMULATE—generate random deviates for specified vehicle characteristics

The Network Configuration Statements are used by the model to describe a particular network configuration. Two such examples are depicted in Figure 7a.

The Vehicle Characteristic Statements are used to identify the various variables, parameters, and functions. Figure 7b depicts a typical example of their use. Finally, the Model Subroutine Statements are used by the model during the simulation process in the model subroutines.

SUMMARY

A modeling method for traffic simulation studies has been developed using a step-through or heuristic procedure. A computer program was written for implementing this technique using a hypothetical intersection controlled by four-way stop signs. This technique is currently being expanded to include the general k intersection, n lane traffic network using TRANS, a problem oriented simulation language. The model developed offers the following features:

1. It is readily adaptable to the traffic network problem.
2. The program used for implementing this model requires minimum memory and computer time requirements.
3. The modular design of the model permits ease in expansion to include the many and varied intersection types.
4. The program can be implemented on a small process control computer such as the IBM 1800 to aid control logic for real time traffic control.

REFERENCES

1 R S WALKER B F WOMACK C E LEE
Traffic network simulation and control with driver response criteria
Proceedings of the Houston Conference on Circuits Systems and Computers 13 pp April 1970

2 R S WALKER B F WOMACK
A model for traffic simulation and control
Technical Memorandum No 15 Information Systems Research Laboratory The University of Texas at Austin 102 pp January 1970

3 R S WALKER
TRANS, a simulation language for transportation models
Paper in progress

4 *Uniform vehicle code and model traffic ordinance (revised-1968)*
National Committee on Uniform Traffic Laws and Ordinances 525 School Street SW Washington DC pp 146-147

Realization of a skillful bridge bidding program*

by ANTHONY I. WASSERMAN**

*University of Wisconsin****
Madison, Wisconsin

INTRODUCTION

The problem of bidding at contract bridge is an "intellectual" task which has never before been performed skillfully by a computer program. Only Carley[1] has made the attempt to handle this problem and his program used a very crude treatment; as a result, inferior contracts were produced because bidding requires many fine lines of distinction which his approach could not make.

Slightly more work has been done on other problems related to bridge. Berlekamp[2] has done work on the double-dummy problem for play of the hand, but double-dummy is a perfect information situation and bears little resemblance, beyond the rules for legal play of the cards, to the actual game of bridge. Furthermore, his program was limited to no trump contracts and to contracts in which the declarer was to take almost all of the tricks so that the number of cases (and hence the size of the search tree) remained fairly small.

More recently, Riley and Throop[3] are developing an interactive bridge playing program. Their program does only play of the hand, given the final contract. It is considerably more sophisticated than Carley's program in that it attempts to determine an overall strategy for the play of the hand rather than the isolated, one-trick-at-a-time method used by Carley. Because it ignores the bidding, however, the Riley-Throop program is unable to determine the likely location of cards held by the opposition which might have been inferred from competitive bidding, thus causing the program to lose tricks which would not be lost with proper application of such information. Some of the more advanced play situations appear to be beyond the scope of their present program as well. However, it seems that the program can still be improved, especially by extending it to react to bidding information.

In addition to the fact that bridge bidding is intellectually interesting and has never before been competently accomplished by a computer program, there are numerous other reasons for studying the task. First, bridge bidding requires a certain degree of generality, with a vast number of bidding systems and conventions in current use. In addition, the goal of bridge bidding is not as well defined as the goals of many other tasks. Thus, there can exist numerous techniques to reach any given contract and there is not always a final contract which is clearly superior to all others.

Second, bridge bidding is a task of imperfect information, since each player is aware of the exact location of only thirteen of the fifty-two cards in the deck. Lack of perfect information is characteristic of a large number of problem-solving environments, from corporate decision-making to play-calling in a football huddle.

Third, bridge bidding is a partnership task. Each player must react to both partner and the opposition, distinguishing between friendly and hostile information, a process which requires a considerable amount of judgment in a human. The partnership aspect of bidding resembles the process of negotiation or arbitration or even an auction.*

Fourth, bridge bidding allows the investigation of

* Work reported herein was supported in part by Wisconsin Alumni Research Foundation and by National Science Foundation under grants GP-7069 and GJ-583. This paper is based on a thesis submitted in partial fulfillment for the degree of Doctor of Philosophy at the University of Wisconsin.

** Currently with Shell Development Company, Emeryville, California.

*** Computer Sciences Department

* Contract bridge is a modification of a game called auction bridge.

programming techniques which can be valuable in many application areas. A task language for bridge bidding was designed using the DEFINE capability of Burroughs B5500 ALGOL so that a person familiar with the game is able to read and understand programs written in that language, despite possible inability to write the program. In addition, one decision-making model, chosen for its speed and efficiency, was employed almost exclusively, with the intent of discovering the extent to which it can be used.[4]

Fifth, bridge bidding is a significant intellectual problem. The proliferation of books on the subject of bridge bidding and the popularization of the Goren point-count bidding system[5] have mistakenly led many persons to believe that the task of bridge bidding is rendered trivial by simple adherence to the rules advanced by one of these books and that skill in this field is proportional to the ability to memorize and apply the many different rules. Indeed, skillful bridge bidding possesses many opportunities for the exercise of "judgment" and "imagination", since no bridge bidding "system" specifies action to be taken in all cases and most systems do not adequately treat competitive bidding.

For all these reasons, then, bridge bidding is an excellent subject for research and investigation.

PARTNERSHIP BIDDING

The initial goal of the research was to design a program which would bid very skillfully in the situation in which only the two members of one partnership make bids (non-competitive or "partnership" bidding), with the opponents passing at each turn to bid. A partnership must be skillful in non-competitive situations as a prerequisite toward skill in competitive situations, since competitive bidding is designed, in part, to make it difficult for the opposition to explore their combined holding thoroughly enough to reach an optimal contract. Accordingly, the program was designed so that it could be extended to competitive bidding and eventually to other related tasks.

The program was further initially limited to the Schenken system of bidding,[6,7] chosen primarily because of personal preference and familiarity. This restriction was also made with the eventual extensions in mind. The final initial restriction was to base the scoring system on match-point scoring in duplicate bridge. This last constraint is considerably less significant than the others, since it in no way prevents the program from performing any task; it does, however, have a slight effect upon the program's performance when another method of scoring is used.

The structure of the program is based on the idea of classes of bidding sequences, each class encompassing a large number of possible sequences. A bidding sequence (for partnership bidding) may be defined as the ordered set of bids by two partners from the opening bid to the first pass made by a member of the partnership. There are more than 10^{10} of these distinct bidding sequences possible, making the problem one of classifying sequences and handling different classes in appropriately different ways.

The various classes of sequences can be defined on the basis of different types of opening bids. Each type of opening bid is designed to give a certain characterization of the opening bidder's hand and, generally, to distinguish it in structure and strength. For example, the opening bid of one heart is of the same type as an opening bid of one spade, but of a different type than opening bids of one no trump or two hearts.

The *major classes* may now be defined as the set of all sequences beginning with a certain type of opening bid. Thus, a program structure evolves. There is a procedure OPENBID,* a procedure RESPOND, and eight "sequence procedures" ONECLUBSEQ, ONEBIDSEQ, ONENOSEQ, WEAKTWOSEQ, TWODYMESEQ, TWONOSEQ, PREEMPTSEQ, and THREECLUBSEQ, defining the major classes for the Schenken system. All of these sequence procedures are similar, each representing individual bidding sequences beginning with a given type of opening bid. In addition, there are three special sequence procedures, BLACKWOOD, GERBER, and STAYMAN, corresponding to special bridge bidding conventions, which may be called from any of the sequence procedures. With this basic set of procedures, the problem of non-competitive bidding can be adequately handled.

The decisions to create sequence procedures based on the opening bid and to isolate the initial response into a separate procedure are based on the fact that the two most important bids in a bidding sequence are the first two, particularly the opening bid. If the opening bidder makes an incorrect bid, an incorrect interpretation will be made of his hand. And the error will be magnified if the opening bid is of a different type than the proper opening bid, rather than just a different bid of the same type. Future bids can clarify this initial error, but can rarely make the contents of the hand as clear as it would be with the proper initial bid. The program structure reflects this statement in that an opening bid of the wrong type will cause the program to enter the wrong sequence procedure.

* Capitalized words refer to program designators.

The flow of control for partnership bidding is from OPENBID to RESPOND to the appropriate sequence procedure. Control remains within one of the sequence procedures throughout the remainder of the bidding unless one of the three special convention sequences is entered, in which case control remains there for the rest of the bidding.* Error recovery, if necessary, is handled automatically within the sequence procedure in control.

Partnership bridge bidding may be envisioned as a giant tree structure in which each path in the structure represents a possible bidding sequence and in which each vertex represents a possible bid. Consider the root of the tree to be the point immediately prior to the opening bidder's bid. Then there are thirty-five branches at the first level (one for each legal opening bid), and each succeeding level has fewer branches from each vertex, corresponding to the number of remaining legal bids at that point. The longest path is of length thirty-six (beginning with one club, then each possible bid in succession, then a pass); the shortest is of length two (any legal bid followed by a pass), ignoring the case in which both partners pass.

Since this tree has more than 10^{10} paths, a number of simplifying assumptions are necessary in order to make the task feasible. First, a much smaller tree may be constructed using the previously defined notion of classes. In this reduced tree, the first level has only eight branches, one for each type of opening bid. Then, for each of the eight opening bids, the responses may be divided into several types, creating subclasses, and these subclasses may be divided still further throughout the tree. In addition, the tree may be arbitrarily severed at a length of ten or eleven, since few realistic bidding sequences contain more bids. These assumptions permit a huge reduction in the size of the tree, leaving it with about fifty subclasses which, when subdivided, leave several thousand paths down the tree.

The number of paths may be reduced still further by observing that a great number of bidding sequences, while superficially different, are really quite similar in many bidding systems, with only slight differences in meaning or slight differences in the estimated strength of the partnership. Thus, two paths of the tree can be merged into one by treating sequences of different subclasses identically, creating a new form of structure, a loop-free directed graph.

It was then necessary to connect the vertices so that the network could be traversed. Each vertex was given a marker which a member of the partnership could use to mark his place in the structure at each turn. With this approach, a sequence of markers could uniquely identify a path through the graph. Each player, however, can only use markers that he has found. (It may be that the program's partner is a human who doesn't even know about the internal representation of the bidding and who, in any event, certainly doesn't find or use any markers.) Thus, each bidder begins at the point at which he last found a marker and each partner discovers the same set of markers, since they are following the same path.

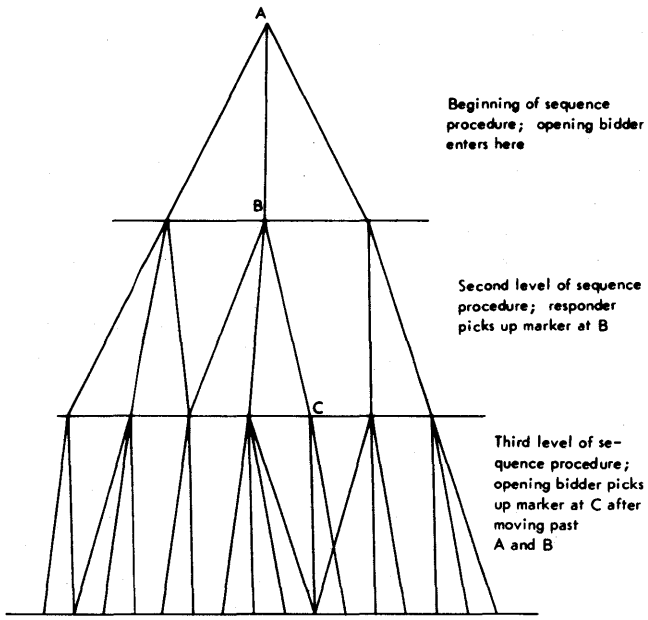


Figure 1—Traversing the network in a hypothetical sequence procedure

Consider a hypothetical structure such as that in Figure 1 and assume that the program is bidding both hands of a partnership. The opening bidder first enters the sequence procedure at A as a result of the opening bid and can then make a bid based on the previous information. The responder then jumps to the same sequence procedure and can move one level down the graph (to point B) based on his partner's last bid to the point at which he must make a bid. He finds the marker at this point, then goes into the bidding process. At his next turn, he will return directly to the marker at B, follow the last two bids (his own and partner's last) down the graph, find the new marker, and make a new bid. In the meantime, the opener, who has not yet picked up a marker, reenters the sequence procedure at A, where he must move down two levels, which he can do based on his own last bid and his partner's last bid. Now he finds the marker

* An exception: STAYMAN can call BLACKWOOD.

at C and makes a bid. This process continues throughout the bidding sequence, with each partner finding the marker and moving down two levels based on the last round of bidding to the point at which a new marker is found and a new bid is made.

Each vertex represents a point at which a bid must be made, a decision-making point. The bridge bidding program attempts to bid in the same manner as a human would, using the same decision-making criteria, since the size of the task is too great to allow programming a completely specified set of rules. The order in which various conditions and alternatives are tested were chosen to correspond closely to the way in which a human bridge expert would select a bid. The program, in effect, simulates human judgment by considering the same conditions and making the same evaluations as would a human.

One of the major mechanisms of the bidding process for each player is a running estimate of partnership strength. At each turn to bid, each player reestimates the minimum and maximum number of points held by the partnership, based on information gained from the bidding up to that point. The early versions of the program used a static evaluation for each player's hand, based on high card and distributional points, computed at the time the hand was first dealt and never changed. This was the figure used in computing the strength of the partnership. Although this scheme worked fairly well, it tended to be inaccurate in extreme cases.

To counteract these inaccuracies, a dynamic evaluation scheme was adopted. Each time it is a player's turn to bid, REVALUEHAND is called, which recomputes his point count, awarding extra points on the basis of length in his partner's bid suit(s) and shortness in his unbid suit(s), and subtracting points from his hand for shortness in partner's suit(s). Further adjustments are made if the partnership agrees on a trump suit. This dynamic scheme results in a more accurate evaluation of the true partnership strength, which, in turn, yields more aggressive bidding when the strength of the two hands coincides and more cautious bidding when the opposite is true.

The overall process for partnership bidding is now completed. Following the opening bid, the responder evaluates his own hand, evaluates the strength of the partnership, and makes a bid. Then, until one of the partners passes, each player finds the correct point in the proper sequence procedure, reevaluates his hand, makes a new estimate of the combined strength of the partnership based on his own new valuation and any additional information gained from partner's last bid, then picks up the marker and makes a bid.

COMPETITIVE BIDDING

There are several significant differences between partnership and competitive bidding. In the partnership case, the bidder always tries to make an offensive bid; the competitive case requires the bidder to consider both offensive and defensive action. In competitive bidding, a partnership must be prepared for the opponents' attempt to interrupt a perfectly normal and logical bidding sequence with an intervening bid. The partnership must also develop methods to reach their best contract after their opponents have opened the bidding. These differences and others make skillful competitive bidding considerably more difficult than skillful non-competitive bidding.

Competitive bidding is further complicated by the existence of a huge number of bidding conventions, very few of which are universally accepted as being standard. For example, if the opening bidder bids one no trump and left-hand-opponent bids two clubs, the two club bid, *a priori*, can be interpreted as being a natural bid or any of several special conventional bids. The partnership using the two club bid, as well as their opponents, must understand the meaning of that bid, since future bidding action is highly dependent on its actual meaning. The competitive bidding portions of the program, then, have to be able to both make and interpret a large number of conventional bids.

Since the program was initially designed with the intent of eventually incorporating competitive bidding, the modifications to perform this task could be made directly, without any major structural changes. A procedure INTERVIEW was created which establishes the bidding conventions to be used by each partnership. Procedures OVERCALL and RESPONDTO-OVERCALL, corresponding to OPENBID and RESPOND respectively, complete the organization of the first round of bidding.

After conventions are selected, the dealer bids first. If and when one of the players makes an opening bid (using OPENBID), OVERCALL is called for the left-hand-opponent. Next, RESPOND is called for opener's partner. The fourth player completes the first round of bidding by using either RESPONDTO-OVERCALL or OVERCALL depending on whether or not his partner made an overcall.

The partnership which opened the bidding then uses the previously defined sequence procedures, which have been modified for competitive bidding. The opposition (the overcalling side) cannot use these procedures as easily, since many of their bids are conventional and cannot be treated in the same manner. In addition, when bidding on a hand becomes very competitive and all four players have made bids, the

original sequence procedures tend to break down, particularly when one of the players has passed once or has doubled an opponent's contract for penalties. For these reasons, a supervisory procedure NEXTBIDDER was created. The responsibility of NEXTBIDDER is to determine whether or not the original sequence procedures are still usable and, if not, to provide the mechanism whereby a player can make a meaningful bid.

The Boolean variable OKTORAISE, computed by NEXTBIDDER for each bidder at each turn, incorporates many of the tests needed for successful competitive bidding. OKTORAISE is true if the bidder (or the partnership in certain cases) has sufficient strength to bid at the current level of the bidding, despite intervening bids by the opposition. If OKTORAISE is false, the bidder will, as a rule, pass or attempt to double an opposing contract.

Additional modifications were also necessary. First among these was the inclusion of doubles and redoubles. The program has to make both penalty and takeout doubles, as well as interpret doubles by other players correctly. Procedures TAKEOUTDOUBLE and PENALTYDOUBLE return the value "true" if the hand meets the conditions to make a takeout double or a penalty double. Takeout doubles make bidding more difficult since they indicate a wide range of strength. Redoubles are much more easily treated since the vast majority of them are made by responder following a takeout double of the opening bid.

Another set of modifications was necessary to handle cue bidding, one of the most sophisticated areas of bridge bidding. Many different meanings can be assigned to a cue bid depending on the exact situation: an overwhelmingly strong hand, control of the enemy's bid suit, asking for control of the enemy's suit, an ace-showing bid, or a general purpose forcing bid when no other useful bid is available. Since cue bids force partner to bid, special checks for cue bids had to be included within the decision-making mechanism.

Many of these checks were made by the Boolean array BIDBYWHO, which also checked to see that both partnerships are not bidding the same suit. Since two players on opposing sides will often have the same strongest suit, one of the players will have to suppress mentioning that suit once the other player has bid it, unless he has some suspicion that the opposition bid is psychic or artificial.**

Some change was also made in the means for evaluating partnership strength. First, REVALUEHAND was expanded to credit extra points for shortness in the opponents' bid suit(s) and to subtract points for apparently worthless honor cards in their suit(s). Second, running estimates of the minimum and maximum strength of the opponents were established for each player. These estimates are only lightly considered, since the natural tendency is to believe one's partner rather than the opposition. However, this information is useful when trying to decide to double the opponents or whether to bid a close contract.

There were a number of options available for handling continuing bids by the overcalling side (the side which did not make the opening bid). One constraint was that the overcalling side might really have the stronger hands and would need to have sequence procedures which were as powerful and versatile as those of the opening side. Another factor was that there are a very large number of overcalls which, if improperly grouped, could lead to an unmanageable number of subsequences.*

Several observations were made which allowed great simplification and eventual solution of the problem. First, the easiest way to have the powerful sequence procedures of the opening side available to the overcalling side was to use them directly as much as possible. Second, in order to use them, the overcalls somehow had to be equated to opening bids. Third, extra provisions were necessary to handle the special conventions and to make the overcall sequences resemble opening bid sequences. (Note that, in actual bidding, there is only one sequence; for purposes of explication, however, two distinct subsequences, one for each partnership, may be considered.)

The above observations were implemented in stages. It became clear that whatever the overcall, the overcall sequence would be highly dependent on the opening bid (rather than the overcall), at least for the first couple of bids. Thus, OVERCALL and RESPONDTOOVERCALL were organized in much the same way as RESPOND, setting up major divisions based on the type of opening bid.

Next, overcalls were divided into classes. In doing so, it was seen that regular (simple) overcalls and takeout doubles indicate approximately the strength as an opening bid of one of a suit, that direct no trump overcalls are very similar to strong no trump openings, that weak jump overcalls are similar to preemptive or weak two bids, and that strong jump overcalls and cue bids are roughly equivalent to strong opening two bids in strength. This division showed that many overcalls

** This inference is very difficult even for the expert human bidder and currently beyond the scope of the program.

* Epstein[8] calculates the number of competitive bidding sequences to be of the order of 10^{47}, compared with 10^{10} for partnership bidding.

could be treated very similarly to certain types of opening bids and that, for these cases at least, the same basic sequence procedures could be used for both the opening side and the overcalling side, simply by providing a set of markers for each partnership in each sequence procedure.

This approach was successful for most, but not all, of the competitive cases. In particular, it failed to cover the highly conventional bids, such as unusual no trumps, Michaels cue bids, and Astro. This failure resulted from the very specialized nature and very limited applicability of these conventional bids; in other words, the approach was satisfactory for all but the most specialized conventions. To solve this problem, the overcalling side treated the opening bids as belonging to one of three classes: strong opening bids, weak opening bids, and no trump openings. Then, three new sequence procedures were constructed to handle the competitive situation, particularly the special cases. These new sequence procedures, named OVERONESEQ, OVERPREESEQ, and OVERNTSEQ, call the regular sequence procedures whenever possible, making the remaining decisions themselves.

The overall program organization can now be summarized very simply. Following convention selection and the first round of bidding, control passes to the supervisory procedure NEXTBIDDER. For each player in turn, NEXTBIDDER calls REVALUEHAND, checks to see if the sequence procedures are still valid for his partnership, and then either calls one of the eight sequence procedures for the opening side, calls one of the three new sequence procedures for the overcalling side (which may then call another sequence procedure), or uses its own methods to make a bid. This procedure retains control until the bidding is ended by three consecutive passes.*

EXTENSIONS OF THE BASIC PROGRAM

A number of extensions were then made to the existing program, including the ability to use other bidding systems, the ability to solve bridge bidding puzzles, and the ability to play with up to three human players. These extensions produced a large amount of generality within the domain of bridge bidding.

The program at first bid only according to the Schenken system, which is less popular among bridge players than Standard American bidding. The extended program contained four basic bidding systems: Standard American, Goren (from which Standard American has been derived), Schenken, and Kaplan-Sheinwold. In addition, many variations on these systems were allowed and a large number of defensive conventions were implemented. This choice of systems was made since an overwhelming majority of bridge players in the United States use one of these systems. In addition, basic bids from the different systems may be combined, as long as no contradictions are created.**

The biggest effect of the addition of the new bidding systems was observed in those sections of the program handling the first round of the bidding and in those where the strength of the partnership was evaluated. Most of the difference among the systems could be incorporated into the estimates of partnership strength. Much of the rest of the program was only slightly altered.

Additional care had to be taken for those situations in which the opening bid may be partially artificial, such as a "short club" in Standard American bidding. Of the eight sequence procedures, only ONEBIDSEQ was significantly affected by these changes, as a result of the greater range of strength indicated by an opening bid of one of a suit in bidding systems other than Schenken's.

Each of the systems has some special quirks that must be specially treated, though, and each of these special conventions was separately handled, creating most of the programming work necessary to implement each additional bidding system.

Defensively, most of the common conventions were made available. At least three alternatives were provided for each defensive situation. Several other possible defensive devices, such as unusual no trumps and Michaels cue bids, were also included. The limits were rather arbitrarily drawn.

The second major extension of the program was made to enable it to bid with people. In its first version, it would bid each hand using only information normally available to that bidder. The result of the program's bidding all four hands is that the program always "understood" the meaning of the other bids. When the program bids with humans, the problem of communication becomes significant, illustrating two major considerations in a partnership game: the amount of faith to place in partner's action and the amount of credence to give to the opposition.

In bidding with one to three humans, the program

* An exception: when the left-hand-opponent of the opener passes and the overcall follows the response, RESPONDTOOVERCALL is called for the left-hand-opponent at his next turn rather than a sequence procedure.

** Procedure INTERVIEW prevents such contradictions.

must interpret all bids as accurately as possible, given the bidding systems of both partnerships. It is important to note that a bad bid by a human might not only tend to confuse the program, but might also confuse any human player. The program tends to put considerably more faith in its partner's bids than in bids made by the opposition.

The inclusion of human players is handled by interactive procedures. Each partnership must decide on its conventions and so inform the opponents, corresponding to the convention cards used in duplicate bridge games. The program interviews both partnerships, which allows it to assign the proper meaning to bids made by the humans. Other interactive procedures allow humans to supply a hand for the program to bid, as well as letting the program shuffle and deal hands by means of random number generation. There are facilities for replaying any hand and for varying the number of hands being bid by humans.

The third major extension to the basic program allows the program to solve bridge bidding puzzles: given a hand, a bidding sequence, and the vulnerability, make the "best" bid. This is a somewhat harder task than merely bidding the same hand at each turn with three human players. Whereas the program previously had to interpret the other three players' bidding, in this case, it must also interpret its own previous bids. In numerous cases, the bids, supplied for previous rounds do not conform to what the program would have bid on its own. (Humans have the same difficulty.) Thus, the program must reconstruct the entire bidding sequence by going through the sequence one bid at a time and computing the strength of each hand and partnership.

PROGRAM PERFORMANCE

Skill at bridge bidding was established as a major goal of the program development. It is trivial to write a computer program to make *legal* bids; it is vastly more difficult to achieve a high level of competence in bidding. Hence, particular attention was given to skill at all stages—first in partnership bidding, next in competitive bidding, and finally in a variety of bidding systems. As a result, the program is a highly skillful bidder, capable of using a variety of conventions to reach contracts which are often superior to those reached by expert human players. Justifying this assessment of the program's ability is the aim of this section.

Several methods are used to illustrate the skill and capabilities of the program. First, the program's bidding on some randomly dealt sample hands will be shown and discussed; second, the results of a noncompetitive bidding contest will be given; third, a summary of the program's performance against two experienced human bridge players will indicate the program's skill in competitive bidding; last, the program's attempts on some bridge bidding puzzles will be discussed.

The bridge bidding program was tested on thousands of randomly dealt hands and continually modified on the basis of its performance on these hands. Every bid on every deal was studied to be certain that the program bid quickly and accurately to the best contract on those hands for which the best contract is relatively obvious, observed the widely accepted rules for describing various distributions and strengths, and made generally "reasonable" bids. In competitive bidding, the program will use all of the specified defensive conventions in addition to the partnership bidding system and can adequately handle highly complicated bidding sequences.

For example, the program bid all the hands on the deal shown in Figure 2, producing a great deal of competition. East will bid up to the four level on the

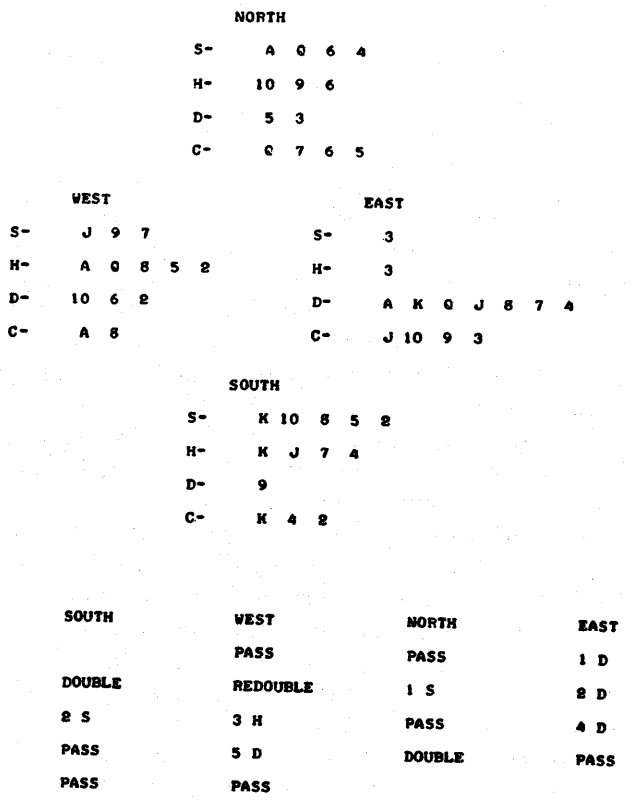


Figure 2—A complicated bidding sequence utilizing the redouble; all hands bid by program

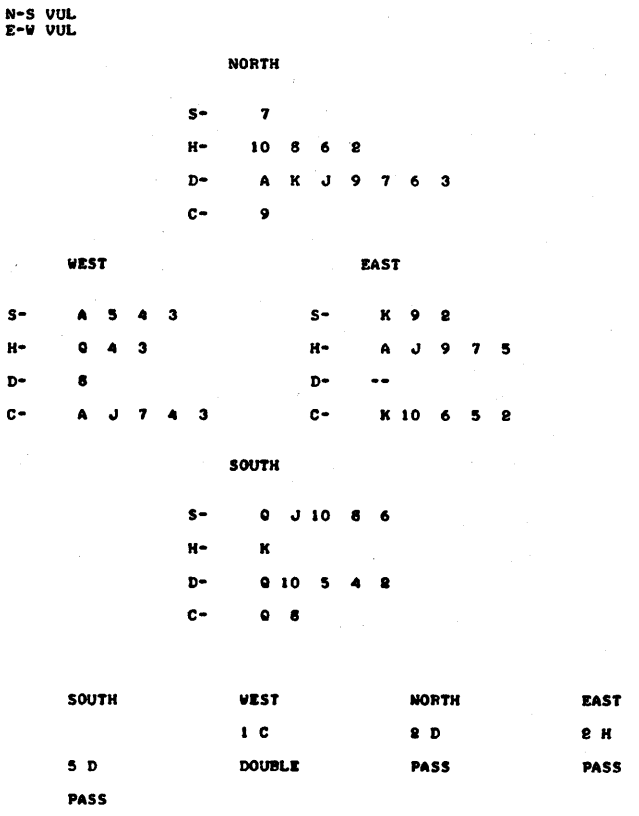


Figure 3—Preemptive bidding as a defensive measure; all hands bid by program

strength of his hand alone; after his partner's redouble, the East-West partnership is simply too strong to allow North-South to play the hand at a low level. North, not counting on East to have such a strong diamond suit and estimating about twenty points in the North-South partnership, doubles the five diamond contract, but East-West should successfully fulfill the contract, losing only a spade and a club.

Very often the program will use weak bids defensively to prevent the stronger partnership from obtaining the contract. On the deal in Figure 3, for example, East-West have the majority of the points, with four hearts being a very logical contract for them. North-South compete, however, with North making a weak jump overcall (one of many defensive conventions regularly used by the program) following West's opening, and South making a preemptive raise to game. North-South do not actually expect to make five diamonds, but rather to obtain a better score by sacrificing than by letting East-West play and probably make a heart game contract. Following South's bid of five diamonds, West has very little choice but to double; the level of bidding has risen too quickly for West to obtain sufficient information about his own partnership to do otherwise. The North-South tactic is highly successful here, since they will only be set one trick, giving East-West two hundred points instead of more than six hundred for successfully making four hearts. (If declarer were lucky enough to correctly guess the distribution of the heart suit, East-West could make either six hearts or six clubs with careful play.)

The program is also able to successfully adjust between being vulnerable and non-vulnerable, and is able to reach a high percentage of possible slam contracts. Although the program occasionally makes bidding errors which produce bad results, it is, on the whole, skillful in using many different conventions, in making many kinds of bidding decisions, and in reaching apparently acceptable contracts. However, no measure of its ability in comparison with human players has yet been given, and such a measure is necessary for accurate evaluation of the skill of the program.

The monthly contest "Challenge the Champs" in *Bridge World*[9] is an excellent means for evaluation of non-competitive bidding. In this contest, two expert bridge partnerships are matched against one another. The editors select ten hands and determine point awards in advance for each possible contract based on the likely result in a good duplicate game, i.e., a duplicate game at a regional or national tournament. By using these scoring awards, it is possible to compare the program directly with the expert players bidding the hands each month. The emphasis is on partnership bidding, with competition either nonexistent, or playing a minor part. "Challenge the Champs" was an excellent measure since it provided an independent evaluation of the program's performance on partnership bidding. Its only drawback is that the scoring awards are only estimates and not substantiated by results in actual play.

"Challenge the Champs" was highly useful in the development of the bridge bidding program. Since the contest is keyed to duplicate bridge results, as is the program, the program could be tested and improved using its ratings, helping to indicate the distinctions that must be made in choosing a bid. The first eighteen months of these hands (January, 1967, through June, 1968) were used in this manner with all major program revisions being tested against the hands to see if there was a net improvement in skill. The following twelve months' hands were then used to actually measure the program's ability. The program had never before bid these hands, nor had any changes been made in the program especially for this test. The program bid these hands according to three different bidding systems: Schenken, Standard American, and Kaplan-Sheinwold.

The average scores of the program in comparison with the human players competing in the match on an overall basis were as follows:

Experts	Score
Winning Partnership	65.7
Losing Partnership	57.2
Bridge Bidding Program	
Schenken System	58.2
Standard American System	51.2
Kaplan-Sheinwold System	50.0

The editors of *Bridge World* consider a score above seventy to be outstanding, with a score above sixty excellent, and anything above fifty to be above average. Thus, the program performed as well or better than the average using all three systems, despite the fact that the program had still not reached the final stage of development. In particular, the defensive bidding sections had not been thoroughly tested, vulnerability was not yet properly taken into account, and there were several coding errors in sections of the program handling Standard American and Kaplan-Sheinwold bidding, since they were added long after the program was operational.

When the program had evolved sufficiently to reduce these weaknesses, the same hands were run again with the program's results considerably improved, as follows:

	Score	Percent Improvement
Schenken System	59.7	2.5
Standard American System	59.0	15.2
Kaplan-Scheinwold System	56.6	13.2

The great improvements in the Standard American and Kaplan-Sheinwold systems can also be attributed to a general improvement in the program, as evidenced by the improved performance of the Schenken System. In this later version of the program, both the Schenken and Standard American systems beat both human partnerships four times, lost to both twice, and finished in the middle the remaining six times. The Kaplan-Sheinwold system won three times, lost three times, and finished in the middle six times. The program has undergone further improvement since these results were obtained, but any general evaluation based on a third attempt on these hands would be biased since some of the improvements in the program were made specifically with respect to bidding problems uncovered in the contest. Even with just the above results, however, it seems fair to say that the program achieves the level of human experts in *partnership* bidding.

One example of the program's performance on "Challenge the Champs" is shown in Figure 4 (Deal 3 of January, 1969), which shows the three systems each reaching the very good contract of six no trump. The Standard American sequence, using the strong three no trump opening, bids directly to the slam. Using the Schenken system, the program began with the Big Club, then jumped in no trump to show additional strength. The Kaplan-Sheinwold sequence is more complex, beginning with the artificial two club bid and continuing with the Gerber four club ace-asking bid, followed by the four heart bid, asking for kings. Missing a king, West stops at six no trump. Although the top scoring award was given to the contract of six clubs, the six no trump contract was second best

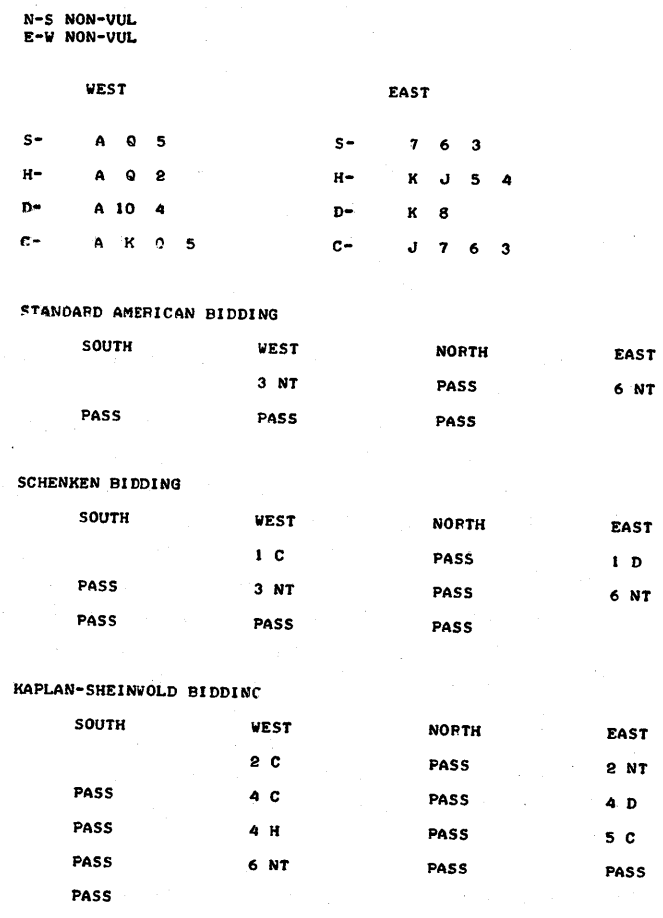


Figure 4—"Challenge the Champs"—Deal 3 of January, 1969; program bid East-West hands according to three different systems

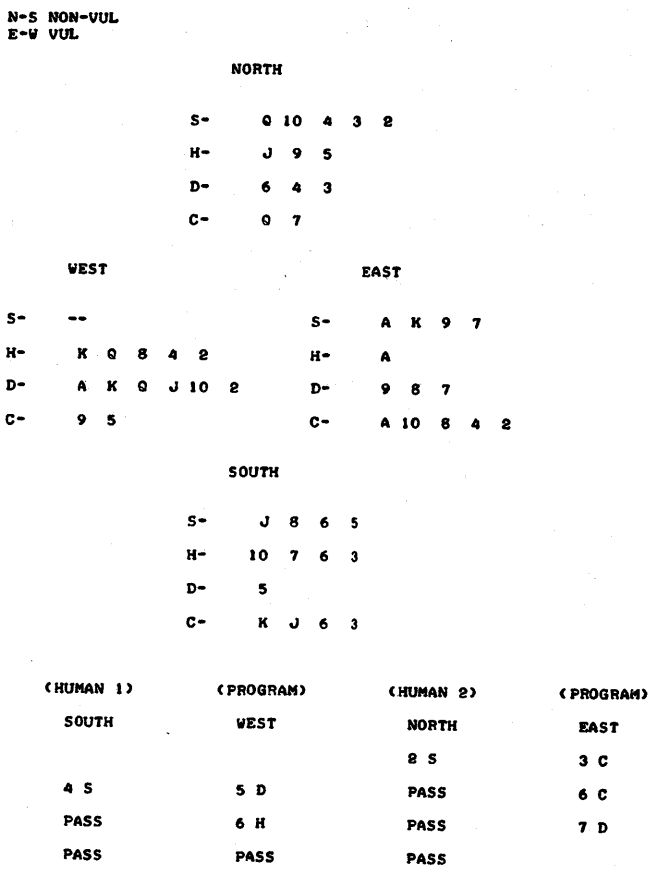


Figure 5—Program (East-West) bidding grand slam during bidding contest

and far superior to the contracts reached by the expert human bidders.

Although the results on "Challenge the Champs" provided substantial evidence of the program's ability, they were almost entirely partnership bidding and the true test of the bridge bidder's skill comes on competitive sequences. Thus, some test of the program's competitive bidding ability had to be designed. Duplicate bridge tournaments were the obvious source for hands to use for testing purposes. The only drawback to using actual results is that the program would have to compete against human opponents (or another program!) in order to obtain an accurate evaluation.*

A bidding contest was set up, matching the bridge bidding program against two human players who have played as duplicate bridge partners many times. One of the two players is a Life Master, with approximately 1000 master points; the other player has nearly a hundred. Hands were obtained from the American Contract Bridge League national tournament held at Cleveland in March, 1969, along with all of the scoring information from the actual tournament.

In the absence of play of the hand, some means for evaluating the bidding alone had to be found. One such means is to consider the average score made by those partnerships bidding a particular contract and to compare it with the average score for other contracts on the same deal. Since this method eliminates play of the hand, it penalizes those partnerships which play the hand skillfully and rewards those partnerships which do not play so well. No other method for evaluating bidding alone, however, seemed to be quite as feasible, and this method was agreeable to all concerned.

The author chose the entire set of thirty hands from the Women's Finals at the national tournament for this contest. The Finals include the top female bridge players in the country, with many partnerships having been eliminated in the two session qualifying round for the Finals.

The contest was conducted in two sessions, with fifteen hands being bid in each session. The program bid according to the Schenken system in the first session and Standard American in the second session. The human players used a Standard American system with some more advanced conventions.

Each hand was scored so that the best score on each hand was twenty-five points and the worst score was zero (matchpointed with a twenty-five top). Scoring is such that the sum of the scores of both partnerships for each hand totals twenty-five, i.e., a seventeen for North-South means an eight for East-West. Twenty-seven of the thirty hands were bid to final contracts which had been reached in the actual tournament. On the remaining three hands, the author and the two human players analyzed the entire deal to determine the probable result for each partnership, assuming intelligent play of the hand by both sides. Only one of these three results favored the program. The overall results were as follows:

	Human Score	Players Percent	Bidding Score	Program Percent
Hands 1–15 (Schenken)	219.75	58.6	155.25	41.4
Hands 16–30 (Standard)	168.75	45.0	206.25	55.0
Total	388.50	51.8	361.50	48.2

* If the program competed against itself, it would simultaneously obtain both good and bad results, i.e., the final contract would be good for North-South if and only if it was bad for East-West. The expected result of such a contest would be a tie, regardless of the skill exhibited.

Thus, the program was victorious in one session, defeated in one session, and narrowly defeated overall. Even in victory, the human players were held to a percentage that would not be sufficient to win most duplicate birdge tournaments. The program's overall loss could have been averted by a swing of just fourteen points, which is available on every deal.

During the contest, the program successfully bid a grand slam (Figure 5) despite intense competition from the human opponents, who used preemptive bidding up to the four level, depriving the program of considerable room to explore for a contract. Note that a sizable amount of investigation by the program came at the six level. This deal, on which the program got twenty-one of the twenty-five points possible, was the best result achieved by the program on its own bidding, and acknowledged by the author and the opponents to be the program's most spectacular bidding performance.

The bridge bidding program made a very creditable showing, very nearly defeating two experienced duplicate bridge players. In discussing the program's performance with the opponents, it was estimated that the program was slightly more skillful than the average duplicate bridge player at competitive bidding.

As a last test of the program's ability, the program was given some bidding problems to solve. Bridge bidding problems are designed, in general, to evaluate skill on intermediate bids; there are some difficulties with such problems, however, which can make them somewhat unsuited for evaluation purposes.

First, as noted, the given previous bidding sequence does not always agree with the bids the program would have made. The program will then try to "correct" its previous bidding error, producing a poor answer to the puzzle.

Second, the specifications for the problems sometimes assume a scoring system other than matchpoints or conventions which are unknown to the program. In these cases, the program does not perfectly understand some of the bids given to it and is unaware of some of the special conventional bids it is expected to make in solving the puzzles.

When tested on problems which had neither of these shortcomings, the program's performance was quite respectable. The problems, however, served to outline several areas in which the program's bidding is weakest. One area in particular is that of cue bidding, which experts use more frequently than does the program, especially when exploring for slams. The program is also still fairly weak in adjusting for vulnerability and in making successful penalty doubles. Combining the evaluations obtained from all the different tests, it also appears that the program has a tendency to be cautious in slam bidding, rarely bidding grand slams, along with a tendency to be slightly aggressive in contracts at lower levels. Last, the program is unable to make psychic bids and has only limited success at counteracting psychic bids made by the opposition.

As a summary result of all these tests, it is fair to conclude that the bridge bidding program is quite skillful, performing at a near-expert level in noncompetitive bidding and somewhat better than the average duplicate bridge player in competitive bidding. In addition, the program is able to provide fairly good solutions to bridge bidding problems, Furthermore, these results were achieved using more than one bidding system skillfully, a talent not possessed by many human players.

CONCLUSION

The bridge bidding program, then, is a contribution to the small quantity of skillful artificial intelligence programs. In addition, the programming methods used in the development of the program are useful tools for other applications. The decision-making procedures, for example, are written in the Bridge Bidding Language which represents virtually all of the important conceptual and strategic notions of the task. Using this language, it is very easy to transfer a bridge expert's bidding technique to program code. The extensible language features which yielded a Bridge Bidding Language from Burroughs B5500 Extended ALGOL could be similarly used to devise other task languages.

The program was also constructed with the goal of having generality within the specific domain of bridge bidding, as can be seen from the program's ability to bid according to various systems and conventions and to handle tasks such as solving bridge bidding problems. It is hoped that this work will help lead to the creation of computer programs which are skillful over a wide range of problems.

REFERENCES

1 G L CARLEY
Program for contract bridge
M S Thesis Massachusetts Institute of Technology 1962

2 E R BERLEKAMP
Program for double dummy bridge problems—a new strategy for mechanical game playing
Journal of the ACM Vol 10 No 4 pp 357-364 1963

3 W RILEY T THROOP
An interactive bridge playing program
Paper presented at Symposium of the National Gaming Council Kansas City Missouri 1969
4 A I WASSERMAN
Achievement of skill and generality in an artificial intelligence program
PhD Thesis University of Wisconsin 1970
5 C GOREN
Contract bridge complete
Doubleday and Company New York 1962
6 H SCHENKEN
Better bidding in fifteen minutes
Simon and Schuster New York 1963
7 H SCHENKEN
Howard Schenken's "Big Club"
Simon and Schuster New York 1969
8 R A EPSTEIN
The theory of gambling and statistical logic
Academic Press New York Chapter 8 pp 270-301 1967
9 *Bridge World*
Vol 38 No 4 January, 1967 through Vol 40 No 9 June 1969

Computer crime

by DENNIE VAN TASSEL

University of California
Santa Cruz, California

One very positive sign in man's existence comes from an unlikely source, that is, his ability to commit criminal acts no matter how difficult the circumstances. He escapes from escape-proof prisons, tampers with tamper-proof devices, and burglarizes burglar-proof establishments. No level of technology has found itself above the ingenuity of a clever, albeit dishonest, mind, not even the computer.

These examples of larceny under difficult circumstances illustrate Dansiger's basic rule: "Whenever something is invented, someone, somewhere, immediately begins trying to figure out a method to beat the invention." Computerized larceny has several advantages over regular old style larceny. Actually, the plain and obvious fact is that computerized larceny is seldom discovered and usually difficult to prosecute even if it is discovered. And since the details are not yet common knowledge perhaps it is worth reconstructing them here, to establish a broad pattern of its development. To start with, address customer files are copied usually with the help of the owner's computer, thus adding insult to injury. Once they are copied the files are sold to a competitor and if the competitor uses the files discretely no one is the wiser, except maybe the sales manager who notices that one company has suddenly become quite aggressive.

Many thefts are simply a by-product of a computer. An example is the computer operator who steals a hundred checks, prints them on a computer on Friday night, cashes them during the weekend, and skips town on Monday. This is not really computerized stealing since the fault lies in the safety of the checks and not the computer. But the crime is usually still blamed on the computer even though a manual check writing machine could have been used just as well.

There are several mythical examples of computer crime. I call them mythical because they actually did happen but the victim of the crime was usually so embarrassed to admit he had been taken so easily, that rather than suffer humiliation, he would prefer to hush up the crime. The first mythical example supposedly took place in a large bank when computers were first being used. An alert programmer noticed that the interest is calculated to the nearest cent and then truncated. That is, if the interest is calculated out to be 2.3333 . . . it is simply left at 2.33—thus contributing nicely to the bank profits. The programmer simply fixed the computer to add some of the truncated portion to his account and in a short while, ended up with a very sizable bank account. All the time the customer accounts stayed in balance. Eventually he was caught by bank auditors who noticed he was withdrawing large sums and not making similar deposits.

Another enterprising young man who received his first set of bank depositors' slips with magnetically imprinted account numbers on the bottom, correctly surmised that the new computer system probably only checked the magnetically imprinted account numbers on the bottom of the checks. So he promptly went to his bank and carefully dispersed his full supply of imprinted slips among the neat stacks at the bank desk. Not too surprising, the slips were used all day by customers making deposits, and even less surprising, the man stopped in the following morning and closed his account, which had mushroomed to over $50,000 and has not been seen since. Needless to say, this scheme no longer works. Crime, like any other business, offers the highest rewards to those who are first to try out a new method.

One of the more interesting aspects of this case is the fact that even though the fault was the improper design of the computer system, the computer was the scapegoat. Using the computer as a scapegoat is a common day phenomenon. Election returns are miscalculated and the computer is blamed when it is really the blame of the programmer. The next time you go into a business and someone blames the computer for an error ask him if he doesn't have people

telling the computer what to do. It is safe to assume that if the computer is screwed up, so is the rest of the business, especially today when most businesses depend so heavily on computers.

Since the computer cannot defend itself, nor prove the accuser at fault, it is safer to blame the computer than another person. This common acceptance of the computer as a "giant uncontrollable brain" has led to at least one very successful embezzlement. Three employees (an account executive, a margin clerk, and a cashier) of the Beaumont, Texas, office of E. F. Hutton & Co., a major New York securities firm, allegedly used the computer as a scapegoat while they were milking customer accounts for more than a half million dollars over a period of several years. They were finally caught in 1968.

This enterprising trio was skimming funds off of customer accounts. Every time one of the clients noticed that his accounts were incorrect the customer was allegedly told that the "dumb computer" had made a mistake, a fable which received instant credibility. The computer all the time was giving the correct results but the excuse covered up the fraud.

Nine years earlier between 1951 and 1959, another brokage firm, Walston & Co., was electronically siphoned of $250,000. By the time the theft was uncovered, the thief had become vice-president. Frank, a manager of a back-office operation programmed Walston's computer to transfer money from a company account to two customer accounts —his and his wife's.

Even though his scheme was extremely simple it nevertheless rates a niche in the history of computer crime. He simply went into the office on a Sunday morning and punched up computer cards to transfer money from a company account to two customers' —his own and his wife's accounts. The computer was further programmed to show the money had gone to purchase stock for the two accounts. Next, he sold the supposedly purchased stock, pocketed the cash and transferred some more money.

The only reason he got caught was because he withdrew a huge sum of money right before the end of the year thus cheating himself out of the interest. The company auditors were suspicious and examined his accounts. Obviously, they found major irregularities, but they were unable to figure out the embezzling system, mainly because Frank hadn't stolen any money from customer's accounts. "What he did was absolutely undetectable without internal auditing," said William D. Fleming, Walston president. "Before it happened no one dreamed such a thing could be done, and if he hadn't explained how he did it, we probably still wouldn't know."

Like most people, Frank took pride in his work and he showed the auditors how he pulled off this smooth embezzlement. The embezzler had been with the company for years and had the most thorough knowledge of the computer system in the firm. Embezzlers who can repay the stolen money seldom go to jail, and since Frank couldn't repay the money he served a year in Sing Sing prison.

At the same time Walston & Company was learning a few things about computers that computer salesmen usually overlook; another brokerage firm, Carlisle & Jacqueline, was being bilked out of $81,000. Richard D., data processing manager of Carlisle & Jacqueline, a brokerage house, instructed the computer to write checks to fictitious persons and send them to his home address. Again a little bad luck stopped this scheme. The post office accidently returned one of the checks to the firm and the clerk who received it blew the whistle. George Muller, a managing partner of Carlisle and Jacqueline, refused to discuss the case. "We'd have to be crazy to give out all the details now so that anyone who wanted to could do it again," says Mr. Muller. The embezzler was convicted, repaid the money, and received a suspended sentence.

In this more recent example it was again only an accident that the computer embezzlement was discovered. In the following case lady fortune smiled with a favor on a programmer, Milo, and frowned on the National City Bank of Minneapolis. Milo had a very bad credit rating and occasionally wrote checks on an empty account but the data processing service center where Milo worked had just been hired to computerize the check-handling system at the bank where he had his account. While writing programs to warn the bank of customers with empty accounts and incoming checks he simply programmed the computer to ignore his personal checks any time his accounts had insufficient funds to cover them. The program allowed each of his bad checks to clear the bank, and didn't debit the employee's account for the overdraft.

The only reason the scheme was discovered was because the computer broke down and the bank was forced to process the checks by hand and without warning in came one of Milo's checks. The check bounced and the scheme was discovered. The check bouncing programmer pleaded guilty in 1966, repaid the money and received a suspended sentence.

Most criminal uses of computers are by individuals but organized crime has not overlooked the possibility of large profits through the use of computerized embezzlement. There are already at least two cases of large scale criminal use. In 1968, a Diners' Club credit card fraud resulted in at least a $1,000,000 loss

to the credit card company. A computer printout of real Diners' Club customers was used by the gang to make up phony credit cards having real names and account numbers on blank Diners' Club cards. According to the police the computer listing was stolen in 1967 by Alfonse Confessore in New York. At the same time 3,000 credit cards disappeared. After the crime was discovered Alfonse Confessore was rubbed out in a gang-land style murder.

The forged credit cards were sold along with other forged identification documents for $85 to $150 per ID package to persons engaged in motor vehicle thefts. Federal agents said that the forged cards were often used to finance a leisurely trip to Atlanta, Georgia, with a stolen car, followed by an air trip home, by way of Miami, Florida.

The most interesting aspect of this case is the sophisticated level of organization. The gang found out that the club's computers were programmed to reject only false names and/or numbers, so the first indication of fraud often didn't come until the real customer received his bill and complained. Thus, ID packages would be completely safe for thirty to sixty days with almost no risk to the user.

Federal agents said that Las Vegas casinos may have been bilked out of hundreds of thousands of dollars after granting credit on the basis of forged Diners' Club credit cards. However, federal agents also said that if any hotel wanted to cooperate in underworld "skimming" of profits, this could be a method of operations since bad credit losses are tax deductible.

In another case a computer was used by a crime organization to embezzle over $1,000,000 in Salinas, California, before the owner was caught in 1968. A service bureau owner, Robert, used his computer to budget embezzlements so smoothly that he was able to take a quarter of a million dollars within a year from a fruit and vegetable firm without the loss being noticed.

Robert was an accountant and he noticed that the fruit company had no complete audit operation. His method included having the computer calculate just how much should be embezzled during a specific period. He did this by using false and real data in different computer runs and by comparing the results on the cost of produce and this way was able to keep all operation costs and profits in balance. The only reason he was caught was because a small-time bank became suspicious of the size of a check made out to a labor organization. Robert was sentenced to from one to ten years for grand theft and forgery.

Banks have traditionally been cautious when protecting their money from embezzlement, so it is not surprising that there have been few examples of computer related crime, but a recent example shows that they also can be victims. In 1970 it was discovered that a total of $900,000 was taken from the National Bank of North America, and a branch of Banker's Trust Company in New York.

The scheme involved five men which included three brothers, a bank vice-president and an assistant branch manager.

The brothers were allegedly able to manipulate bank funds without the banks computers detecting them by making out deposit slips for cash transactions when they were actually depositing checks, according to the district attorney's office.

Since cash transactions are recorded as immediate deposits, checks subsequently drawn were covered by the false cash deposits.

If the deposits were made as checks, the computers would not credit the money to the account immediately. When checks were drawn, the computer would indicate insufficient funds with an uncollected check on deposit, a spokesman for the district attorney's office said.

Two companies were involved in the operation of the scheme, according to the district attorney's office. Bay Auto Sales had an account at the National Bank of North America and Baywood Stables had an account at the Bankers Trust, both in Jamaica, Queens.

The brothers were members of both companies. The scheme was uncovered when a bank messenger failed to deliver a bundle of checks to the clearing house, leaving $440,000 worth of checks uncovered. According to authorities the scheme had been going on for four years.

As the three previous examples show organized crime has already discovered the possibilities available in criminal use of computers but so far no really big embezzlements have been discovered. Yet several very ripe possibilities exist. One of the most obvious is in the area of large payrolls in companies as in the old story about the bar that was losing money. When a check was run, it was noticed that the bartender rang up each sale on one of four registers. Of course, when it was discovered the owner had only three registers, the problem was solved.

Similar scenes have been used with payrolls. Either friends, or fictitious people are paid extra amounts each week. This is especially easy if there is a high turnover of help, or lots of overtime, or piece work pay. Another payroll trick is to deduct extra amounts for tax or other payroll deductions each week and transfer the money to your account. Then at the end of the year calculate everyone's deductions correctly for income tax purposes. The only way someone could

catch this is to save all your weekly payroll stubs and see if the deductions add up correctly at the end of the year. People have a tendency to believe the veracity of a computer printout but careful observation shows that computer programmers and auditors usually sit down each week and calculate their pay to see if it is actually correct. Just a couple of years ago an engineer of an aerospace firm calculated his own interest on his bank account and noticed that it was incorrectly calculated by the bank—in the bank's favor. After several letters the bank decided to humor the guy and check out his account and sure enough the customer was correct. No one had thought to question the computer. When is the last time you calculated your bank interest or paycheck to see if it was correct?

Another area of computer crime which is especially vulnerable is in the area of payroll manipulation. This fact is known by most auditors so payrolls are usually audited rather closely. There was at least one case where a large payroll theft was committed. A group of young men manipulated the computers of the Human Resources Administration in New York City in order to divert over $2.7 million from the anti-poverty program budget. Over a period of nine months false pay checks were made out to 40,000 non-existent youth workers. It is estimated that up to 30 people may have been involved in the scheme.

We have already seen one example of a computer being used to calculate how much to embezzle in the Salinas, California, case. Police can expect to see more of this since organized crime has both the money and the know-how for computer usage. Some of the ways in which computers are used to prevent crime include the analysis of payrolls for excessive overtime pay, or the analysis of inventories for excessive breakage, or selection of any large change in price of items being purchased or sold. All these could be mistakes or legitimate changes but they could also be an indication of embezzlement.

The use of breakage or tolerance allowances is another especially vulnerable area for computerized stealing. Most companies such as warehouses or department stores have a shrinkage allowance to cover items which are lost, broken, or the result of bookkeeping errors. But if a programmer modified the shrinkage allowance at the same time a large scale theft was going on, the theft would probably not be noticed. Once the theft was completed the shrinkage allowance could be reset to its original level. The previous examples of crime have been just criminals modifying the old techniques for the field of crime. But computers have brought forth a new era of crime. This is already evident in the case when a computer was used to calculate how much to embezzle. But there are areas of crime which are unique to the computer field.

A June, 1968 Baltimore headline stated that "Computer Gambling" was taking place in the Social Security Administration. But an investigation showed that all that was happening was 80-column cards were being used in the passing of wages in the numbers racket by a data processing operator. The possibility of computerized gambling is quite real because blackjack and roulette programs that work quite well are available. The only thing holding the computer back is most people prefer the friendly blackjack dealer or the spin of the roulette wheel.

There has always been a rather good market in hot computer gear such as cards, tapes, or disks but because of their size, stolen computers have not until recently entered the picture. In early 1969 a $2,500 Wang Computer disappeared from the Argonne National Laboratories. It was later traced to Iowa State University by the F.B.I. A student working in a training program of Argonne had fallen in love with his Wang computer and took it back to college to do his homework. However, as computers decrease in size we can expect to hear of more stolen computers.

Not only is there a good market in computer gear, but also many a computer hour has been quietly sold by a third shift operator or a dishonest EDP manager. Quite often the buyer is even in direct competition to the establishment where the time is "stolen" since one's competitor would have the most suitable machine and software. In late 1967, the Chicago Board of Education accused five employees of its data processing bureau of setting up their own data processing firm while using the Board's equipment. These five, who have since resigned, allegedly were operating equipment during slow hours and had been doing business with many reputable firms in the city.

The state investigating office found itself in a legal quagmire when becoming involved in this case. Since matters involving computer misuse, and especially any unauthorized use, are so new there are very few precedents to guide lawyers as to whether something like this could be considered criminal.

In addition to answering the veracity of the accusation, the following questions must be answered. Does the use of an unmetered scanner—paid for but not in use—by those authorized to use it for the Board but who use it for non-Board business constitute a crime?

The most common theft in the computer business is in the area of software. Programs can be copied and sold and the copier is almost guaranteed immunity from any legal action since the original never dis-

appears. Competitors hire programmers sometimes on the hope that even if the programmer won't bring any software with them they will at least bring all the software ideas with them to their new jobs. There is no way to estimate software thefts because they are so seldom discovered and quite often are not even of concern to the loser.

One rather large software theft case came to light on the British computer scene. The case involved the biggest commercial installation in progress in Europe, the state-financed airline BOAC. The programming projects involved $100 million programmed on 360/50's and 700 Ferranti terminal displays. The *London Times* at the end of April, 1968, printed a short story which revealed that BOAC was investigating the circumstances in which some employees had expropriated information for consultancy work.

The alleged plagiarism included a combination of IBM's PARS (Programmed Airline Reservation System) and the corporation's own seven million dollar investment in software.

Another software house was implicated as the receivers of the information. It has not been determined whether any legal action will be taken but company disciplinary measures have already been taken against employees.

Another software theft which was discovered took place in Texas. In this case the man was prosecuted criminally for taking computer programs. He worked for a company that developed geophysical programs for oil companies. Each program had a value of about $50,000. He took programs home to work on them and kept copies of them. Within a short span of time he had 50 programs and convinced his roommate to approach a major oil company with the programs. The oil company acted like it was interested and cooperated with the police in accumulating evidence. Both the programmer and his roommate were tried and convicted and both received five year prison sentences.

The Internal Revenue Service has long heralded their computers as devices to prevent income fraud so there was some poetic justice involved in the discovery in June, 1970, that these same computers had been used to embezzle money.

No programming frauds have been discovered but clerical staff has been discovered manipulating input documents.

One would-be computer embezzler was an adjustment clerk who came upon information that some tax credits were not being claimed, possibly because they had been misfiled.

Through data she prepared for the computer, she transferred the credits from one taxpayer's account to another. Each time the credit was recorded, she transferred it to another account. When she felt sure she had covered her trial enough, she credited the tax credit to a relative and refund checks for $1,500 were duly issued.

The embezzlement was uncovered when the IRS Inspection Service, pursuing its regular audit program, came across a complaint from a taxpayer who claimed he had never gotten credit for $1,500 he had paid.

Another misbehaving computer clerk was caught through a banker's alertness. This clerk had manipulated records and established a false tax credit from a true taxpayer for a relative. When the relative took the refund check to the local bank, the banker became suspicious about the size of the refund and alerted IRS.

Inspectors retraced the path of the check back to its source and found the document effecting the transfer to the relative.

Recently a news item reported that a spy had turned over to Communist East Germany business information on over 3000 West German companies. A former data processing department employee made duplicates of tapes stored at his company's leased-time facility and passed these behind the Iron Curtain.

And last but not least, there is the young man who simply changed the program to accept the last card of the file as the final total. This was accepted by the company because no one had time to check out the computer totals. His only mistake was he went skiing one weekend and broke his leg.

CONCLUSION

If better protection measures for computer information are not developed soon, the examples in this paper will seem small in comparison to the new crimes that will take place. I have purposely skipped all examples of sabotage, accidents (man-made or natural), errors and information thefts that could occur with computers. The future holds a real gold mine for a criminal who specializes in manipulating or stealing computer information. One good computer raid could have an immense payoff. If there is any truth in the wise old saying that we should be able to learn from our mistakes, hopefully this short history of computer related crime will alert us and help us to prevent crime in the future.

REFERENCES

1 W AARON
Embezzlement-detection and control
Speech before the National Retail Merchants Association EDP Conference 1968

2 A M ADELSON
Computer bandits
True February 1969 pp 50 74-77
3 A M ADELSON
Whir, blink-jackpot!
Wall Street Journal April 5 1968 pp 1 15
4 B ALLEN
Danger ahead, safeguard your computer
Harvard Business Review November-December 1968 pp 98-101
5 *Calculated computer errors manipulate three bank's security; $1 million lost*
Computerworld March 25 1970 p 1
6 *Computer takes rap in securities swindle*
Datamation August 1968 p 111
7 J DANSIGER
Embezzling primer
Computers and Automation November 1967 pp 41-43
8 *Diners club fraud involved printout*
Computerworld September 18 1968 p 1
9 *Employee accused of illegal computer use*
Datamation December 1967
10 *FBI tracks wandering Wang*
Business Automation April 1969 p 38
11 *Fortifying your business security*
The Office August 1969 pp 39-52
12 *Individual responsiblility*
Data Systems News Volume 10 Number 2 February 1969 p 4
13 M OTTENBERG
Electronic tax fraud investigated at IRS
The Evening Star Washington DC June 24 1970 p A-1
14 *Program plagiarism alleged in UK case*
Datamation June 1968 p 91
15 *$1 million embezzlement arranged by accountant*
Computerworld 1969

TRAN2—A computer graphics program to make sculpture

by ROBERT MALLARY

The University of Massachusetts
Amherst, Massachusetts

INTRODUCTION

Historically the techniques of sculpture have tended to reflect the technological character and level of the society in which the sculpture was made. If primitive man carved bone and the Greeks cast in bronze it should not be surprising that sculptors today are using plastics, lasers, strobes, electronic circuitry and transducers to link art with contemporary technology. Even so, this still does not account for why some of us have been using the computer, assigning it a status above other "art-and-technology" possibilities and viewing it as nothing less than portentous in its implications for art.[1]

The computer is special among the technical resources previously available to the artist because for the first time he has a tool, not only for executing a work of art, but for conceiving one as well. Once a computer has been programmed to generate a first rate work of three-dimensional art with no direct assistance from a sculptor it will be legitimate to speak of cybernetic sculpture in the fullest sense of the word. Until then computer sculpture will qualify as cybernetic only in the sense that the design process is substantially facilitated by "intelligence amplification"—which is to say, by the use of advanced computer graphic interactive systems.[2]

The core problem in computer sculpture is to program the machine to take in, manipulate and give back three-dimensional information which can be used to make sculpture. For example, a Massachusetts sculptor, Alfred Duca, with the help of IBM programmers, has used a computer tape and an N/C machine tool to carve out a large and intricate spherical sculpture in metal.[3] Michael Noll of the Bell Telephone Laboratories has programmed linear stereo-drawings which appear three-dimensional when seen in a stereoviewer. On the other hand, he has apparently made no provision for constructing an actual sculpture from this authentically three-dimensional information.[4] Others have used 2-D plotter drawings to "suggest" sculpture, or have used computer graphic material to embellish the surfaces of a sculpture, or have used computer graphic output to determine the flat shapes to be incorporated into a sculpture. But in none of these latter programs has the computer generated, processed or delivered up the real thing in the way of three-dimensional form information.

TRAN2

In beginning about three years ago to work on TRAN2 our intention was not so much to anticipate the computer sculpture of the future, with its awesome kinetic and form transformational capabilities, as to take a small but real step ahead within our immediate resources. However, TRAN2 does qualify as a basic, or prototype, computer sculpture program in the sense that it provides for a full description of volumetric objects within the machine, processes this information in a meaningful way, and generates a usable output. Moreover, the program is workable in that it has actually been used to make a series of sculptures (see Figure 1 and Figure 2). Now written in Fortran IV for the IBM 1130 computer and plotter, when it has been rewritten for the display it will be upgraded as a more fully interactive program allowing for the almost instant manipulation and transmutation of forms.

Form description

Crucial to the processing of three-dimensional form information—be it architecture, sculpture or indus-

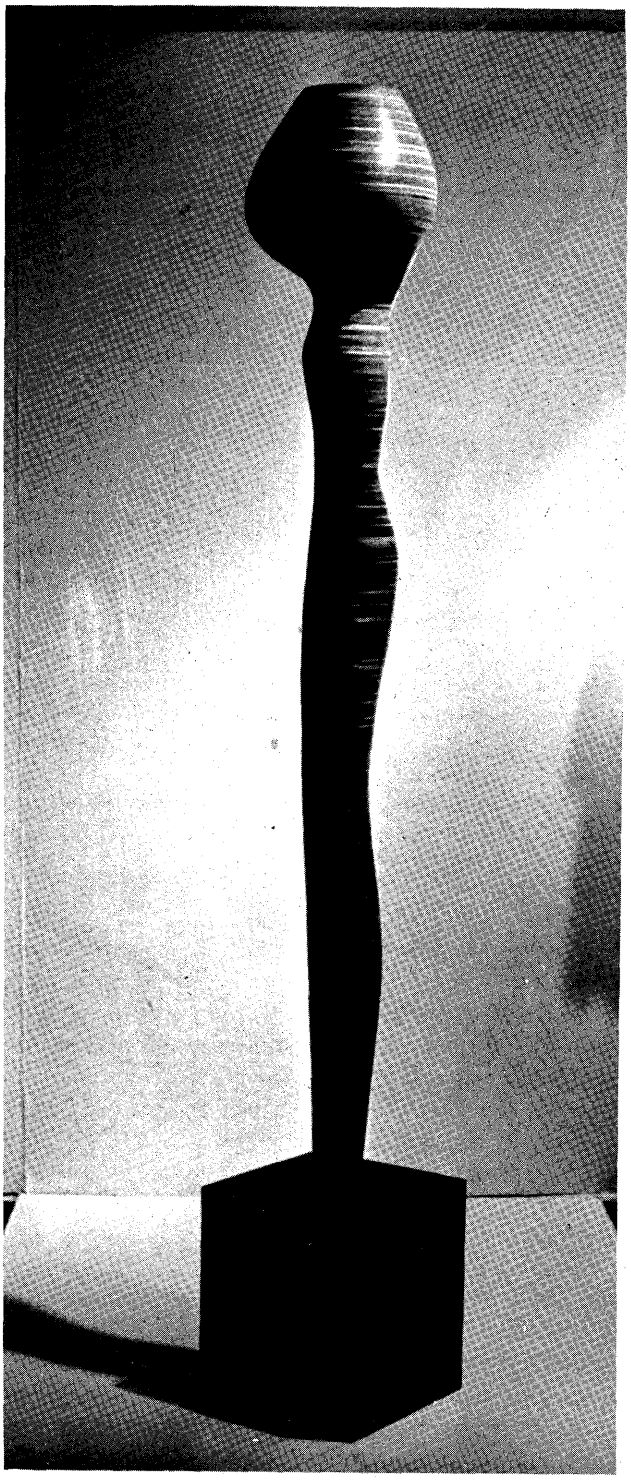


Figure 1—"QUAD III" TRAN2 computer sculpture in laminated veneer. 60" high—1968. The sculpture was made from plotter templates provided by the Amherst College IBM 1130 computer

trial objects—is the 3-space depiction of the object within the machine.[5,6] Perhaps there is no "best" way of doing this, keeping in mind that some methods of form and space description are more suitable than others only in respect to the purpose of the program. For example, MAGI uses a system called "combinatorial geometry"[7] to assemble forms within the computer, but it is difficult, using this system, to revise and further develop the forms as freely as a sculptor would like. Other methods, which refer to surfaces rather than volumes, would seem to be more promising for shaping and processing three-dimensional material—assuming, that is, that provision is made at some point for fully enclosing and defining these surfaces as bounded and complete sculptural volumes.[8] But whatever the system used, if the computer is to be involved with sculpture in an authentic way it must be given either a comprehensive numerical description of the material it is to work with or the means to generate this material for itself.

In this respect TRAN2 does both, using contour sectioning, or "slicing," as the basic method of form

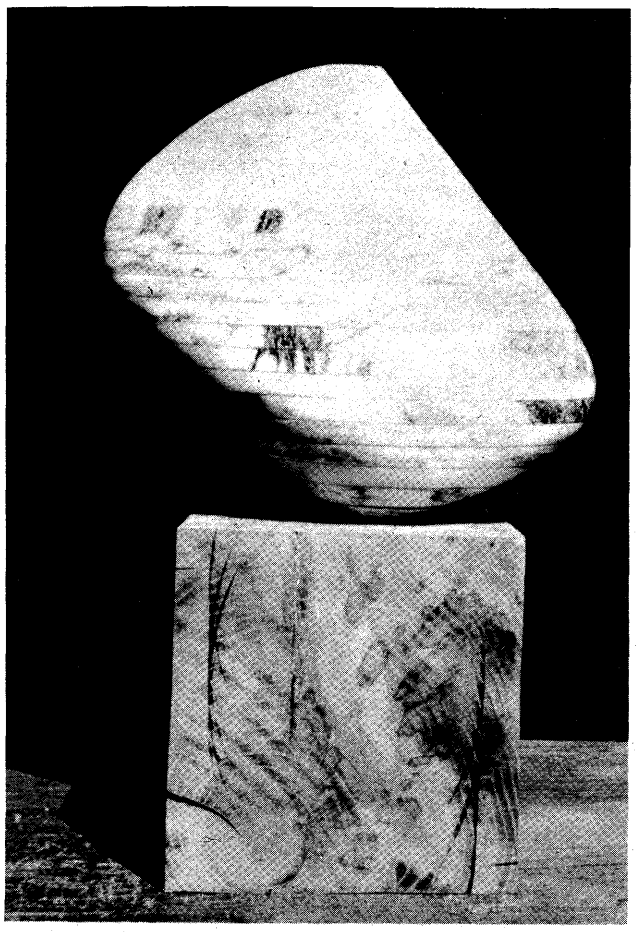


Figure 2—"QUAD IV" A TRAN2 computer sculpture in laminated marble. 11" high—1969

description and form generation. In effect, the form is sliced—much as an apple or a chunk or baloney might be sliced, into a series of thin parallel cross sections of equal thickness. These two-dimensional slices comprise a vertical set, or array, of modular form information units which are then graphed, digitized and encoded onto computer punch cards. Each of the slices has an axis point and a reference ("tick") mark to position it on the vertical axis relative to all the other slices comprising the set. It is by means of this "stacking" of two-dimensional data that the program converts standard computer graphic capabilities to the requirements of three-dimensional form description.

Two modes for form description input

TRAN2 provides for two modes of form description input, though others might be devised. The first, called INITL, requires an antecedent hand-made "prototype" form which must be traced three-dimensionally using a special contour grapher designed for this purpose (see Figure 4). The grapher has a swinging probe which, when held gently against the slowly revolving form, traces off the contour levels one-by-one and transfers them to graph paper. These graphs are then digitized with X/Y coordinates, using the vertical axis as the Z coordinate, and transferred to punch cards. Between 48 and 100 contours are needed, which is insufficient to guarantee a smooth, continuous definition of the form (i. e., without a visible demarcation, or "step," between each contour and the next), but is a practical minimum considering the relatively small capacity of the computer which has been available

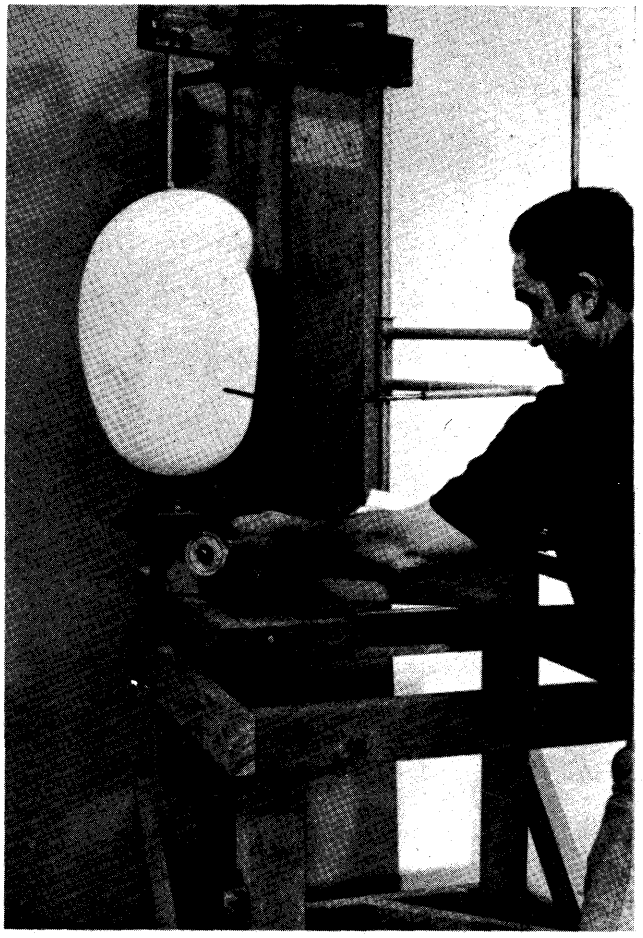


Figure 4—Contour sections are being traced from a styrofoam prototype form mounted in the contour grapher. The contour "slices" are then graphed and punched onto cards to provide the form description input for the INITL input mode

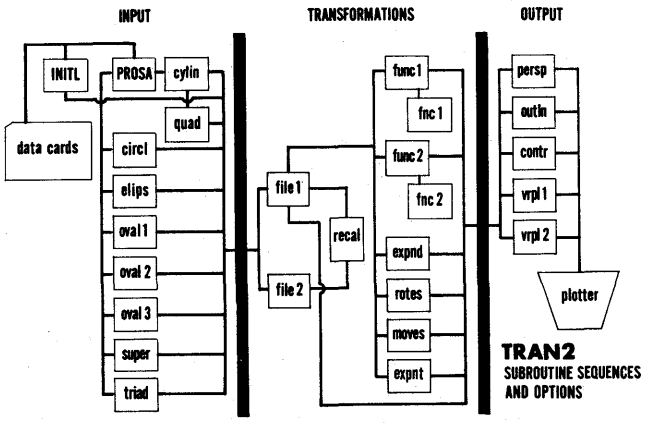


Figure 3—Block diagram showing the basic program structure of TRAN2

to us and the amount of hand work required to translate the computer output into an actual sculpture in-the-round.

The second input mode, called PROFIL, dispenses with the prototype form, but in its place the computer must be given coded profile drawings (see Figure 5). Eight profiles are stored at a time, though the computer needs only one to generate a form with radial symmetry. Two profiles generate a form having two planes of symmetry and three generate a form with bilateral symmetry. A set of four different profiles generates an asymmetrical form, this being the number which is normally specified unless one of the symmetrical schemes is used.

By following the typed instructions given to it at the console the computer "fills in" the form between the profiles (see Figure 6). In calling up subroutines

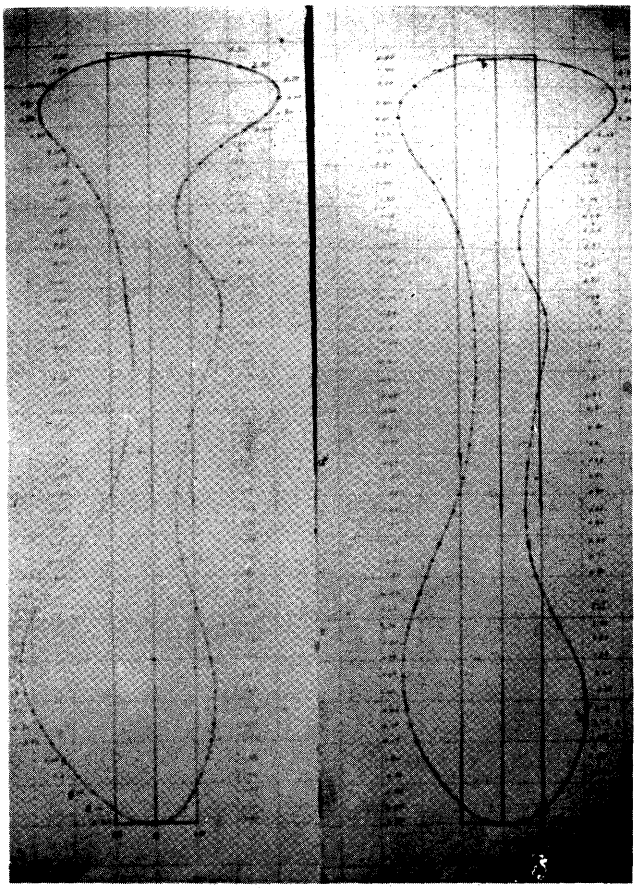


Figure 5—Profile drawings for use with the PROSA input mode. Needed are two facing profiles on the *Y-Z* plane, making four in all. The profiles are generally designed as matching sets, though eight are filed in the computer at a time and are interchangeable

such as ELIPS, OVAL1, OVAL2, and SUPER the sculptor can generate forms for which all the cross sections at all levels along the vertical axis are either perfect circles, ellipses, super-ellipses, or are one of two kinds of ovals. In other words, the coordinate values taken from the profile drawings are used by the computer to shape these geometrical cross sections (see Figure 7).

While this ability to generate an asymmetrical sculpture characterized by a uniform, and perhaps symmetrical, geometry throughout all the cross sections is no guarantee of "beauty," it at least unlocks some interesting possibilities for sculpture which deserve further exploration. For that matter, the extensive use (or should we say *over-use*?) of symmetry in so much current computer art has yet to be applied to three-dimensional structures and configurations, and even in our use of TRAN2 we have hardly begun to

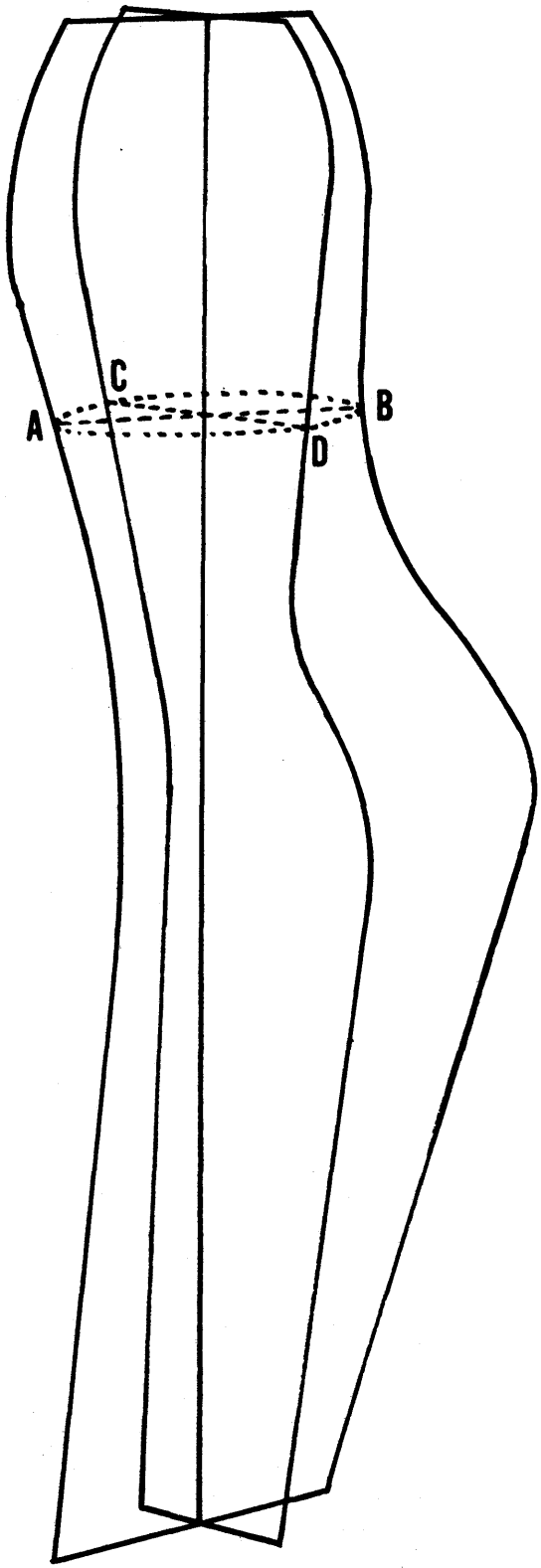


Figure 6—Diagram showing how the computer uses the PROSA input mode to combine 4 profiles and from them derive a contour slice ADBC. 48 of these sections are stacked on the vertical axis to define a complete form. It can be seen that if the same profile were to be used in all positions ($+X$ and $-X$, and $+Y$ and $-Y$) the form would have radial symmetry and all the sections would be circles. Available as options are yet other systems of symmetry

scratch the surface in this area. Evidence of our continuing commitment to asymmetry, as against symmetry, is the fact that QUAD, which shapes an asymmetric contour section based on quadrants taken from four different ellipses, is still the most used subroutine within the PROFIL group of input subroutines.

The transformation subroutines

The computer, once it has either been given or has generated for itself the form description data it requires, in effect is converted into a sculptural modeling and shaping tool. This is accomplished by calling up one or more of the transformation subroutines such as EXPND, EXPNT, ROTES, or MOVES. These generate permutations on the form description data by means of various mathematical functions which stretch and compress the form in a variety of ways.[9] The computer requests instructions regarding the kind of transformation which is wanted and the specific values involved. For example, EXPND is used to stretch or compress a contour section on the X or Y coordinate or on both. If 1.0 is typed for X and the same for Y there is no change in contour section or in the over-all form. But if 0.5 is typed for X and 2.0 for Y the form is halved on X and doubled on Y. Because the computer calculates these values for only one contour at a time, and in sequence, it is possible to specify incremental transformations, either of a positive or negative kind. In this way the sculptor can induce a more drastic transformation at the top of the form than at the bottom (or *vice versa*).

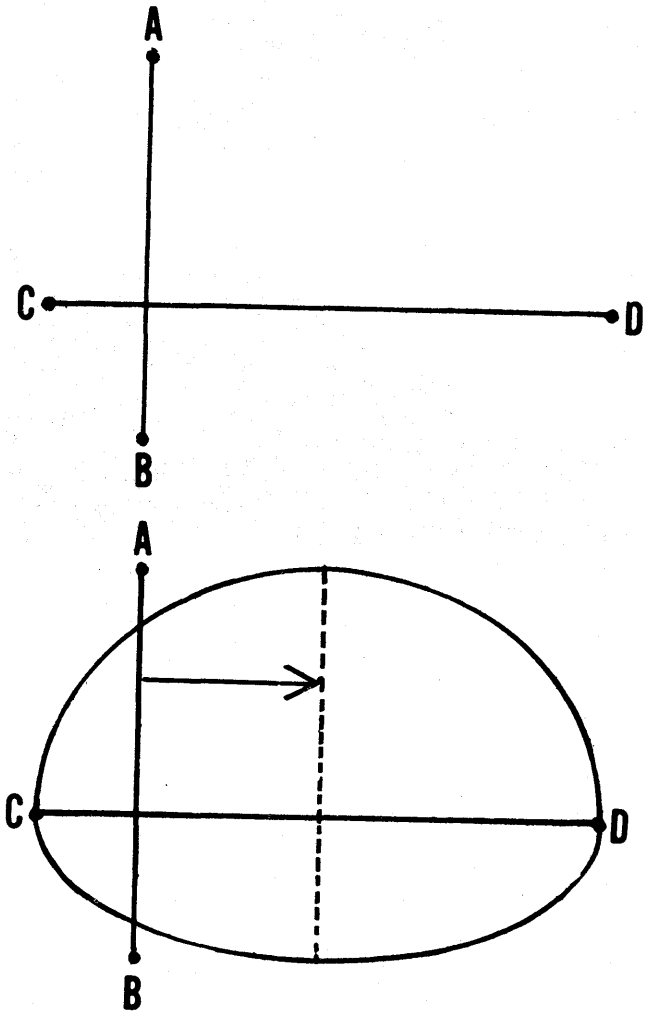


Figure 7—This diagram shows how the computer, using OVAL 2, moves the minor chord *AB* to the center of the major cord *CD*, making a cross section which is a perfect oval (*AB* and *CD* are opposite points on two adjacent profile drawings). OVAL I moves the major chord to the center of the minor chord, ELIPS moves each chord to the center of the other to form a perfect ellipse, and QUAD shapes a cross section comprised of quadrants from 4 different ellipses

In understanding the action of these transformation "templates" it is important to bear in mind their dependence on the underlying structure of the program, based on the stacked layers of contour sections, which limits them to the X/Y horizontal plane. As yet no provision has been made in TRAN2 to introduce transformations on the Z, or vertical, coordinate. EXPND, which is TRAN2's most basic "modeling tool," evenly stretches or compresses the contour section over the X/Y plane of transformation. In other words, the form, in respect to its length and breadth, is *uniformly* stretched or compressed—which means that the kind of transformation is constant even when the values are changed.

EXPNT (for exponential) surmounts this invariance in offering the sculptor a wider range as regards the kind of transformation permutations available to him. Crucial to this much enlarged transformational capability are various logarithmic and exponential functions and values which can be typed in.[10]

For example, using EXPNT the form can be compressed on one side and expanded on the other, or the degree of expansion or compression can be made to vary continuously along the X/Y plane of transformation. Once TRAN2 has been given a graphic console and provided with proper instrumentation it should be possible for the sculptor, using the typewriter and function keys, to specify the class of transformation he is seeking while he spins a knob in order to continuously vary the values. This will enable the sculptor to scrutinize his creation, which is slowly swelling and contracting on the display, while he waits to seize the moment it "gells"—either as the original image in

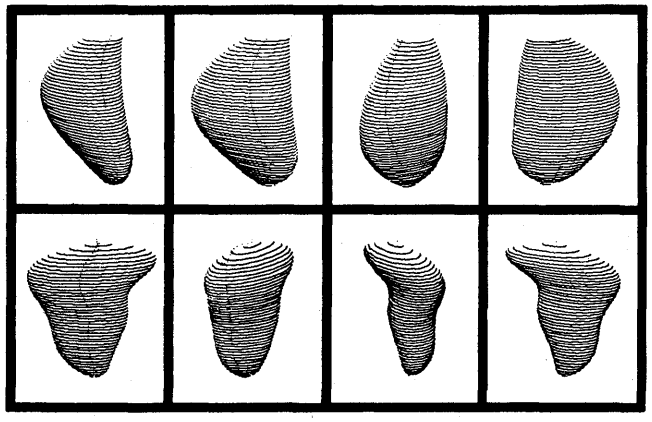


Figure 8—TRAN2 perspective plotter drawing of a sculptural form, which can be thought of as a preliminary sketch or "study." The sculptor decides on the basis of these drawings whether to complete the sculpture in some durable material

his mind's eye or as an unexpected discovery. The sculptor should also have the option of subjecting his form to a series of transformation sequences which are more or less automatic in their operation. As programmed transformation "scenarios" they will be

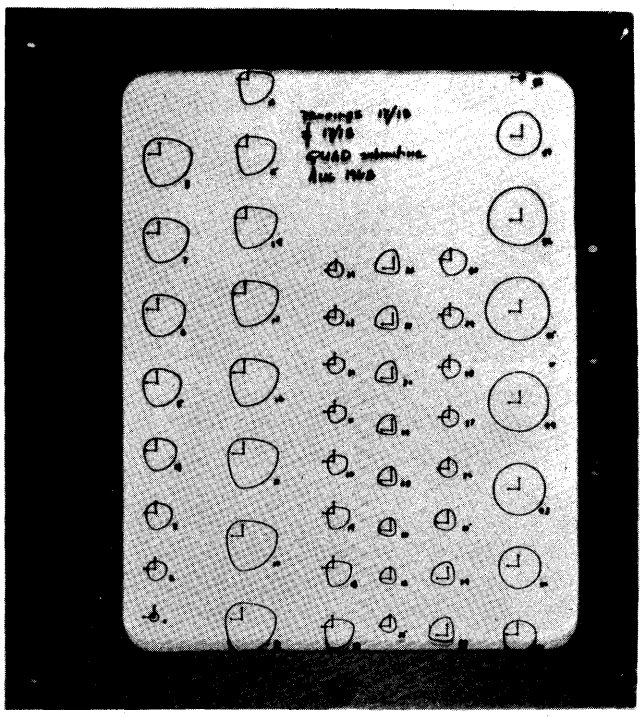


Figure 9—Photo of a complete set of plotter drawn contour sections ready for projection onto the material to be used in making the TRAN2 sculpture

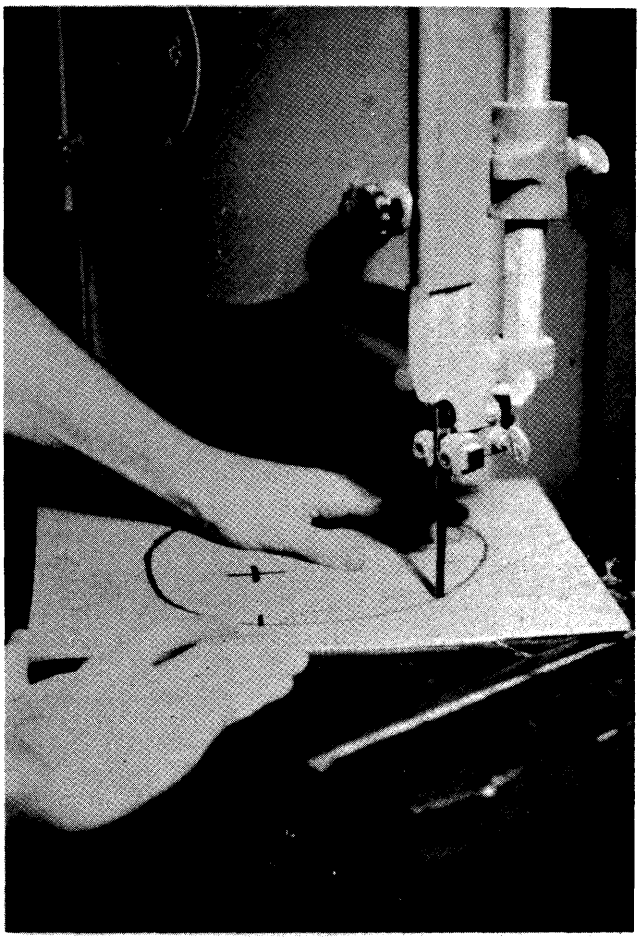


Figure 10—A contour section has been projected and traced onto a ⅜″ thick piece of luaun veneer and is being cut out with a bandsaw. Marked on the veneer are the center hole and a registry mark used to properly orient the slices in the stack as the sculpture is being assembled and laminated

only partially under the control of the sculptor in the sense of his fully anticipating what happens next. They will be useful, not only as an expanded computer-aided design resource, but for their cinematic possibilities. As a third option the sculptor should also be able to "harden," or "fix," any of the transformation permutations at any given strategic point in the procedure, using this new data set to replace the original form description material as the basis for a further round of transformation sequences.

Using ROTES incremental transformations can also be used in order to rotate, or twist, the contour sections sequentially—the final result being to twist the form as a whole. Using this subroutine forms have been twisted 360°, and even 720°, about the axis (with rather bizarre results, it might be added). In general a subtle, less drastic, rotation is preferable—for ex-

ample, the 60° rotation used in designing QUAD III (see Figure 1).

It should be emphasized that the TRAN2 transformations are cumulative—i.e., in being added together in sequence they are in effect combined. For example, the initial input form can first be expanded using EXPND, then have its center point shifted on the X/Y plan using MOVES, then be twisted using ROTES. Sorely needed is a subroutine on the order of MOVES, but more versatile and drastic in its transformations. This subroutine would completely reorient the sculpture relative to its axis, thereby assigning it a new top and bottom and a new set of contour sections. By enabling the transformations to work on the form from any direction, and along any designated plane of transformation, the number of transformation possibilities would be multiplied many times over.

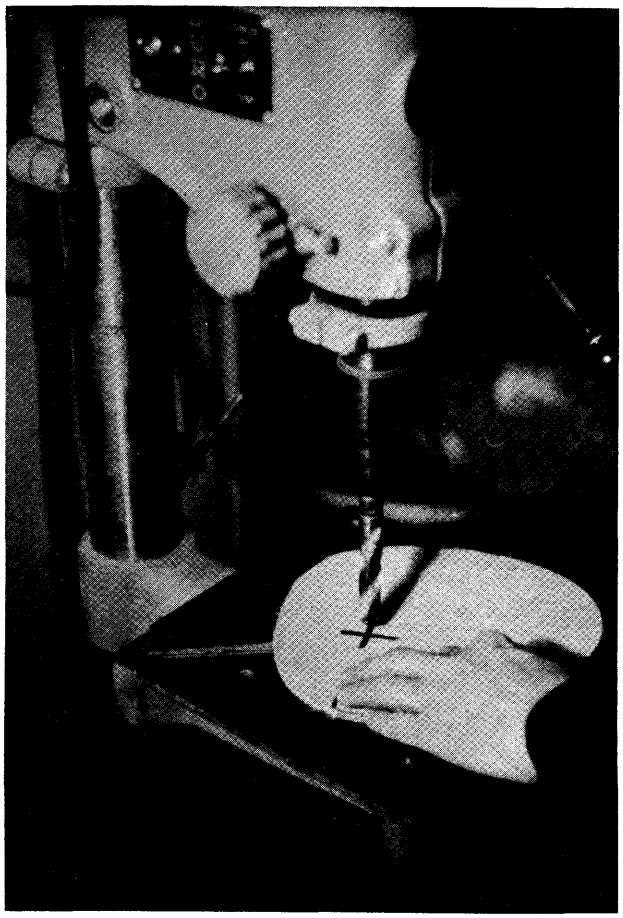


Figure 11—The center hole is drilled

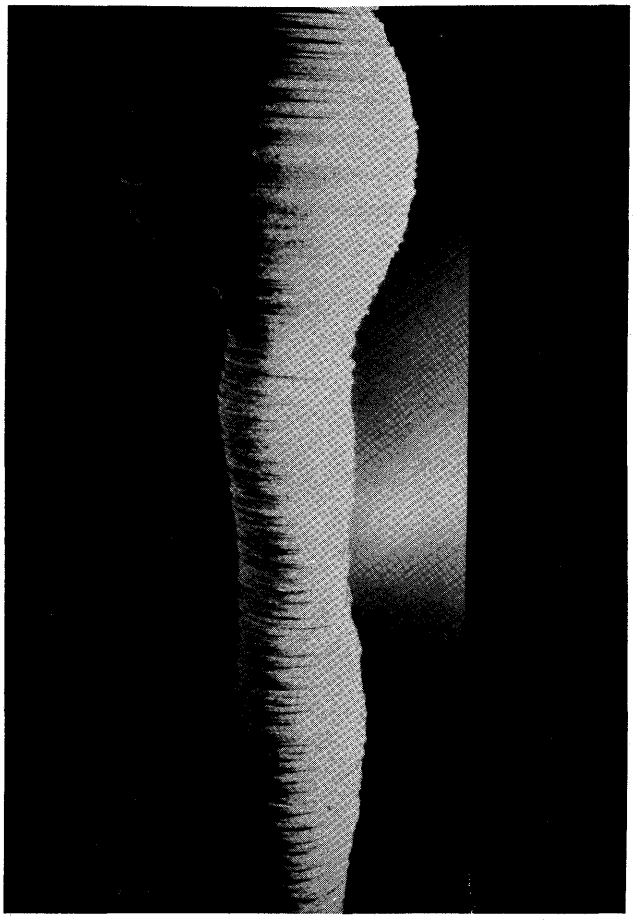


Figure 12—The cut out luaun sections are stacked on a steel rod preparatory to gluing and laminating

The output subroutines

The group of output subroutines determines the kind of drawing the computer is to make. PERSP specifies a perspective drawing (see Figure 8), the sculptor typing in the angle of vision he wants—at, above or below eye level. He also specifies the view or views he wishes, which is apt to be a complete set taken at regular intervals around the form. By instructing the plotter to make a series of drawings—say, at 15° or 30° increments, he in effect revolves the form before his eyes and achieves a rough idea of what it would look like if he were actually to construct it. It is in this sense that TRAN2 is an example of how computer graphic techniques can be exploited by the sculptor to extend and enhance the usefulness of drawing as a way of sorting and clarifying visual ideas preparatory to executing them in a three-dimensional medium.

CONTR calls up a plot of the entire set of contour sections as orthographic projections (see Figure 9).

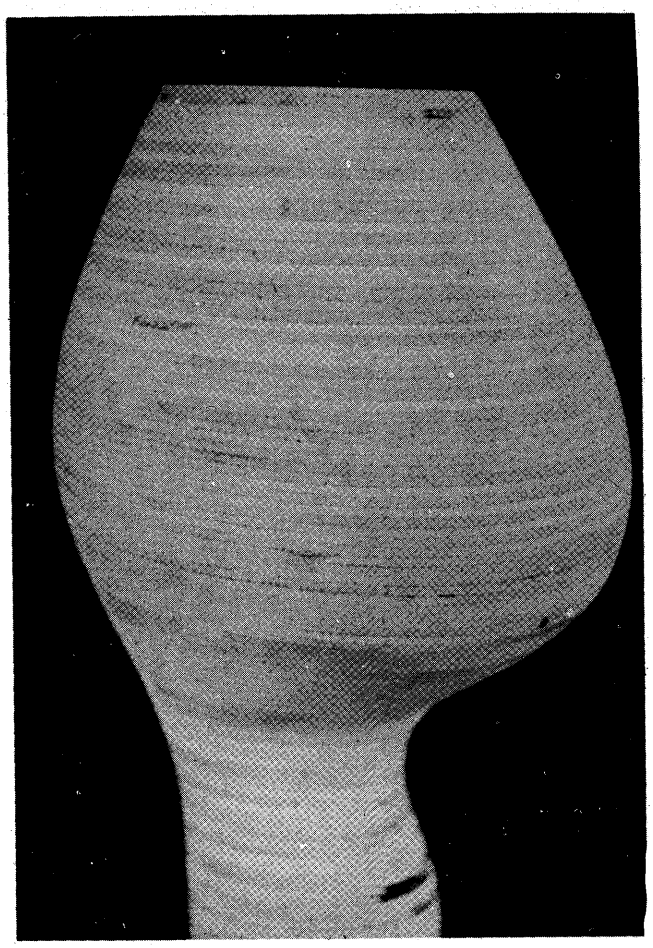


Figure 13—The stacked sections have been laminated together and the irregularities ground down and smoothed

Each of the contours includes both a center point and a reference mark to orient the contours in the proper position one to the other. The entire set of contours is photographed as an 8×10 inch positive transparency, inserted into an overhead projector, projected onto some appropriate material such as wood or plastic, and traced. The set of traced contours is then cut out (see Figure 10), the center holes are drilled (see Figure 11), and the contours are stacked over a metal rod (see Figure 12). Finally the contours are glued, laminated together under pressure, ground down to remove the "steps" and irregularities (see Figure 13), and smoothed and polished. These, of course, are manual operations at the handicraft level, but in principle, inasmuch as the computer has generated all the essential three-dimensional information, the sculpture could also be made using an N/C milling machine.

Capabilities of an improved TRAN2 program

TRAN2 is slow in its operation when measured against the potential of a large c.p.u. and a true interactive program. It is also limited in that it can handle solid, volumetric forms oriented around a central axis; concavities are possible using the INITL input mode, but undercuts are ruled out. Nor is it possible, using PROFIL and its maximum of four profiles, to generate a concavity—though the addition of, say, 12 profiles might improve the situation in this regard (see Figure 14). It will be necessary either to enormously expand the resources of TRAN2, or to develop a library of specialized computer sculpture programs, if planar, linear and open-form sculptures are to be made using the computer.

An ideally interactive program along the lines of TRAN2 would allow the sculptor to draw his profiles directly on the display using a light pen[11] or sketch pad arrangement.[12] He should then be able to evaluate his work by referring to a graphic display which shows the form slowly revolving about its axis. Next he should be able to set in motion a series of transformation scenarios of the type already described, then switch back to the sketch mode—perhaps to refine the details of the form with the light pen or stylus. If he should

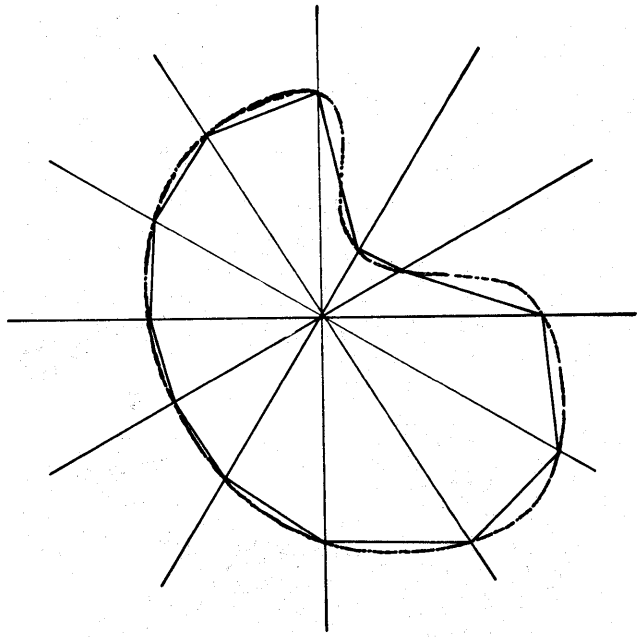


Figure 14—A diagram showing how 12 profiles, instead of the 4 now used in PROSA, will permit the sculptor to introduce concavities into his form and generate more complicated and interesting surfaces. Needed is an elegant algorithm to generate the smooth continuous curve which links the 12 points

decide he has taken a wrong turn at some point he should be able to call upon the computer's memory and return to an earlier stage in order to try something else. In other words, the program should have what amounts to a quick rewind and playback capability based on a complete and permanent log of the entire design process. Apart from its value to the sculptor, a record of this kind might also be valuable as a way of investigating systematically the dynamics of the creative process and determining how one sculptor works as against another in creative problem solving.

It might also be helpful for the computer sculptor, as for the architect as well, if he could rely on multiple displays, each unit providing information regarding a different aspect of the on-going problem. For example, he might study several views of the same sculpture at once, compare two or more current versions, or refer back to an earlier version for comparison. Eventually a practical stereo display, and possibly a holographic display,[14] will optimize the efficiency of computer graphic systems for communicating three-dimensional form information with maximum visual clarity and precision. In fact, we might define the eventual goal of computer graphic systems as providing the sculptor, architect or designer with a virtual real-life experience of the form-in-progress so he may design it better, more rapidly and with more assurance that it will conform to his expectations once it has actually been made.

A look over the horizon

The computer sculptor will make better use of the computer to evaluate his work-in-progress when he no longer has to rely on the rather crude wire cage drawings to which graphic displays are now generally restricted. A minimal step in the right direction would be a drawing consisting of black lines against a white field to replace the reverse image of white on black. More remote, though beginnings are being made,[13,14] would be a simulated light and shadow version of the form on the display giving the sculptor the option of simulating a procedure he is apt to follow in his own studio—namely, to adjust and vary the lighting on the form for the clearest effect. Sequencing the shifting patterns of mobile lighting configurations, either with real lighting equipment or as simulated images on displays, is an obvious computer potential.

Cybernetic sculpture

Ultimately interactive programs will become truly cybernetic in the sense that the computer will move beyond computer-aided intelligence amplification (computer-aided design, in other words) into the more creative aspects of the design process. The computer will be more than a "slave"; it will be more like a collaborator or a virtual surrogate for the sculptor himself. According to his inclinations the sculptor will vary the manner and degree of his involvement with the computer. He will use and interact with it, monitor it or leave it, as it were, to its own resources.

CONCLUSION

The sculpture now made with the help of TRAN2 does not forecast the *look* of the computer sculpture of the future, which will be mainly kinetic, have multimedia features and will probably be based on a cinema type projection system linking the computer with holographic techniques. The relevance of TRAN2 in this connection is that—apart from the claims to be made for it as one of the very first pioneering efforts in the field—it does forecast, in its use of mathematical methods, an approach to form description, as well as form manipulation and metamorphosis, which will be crucial to these kinetic media of the future.

In conclusion I feel it an obligation to remind those who know more about computers than they do about art that I am at outs with some of my more conservative artist colleagues who deny that the computer can make any constructive contribution to art at all. But what is more nettling is that I am also at outs with some of my more avant-garde colleagues who will acknowledge (or even *insist*) that the computer can play a role in art but that TRAN2, which according to them is involved in an anachronistic kind of "object" art, is not a valid way to go about it. I differ with these latter critics in holding that "object" sculpture (which is the kind most people think of, whether it be realistic or abstract, when they think of sculpture at all and which is the only kind we can as yet make with TRAN2) still offers unexplored potentialities for the computer to help uncover.

In any event, a beginning must be made at some point, and for the present it may be a sufficient achievement to have demonstrated that the computer *can* play a role in the making of sculpture—all apart from the question of how well it has done so.

REFERENCES

1 J REICHARDT
Cybernetic serendipity, the computer and the arts
Studio International London and New York 1968

2 R MALLARY
Computer sculpture: six levels of cybernetics
Artforum Vol 7 No 9 pp 29-35 May 1969
3 R CHANDLER
Design for numerical control machining
Machine Design pp 4-24 February 15 1968
4 A M NOLL
The digital computer as a creative medium
IEEE Spectrum Vol 4 pp 87-95 October 1967
5 W FETTER
Computer graphics
Annual meeting of the American Society for Engineering Educators 1966
6 C M THEISS
Computer graphic displays of simulated automobile dynamics
AFIPS Conference Proceedings Spring Joint Computer Conference Vol 34 p 289 1969
7 MAGI
Description of the MAGI technique for accurate modeling and graphic display of three-dimensional objects
Mathematical Applications Group Inc White Plains New York 1967
8 T M P LEE
A class of surfaces for computer display
AFIPS Conference Proceedings Spring Joint Computer Conference Vol 34 p 309 1969
9 C CSURI J SHAFFER
Art, computers and mathematics
AFIPS Conference Proceedings Fall Joint Computer Conference Part 2 Vol 33 p 1293 1968
10 C CSURI
Leonardo: circle to square transformation
The Magazine of the Institute of Contemporary Art Number 5 London pp 27-28 August 1968
11 I E SUTHERLAND
Sketchpad: a man-machine graphical communication system
MIT Lincoln Laboratory Technical Report No 296 January 1963
12 M R DAVIS T O ELLIS
The rand tablet: a man-machine communication device
AFIPS Conference Proceedings Fall Joint Computer Conference Part 1 Vol 26 p 325 1964
13 C WYLIE G ROMNEY D C EVANS A ERDAHL
Half-tone perspective drawings by computer
Technical Report 4-2 Computer Science University of Utah-Salt Lake City Utah February 12 1968
14 H WILHELMSSON
Holography: a new scientific technique of possible use to the arts
Leonardo Pergamon Press Oxford England Vol 1 Number 2 pp 161-169 April 1968

Manufacturing process control at IBM

by J. E. STUEHLER

IBM Corporation
Boulder, Colorado

INTRODUCTION

IBM manufacturing facilities in both the United States and Europe have installed computer systems of essentially identical design to aid in the control of many types of manufacturing processes. The basic structure of the system is depicted in Figure 1. One or two central computer systems (IBM System 360) are attached to several satellite computers (IBM 1130, 1800 and 360 processors) via a high speed Transmission Control Unit (multiplexor).

The satellite computers attach to, and control, various types of manufacturing process and test equipment. The central computer system serves as a data bank, processor and shared input/output device for the satellite computers. It provides for storage and analysis of process data. The central computer minimizes the cost and size of the satellite computers by performing tedious calculations, providing the facilities of a large data base, and reducing input/output requirements. When used, the second central system provides backup, additional capacity, and a better response in a duplexed mode of operation.

This system structure was developed so that one basic design could serve several IBM facilities, thus reducing the hardware and software development costs that would be incurred if each facility were required to develop its own manufacturing process control system(s). In addition, the common design was able to draw upon the combined resources of several locations in order to make optimum use of critical skills.[1] Another major advantage of the common control system is in minimizing the cost of transferring products for manufacture from one location to another.

This paper will describe the requirements for, and development of, the common system. Also included are some of the problems encountered and the oversights, now corrected, that occurred during development and implementation.

HISTORY

The earliest major manufacturing process control system to be implemented in IBM was the system known as COMATS (**C**omputer **O**perated **M**anufacturing and **T**est **S**ystem).[2] This system consists of a pair of duplexed 1460 Data Processing Systems which are attached to numerous satellite processors through a high speed multiplexor (Figure 2). The satellite processors are specially developed computers which provide the control required to test most of the disk storage products for which the system was designed. The system was conceived to reduce the costs associated with developing, building and maintaining special test equipment to do a similar job.

After COMATS became operational, it was obvious that many other manufacturing facilities in IBM had control requirements that could be met by a similar approach. However, each IBM plant manufactures, in general, products requiring their own particular test and process control philosophies. COMATS, as implemented, would not meet all needs. Therefore, in order to save each manufacturing location the cost of developing a unique system, a system versatile enough to meet the needs of most plants was developed.

THE MANUFACTURING ENVIRONMENT

The types of manufacturing processes in IBM extend from the fabrication of microelectronic components through the assembly and testing of complex electronic processors; and involve tiny, precision machined parts through large, electromechanical data processing input/output equipment. The processes required to produce these and other IBM products can, however, generally be classified among five categories:

The first is a process which produces a large quantity of a single type of electrical or mechanical component such as a magnetic disk or core, or a tape or disk head.

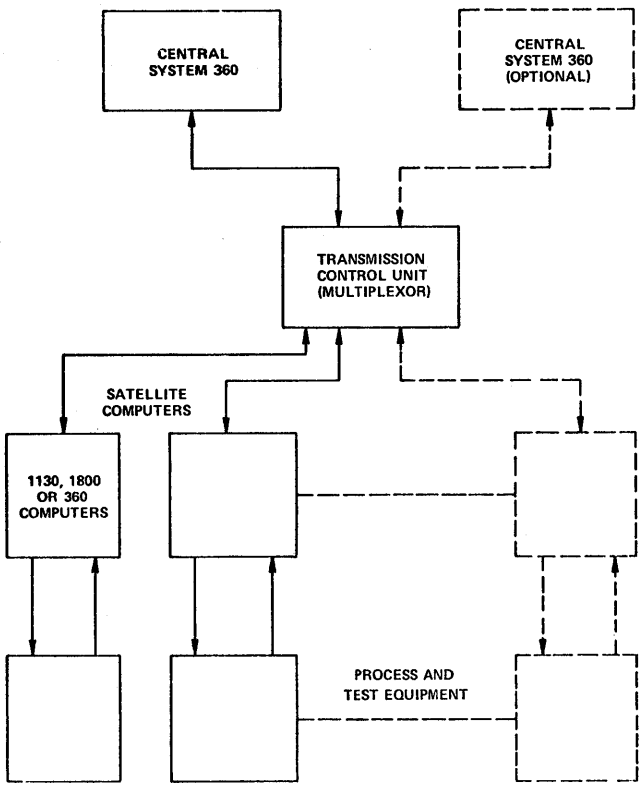


Figure 1—Manufacturing process control system used in IBM

The computer may be used here to control the mechanized operations. In addition, a great deal of process optimization is possible when the computer is used to collect and analyze data to determine the effect of each process variable.

The second type of process produces many "customized" variations of a single product. Examples include integrated circuits and printed circuit boards and cards. The computer can play an important role here by optimally controlling the process and test equipment. However, the computer must also supply information to the manufacturing process to "customize" the product. This requires obtaining and storing large amounts of engineering information as to how each component is to be made and tested.

The third type of manufacturing process produces mechanical and electrical assemblies and subsystems. Product examples include central processing units and input/output equipment such as printers, tape drives, displays, etc. The process consists of assembly and test operations. Most of the assembly operations are difficult to mechanize. However, the computer may be used to give assembly instructions. An important role the control computer can play is in testing.[3] The computer can supply customized diagnostic programs for assemblies under test, provide the control logic required to test electromechanical input/output devices, and retrieve and analyze test results to provide processed output for use in correcting assembly problems and controlling quality.

The fourth type of manufacturing process is general machining of mechanical components. The computer may be used to control the machining and measurement equipment. It is possible for the computer to feed back measurement data to machine tools to control and optimize the process. In addition, the computer is required to convert engineering information into tool instructions for each unique part to be machined.

The fifth type of process actually resides in development rather than manufacturing. An important step in the development "process" is in proving the product to be manufacturable. In order to experiment with process variables, a flexible and easily programmable control system is required. Furthermore, test data analysis is important to determine the effects of varying process parameters.

The manufacturing environment may be further characterized by considering that any one plant may employ more than one of the above types of processes.

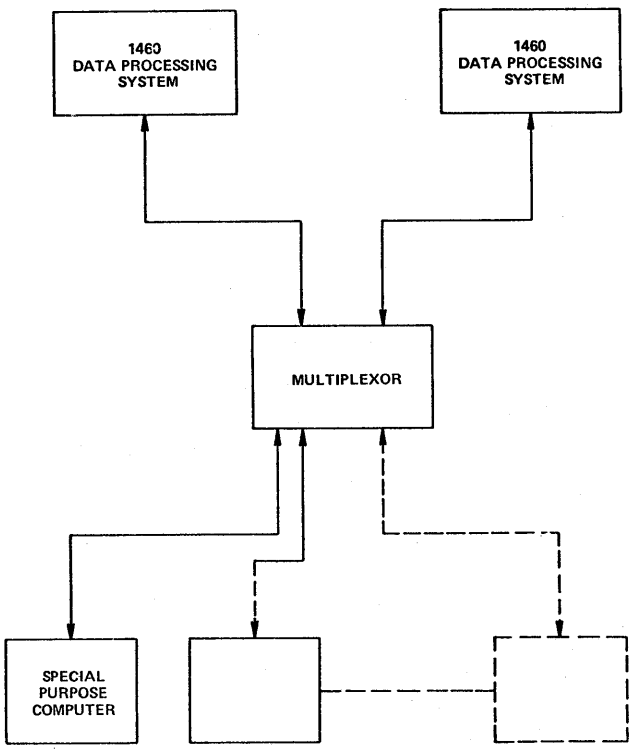


Figure 2—COMATS

Furthermore, a manufacturing process may be spread out in several buildings at distances of up to a mile apart. The processes continuously change to allow the introduction of new products or changes to existing ones. In general, each change must occur rapidly in order to keep pace with development and market requirements. Another trait of the manufacturing environment is that entire manufacturing processes are sometimes transferred totally from one location to another to balance work loads. Often a manufacturing process will be installed in two or more facilities for increased production and/or emergency production.

SELECTION OF A SYSTEM APPROACH

With an understanding of the manufacturing environment, the requirements for a common control system can be identified. The most significant of these includes sensor based input/output capability, extensive information handling and storing capability, modularity with the ability to easily and rapidly install new applications, the ability to mix and transfer all types of applications, and certainly not least important is the requirement for economy.

The satellite computer system concept,[4–9] as illustrated in Figure 3, best implements the above requirements. The satellite computers interface to process and test equipment through sensor based input/output. These satellite computers may be transferred from one location to another and additional satellites may easily be added to expand an existing system. The central processor can provide extensive data analysis and large data banks, while minimizing the need for such capability at the satellite.

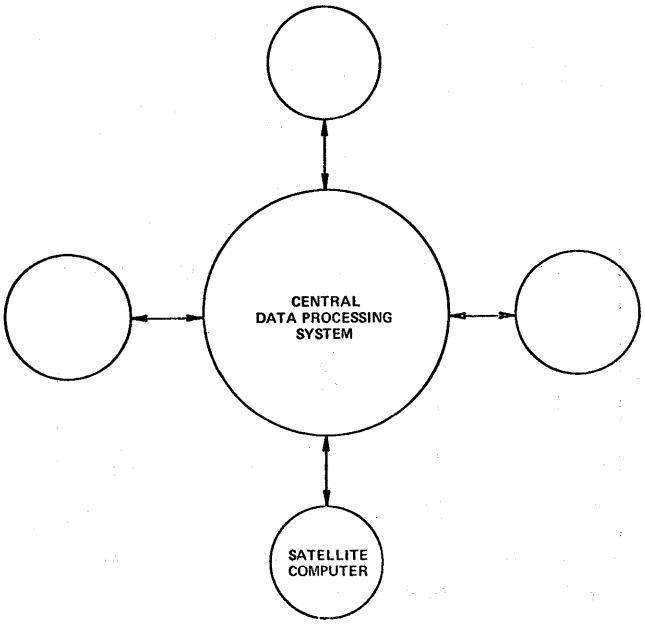


Figure 3—Satellite computer system concept

Two other possible approaches to developing a manufacturing process control system were considered and eliminated for the purposes described here.[10] A single, large central computer system would not provide the power, versatility and modularity required. This is because of the number of different types of control applications in a given IBM plant, and because each such application normally undergoes frequent change which would be difficult to cope with on a single computer system without affecting other applications. The use of a separate control computer for each process would not be adequate because of the expense in providing data banks and information analysis capabilities. In addition, the cost of duplicated input/output equipment and redundant programming would be much greater than with the satellite approach.

DEVELOPMENT AND IMPLEMENTATION

Central computer system

The IBM System/360 was considered the most practical system for use as the central computer. The primary considerations were growth capability plus the existence of many types of input/output equipment and commercially available programs. Only third generation data processing equipment was considered in order to provide state-of-the-art experience and motivation for the skilled programmers who would be needed to design the applications programs and implement an operating system. Other IBM data processing systems were considered which were generally lower in cost than the 360. These systems may have satisfied the needs of some types of processes where large data banks and a great deal of data analysis were not required (example: testing and process development). However, they had limited growth capability compared to the 360; and they did not have the larger analysis and input/output capabilities required in the other process applications. Therefore, at the expense of possibly "over computerizing" some very few locations, the System/360 was selected to maintain commonality.

Satellite computers

It would be desirable to use a single type of satellite computer in order to minimize programming and maintenance costs through familiarization. However, because of the diversity of control requirements at the process level, this was not possible without greatly increasing the average cost of each control application. The amount of control logic required for process or test equipment is inversely proportional to the logic capability of the product being produced or tested. That is, more "intelligence" is required to test components than is required to test input/output devices. Similarly, more intelligence is required to test input/output devices than is required to test systems. Thus, in general, IBM 1800 and 360 Systems are used to control process and test equipment which produces components (process types one and two) while the lower cost and lower powered 1130 System is used on products having higher intelligence (process types three and four). The 1800 Systems are interfaced to process/test equipment through digital and analog input/output channels. The 360 and 1130 Systems are interfaced via special hardware connected to the **O**riginal **E**quipment **M**anufacturers (OEM) channels.

System response

To keep the cost of the satellite computers low, it is necessary to minimize the data processing requirements (amount of core and speed) and the input/output equipment required at the satellite. Thus, the satellite computer will be heavily dependent on the central computer for these services. However, when the satellite computer requires data or programs from the central computer, or is required to send data to the central computer, it cannot wait for a long period of time as the process or test equipment may also have to wait (requiring more production equipment and higher implementation costs). Ideally, the satellite computer should be able to send or receive data from the central system as fast as the satellite could access its own files if it had them. Thus, a system design specification for simple data/program transfer was established at 500 milliseconds 95 percent of the time with the central system handling 3600 interrupts/hour (an average of one interrupt per second). This specification is the length of time the satellite computer must wait from the time it requests a program or data from the central system (or requests the central system to take data) until the program or data (an average of 2,000 bytes) has entered the satellite (or the satellite has sent 2,000 bytes of data). This specification places severe requirements on the communications system between the satellite and central computers as well as on the central computer's operating system.

Communications system

A multiplexor was required to allow communications between the satellite and central computers. The requirements of such a device included:

Modularity in the number of attachable satellites to allow growth (a maximum of several hundred satellite computers in some facilities was considered reasonable while other facilities might never have more than a dozen).

Distance capabilities for communicating up to one mile were required to be able to cover a plant site.

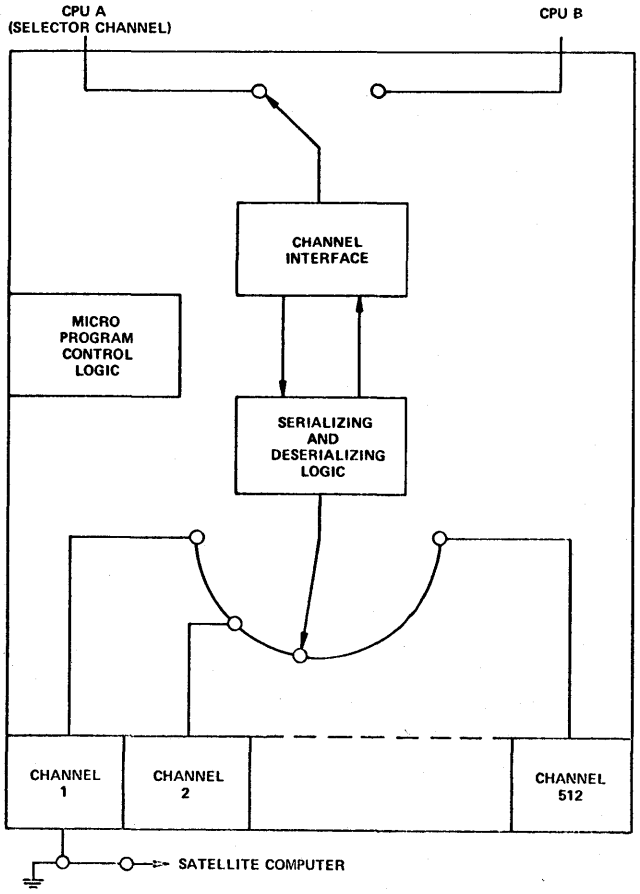


Figure 4—Transmission control unit (TCU)

Channel bandwidth for transmission in the megabit/sec range was needed to minimize satellite waiting time. That is, the channel bandwidth should be of the same order of magnitude as the central CPU channel and file transmission rates.

Serial by bit transmission was required to minimize cabling costs.

High reliability of transmission was required to allow operation in an electronically noisy factory environment.

No commercially available multiplexors were found to be suitable under the above requirements. A **T**ransmission **C**ontrol **U**nit (TCU) was, therefore, developed; this unit is shown schematically in Figure 4. The TCU is basically a micro-programmed, solid state switch which provides polling and allows the communication of either of two central 360 computers with any of up to 512 satellite computers (modular in groups of 64). The TCU also provides serializing and transmission logic which allows transmission of serial by bit data over a single coaxial cable at a rate of 2.5 megabits/second. The number of bits in error is less than one bit for every 10 to the eighth bits transmitted. The TCU communicates with a satellite computer via a transmission adapter.

Central computer operating system

Some of the more important characteristics of an operating system in the central system include the following:

Response time to accept an interrupt from the TCU and begin processing should be of the order of a few milliseconds to provide the response required by the satellite computer which may be waiting.

Input/output support for many different types of devices is required to allow the system to be applied effectively in the different process environments. That is, one environment may require small, fast-access files while in another environment slower access to large quantities of information is required. Differences may also be found in the requirements for graphic terminals, printers, tapes (for history), etc.

Multi-programming is required to allow one or more satellite computers to receive service while file accessing or other input/output operations are pending for another satellite computer. This capability greatly reduces the waiting time due to queues for the satellite computers.

Support software such as compilers, assemblers, analysis routines, etc., is required to minimize programming costs and the need for programmers.

Modularity is required in order to minimize the price the small user must pay in core storage overhead which is required to obtain the sophistication needed by the larger user (examples include number of levels of multiprogramming, requirements for compilers, concurrent operation of peripheral input/output devices, types and amounts of input/output equipment, etc.).

Two approaches were considered to implement the above major requirements. One was to develop a special operating system and the other was to implement a commercially available one. The special operating system would be best from the standpoint of response and amount of core required since it could be customized to perform well in these areas. The disadvantages were that a great deal of development work would be necessary to provide the versatility required to support the varied input/output requirements at each using location. Furthermore, advantage could not be taken of already available compilers and utility programs designed to operate under a commercially available operating system. For these reasons, the decision was made to use a commercially available operating system.[11,12]

The best system available to provide multiprogramming capability was the IBM Operating System/360 (OS/360) which was augmented by a "Secondary Control Program" to provide a "real time" multiprogramming environment for supporting the satellite computers. This combination of OS/360, the Secondary Control Program and an input/output appendage to support the TCU is referred to as the **P**rocess **C**ontrol **O**perating **S**ystem (PCOS). The core map of the central computer is illustrated in Figure 5. TCU communications and TCU-detected errors are handled by the TCU appendage. All TCU interrupts are passed to the Secondary Control Program which invokes either a core or disk-resident Service Module (real time program) to handle the interrupt. The Service Module may (if required) initiate a background program to perform analysis on information the satellite computer has sent. The Service Modules always have priority in utilizing the central system resources. This allows a fast response to a satellite request for service.

Normally, the time required to enter a core-resident Service Module after the TCU posts an interrupt to the central computer is less than 25 milliseconds (if the service module is not active).

A drawback of using OS/360 as compared to a special purpose operating system is in the amount of core required. A Model 40 is the smallest system in the 360

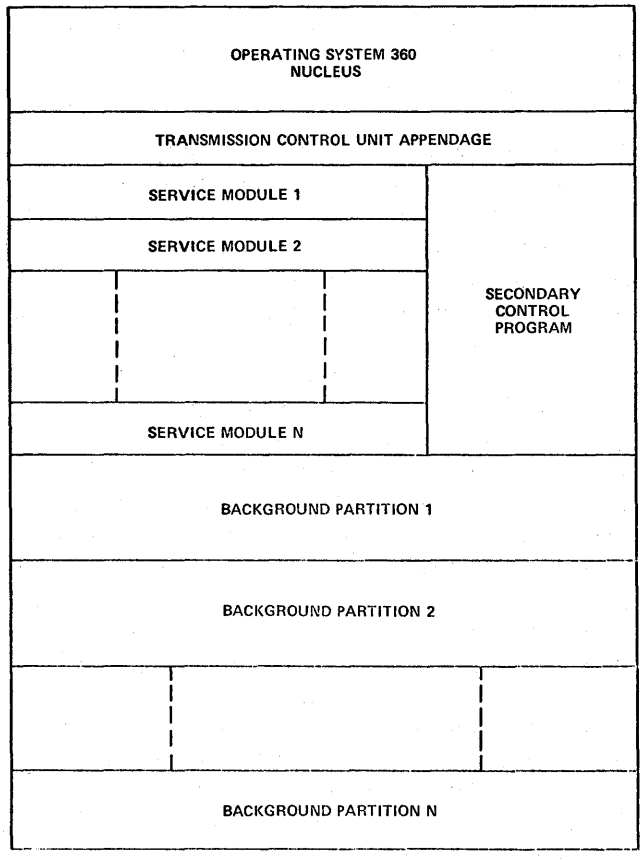


Figure 5—Process control operating system (PCOS)

line which can effectively run OS/360. This again is an expense to some locations which might have started off with a 360 Model 30 as the central computer if a "special" operating system had been developed.

CONSIDERATIONS IN RETROSPECT

Experience has revealed two problems that have now been solved but were not originally anticipated. One was a hardware design problem and the other a software problem. After the TCU specifications had been determined, any location planning on the first usage of a type of satellite computer (i.e., 1130, 1800, or 360) had the responsibility to develop the unique transmission logic adapter between that type of satellite computer and the coaxial cable which connected to the TCU. It was later learned that each designing location had developed an interface completely different from the others. As a consequence, common software in the central system could not be used to communicate with every type of satellite computer. This was because each type of transmission logic adapter presented different status indicators to the TCU or responded differently to TCU commands. Furthermore, since each adapter was designed differently, an engineering change placed in the TCU might affect some adapters adversely while not affecting others. This put a handicap on the TCU designers.

This problem was solved by developing a common transmission logic interface which could be used by every type of satellite computer. The common adapter was then uniquely interfaced to a particular type of satellite computer as illustrated in Figure 6. Now all satellite computers look the same to the TCU and central system software.

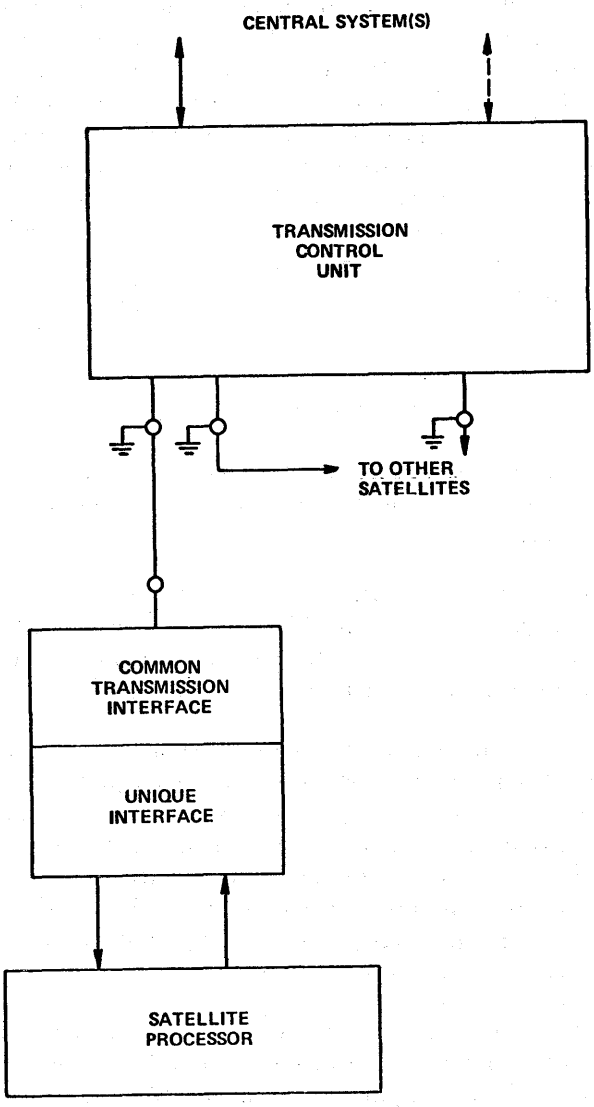


Figure 6—Transmission logic scheme

The software problem had to do with the definition (or lack of definition) of the Service Modules (real time programs) which reside under the Secondary Control Program. Most system users were developing Application Service Modules to be unique to a given application. As an example, a particular tester would require one or more unique Service Modules to completely support it; and these "Application" Service Modules could support no other application. Other using locations, however, began development of "System Service Modules." A single System Service Module could provide service to two or more applications. An example might be a System Service Module to retrieve information and store it on a file for any application, whereas each Application Service Module contained its own retrieval logic. The benefit of System Service Modules is that these routines, which can be common, would have to be developed only once. However, there is an initially high development cost for each such Service Module since they must offer a great deal of versatility.

The problem presented by having these two philosophies was that applications developed to be run on a system using System Service Modules could not easily be transferred to another location without also transferring the System Service Modules or rewriting the application programs. This problem was solved by defining and developing a common set of System Service Modules which can be considered an integral part of the operating system. Being able to change this philosophy of operating system function attests to the need for a great deal of operating system versatility. It is not unreasonable to expect that other conceptual changes will be made in the future based on knowledge not yet gained.

DEVELOPMENT AND IMPLEMENTATION

Architectural, as well as hardware and software specifications for the system were developed by representatives from each user location in 1967. Development responsibility for portions of the hardware and software were assumed by several of IBM's manufacturing locations. In 1968, the components of the system were brought together at a single IBM manufacturing facility in Boulder, Colorado, for successful system testing. The system is presently installed and operating in nine domestic and two European IBM manufacturing plants with plans for implementation at several other plants. The need for a versatile system approach can be attested to by looking at how the common process control system is applied at several IBM locations. At two locations, 30-50 satellite 1130 computers are being used for testing electromechanical input/output devices. Both of these locations utilize the central system as the satellite's input/output device (storing programs, reporting, etc.). Furthermore, the central system performs test data and defect analysis and reporting for the Quality Engineering organization. Two other locations use the system heavily for controlling processes producing magnetic components. Here some satellite computers (e.g., 1800's) are used to control process variables while other satellite computers (e.g., 1130's) control test equipment. The central computer is used to store and correlate test results with process variables, thus allowing process optimization. As soon as enough history can be built up, process models can be designed and installed in the central system to better control the processes. Two other locations use the system primarily for supplying test programs to central processing units undergoing test. Here the central system stores and supplies large diagnostic programs to satellite computers. The central system collects test and diagnostic data for engineering analysis. One of these two locations uses its system to give assembly instructions via display units to assembly personnel working on complex electronic subassemblies. One IBM location uses its system to test complex integrated circuit memory modules. Here the central system must supply test data to satellite computers which control test equipment. Again, the central system receives, analyzes reports and stores relevant test data received from the satellite computer. Other IBM locations have combinations of the above applications installed on their systems. The number of satellite computers range from half a dozen at one location to over 50 at another. Central system configurations include a single System 360 model 40, a pair of model 50's, and a single model 65. All using locations appear satisfied with the flexibility the system affords.

SYSTEM PERFORMANCE

It is impossible to generalize any system performance criterion because of the differences in system configuration implemented by each using location (i.e., types of input/output, size of processor used, features of operating system utilized, etc.). However, a "representative" location has installed 22 satellite 1130 computers on a System 360 model 40 central system with 256 K bytes of core. They utilize the multiprogramming with a fixed number of tasks (MFT) version of the 360 Operating System and use a 2314 Disk Storage Unit as their bulk file. The observed response of their system closely follows the results of the simulation of their system which is depicted in Figure 7. This shows that 1890 messages per hour can be handled with a response of

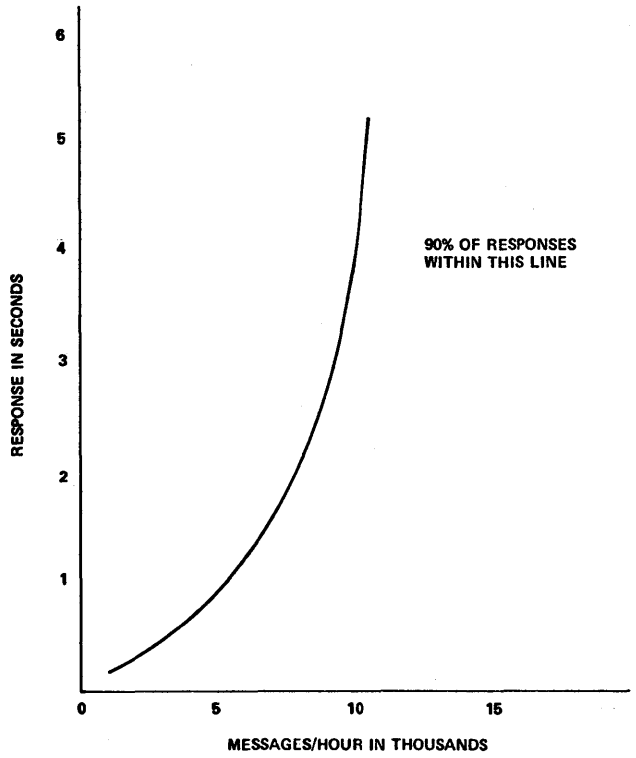


Figure 7—System simulation

300 milliseconds (90 percent of the time). The system performance specification of 3600 interrupts/hour being handled within 500 milliseconds is shown to be met. Other data show that with 1890 interrupts/hour, the system is 20 percent utilized (with no background processing). Input/output is one percent utilized. The Service Modules (application programs) servicing the interrupts had run times ranging from one to ten milliseconds with an average of seven milliseconds. There was an average of two file accesses per interrupt.

Another measure of system performance is central system availability. This again varies greatly from location to location. However, a representative system (as described above) reports that their system is available 423 hours out of approximately 429 per month. (The system is utilized on a two and a half-shift, five-day/week basis.)

Most using locations agree that the major cause of system down time is due to central system software problems. The causes of these problems include: operating system faults, PCOS faults and (most frequently) operator or programmer errors. This problem is being reduced significantly by developing better error handling and recovery routines into PCOS.

THE FUTURE

With a common system structure established in these IBM manufacturing plants, it is now possible to develop additional common system services and applications.

A present concept is a high level **P**rocess and **T**est **L**anguage (PTL) which will allow an engineer with minimum programming experience to efficiently and rapidly apply a satellite computer to control process or test equipment using the central systems resources for compilation.[13] The test or process engineer can develop the control logic using macro statements in high-level languages he can easily learn.[14] Similar statements can be used to send process/test information to the central computer where programmer-written routines can analyze, store and report on the information.

Projected for the future are satellite computer driven machine tools with numeric control processors and post processors resident at the central computer and accessible by the satellite. It will be possible for machine parts programmers to directly enter macro program statements in the satellite computer or a terminal device to make a new part or modify an old. These statements will be sent to the central computer for checking and compilation. The result of the compilation will return to the satellite computer to cause execution by an on line machine tool.

Process Automation programs which reside in the central computer will be developed to directly accept raw development (engineering) information.[15] These programs will then interpret the design information and send process information to satellite computers which will be used to control process and test equipment. Among other benefits, this will allow a very fast response to product/process engineering changes.

A common quality assurance analysis programming system which will reside in the central computer is being considered. This will allow the Quality Control Engineer to easily use complex statistical methods in analyzing process information when a new application is installed on the system. It is possible that the Quality Control Engineer will, in the future, be directly entering high-level (macro) statements in a language like PTL into the satellite computers. These statements would specify data he would want collected and indicate to the central system what kind of analysis to be performed on the data.

Another important future aspect is the more sophisticated use of data management techniques. Such techniques will allow a user to directly access system data banks to retrieve test and/or process data without the aid of a programmer. The user will then be able to

specify statistical programs to operate upon the data and methods for presenting the final output.

The process control central system will be interfaced to other manufacturing information systems which process information pertaining to production, warehousing, maintenance and in-process inventories. Information to be passed to those systems from the process control system includes production yields, equipment down time, units in process, units tested, etc. By completing the tie of process control systems to manufacturing information systems, plant automation becomes possible.[16,17]

SUMMARY

The justification for undertaking a common manufacturing process control approach in IBM was to reduce redundant hardware and software development and implementation costs at each IBM location. This has been accomplished. In addition, it was possible, by pooling ideas, to develop a system superior to that which a single location could have developed. This pooling of ideas and knowledge has ultimately led to a process control solution which, in general, reduces the costs of installing new manufacturing processes and of modifying existing ones.

Although the system as described in this paper is installed and operating at several IBM facilities, it appears to be only a first step toward automation. Almost daily, new ideas are born by one or more users of the system as to how new functions could be added to the system to either further reduce the implementation cost of new applications or to improve the quality of products being manufactured under control of the system. It appears that these ideas, which are born as a result of experience in using the system, are the real justification for the common system approach since many users can now benefit from a single idea and development.

ACKNOWLEDGMENTS

It would be lengthy for the author to acknowledge all individuals who contributed directly or indirectly to the success of the system. However, the key hardware developers were Messrs. R. Watkins, F. Thoburn and T. Rall all of San Jose, California, and Mr. K. Cisewski of Rochester, Minnesota. The primary software developers were Messrs. M. Mauldin and T. Reilly of Kingston, New York, along with Messrs. R. Henry, San Jose, California, and J. Calva, Rochester, Minnesota. Messrs. R. Boydston and C. Connoy of San Jose, California, developed the system simulator discussed in this paper. Special acknowledgment goes to Mr. W. Moore in Harrison, New York (IBM's System Manufacturing Division Headquarters), for the excellent job he performed in initiating and coordinating the entire project.

REFERENCES

1 W ANDERSON
Controlling processes with computers
Automation p 70 January 1969

2 J STUEHLER R WATKINS
A computer operated manufacturing and test system
IBM Journal of Research and Development Vol II No 4 p 452 1967

3 J STUEHLER
Hardware-software trade offs in testing
IEEE Spectrum p 51 December 1968

4 N GAINES et al
Union carbide integrates multi-computer process control
Instrumentation Technology p 49 March 1967

5 J WAUGH A YONDA
NSRL on line computer system
IEEE Transactions on Nuclear Science p 129 February 1968

6 R HORST
The justification of digital process control
Modern Data p 20

7 V LOSKUTOV
Computation technology in automatic control systems
Mekhanizatsiya i Avtomatizatsiya Proizvodstva, No 32p 442 1964

8 C BOND
A digital-computer system for industry
Industrial Electronics p 221 May 1966

9 M MESAROVIC
Multilevel systems and concepts in process control
Proceedings of the IEEE 58-1 p 111 1970

10 J STUEHLER
The devoted, shared or satellite approach for computer control of manufacturing processes?
Proceedings of Western Electronic Show and Convention 8-1 1969

11 J SPOONER
Real time operating system for process control
Instrument Society of America D1-1-DAHCOD 1967

12 P WEILER et al
A real-time operating system for manned spaceflight
IEEE Transactions on Computers C19-5 p 388 1970

13 E JOHNSON J McCARTHY
Development of software systems for automated test equipment
Proceedings of Western Electronic Show and Convention 21-2 1969

14 *Special issue on process control languages*
IEEE Transactions on Industrial Control December 1968

15 R BOEDECKER
The computerized factory
Assembly Engineering June 1966

16 N CHIANTELLA
The systems approach to plant automation
ASTME Vectors 4 5 1968

17 F GUT
Operations control systems
Automation p 55 January 1969

Extending computer - aided design into the manufacture of pulse equalizers

by LAWRENCE A. O'NEILL

Bell Telephone Laboratories
Holmdel, New Jersey

INTRODUCTION

A fundamental difference exists between the techniques used to evaluate the performance of a circuit during the design phase and during manufacture. One strives for reality at all cost during design but wants an inexpensive test in the factory. For computer-aided design, a sophisticated performance criterion is usually selected that reflects how the circuit will function when installed in a system. For economic reasons, however, usually only a few simple tests on individual circuits are performed during manufacture. Since it is difficult to devise a simple test that will be indicative of system performance, the tests performed may be inadequate to reveal unsatisfactory units. Frequently, therefore, the designer specifies extremely tight component tolerances so that performance will approximate the nominal design; this practice can lead to unnecessarily expensive circuits.

To insure adequate performance at reasonable cost, it is desirable to use the same criterion during manufacture that was used during the design phase. The cost of maintaining and using complete systems to test individual circuits has made this approach unacceptable in the past. Now it is feasible to employ sophisticated criteria because many manufacturers, such as Western Electric Company, are using computer operated test facilities. The computer that controls the tests can also contain a program that converts the measurements into the system criterion. This is accomplished by storing in the computer a representation of the remainder of the system. This stored data is combined with data measured for a particular circuit and the criterion calculated. In addition to providing a realistic evaluation of performance, the stored representation makes it possible to simulate worst case field conditions at the manufacturing level. Furthermore, any modifications that are made to the system can be included in the test by simple software changes.

The application of system criteria to manufacturing tests will be illustrated by considering the pulse equalizers designed for a digital transmission system. First it will be shown why a system criterion is required, then the programming of the test will be discussed. It will be demonstrated that even a complicated criterion like error rate can be evaluated on the small computers contained in the test facilities if appropriate algorithms are developed.

DESIGN REQUIREMENTS

A set of pulse equalizers were designed for the T2 digital transmission system which operates at 6.3 million bits per second.[1,2] The equalizer is a linear part of the regenerator shown in Figure 1. Its function is to reshape the pulses dispersed by the cable into a shape suitable for deciding what level was sent. These pulses are then sampled and regenerated for retransmission. The response of a properly designed equalizer is shown in Figure 2. The upper trace is an isolated pulse after dispersion by the cable and the lower is the response of the equalizer. An actual transmission consists of a sequence of these pulses spaced one time slot apart. Since a time slot corresponds to two divisions on the graph, it is evident that equalized as well as unequalized pulses would overlap. If the equalizer output pulse was sampled exactly at the pulse peak, the nonzero values of the pulse in neighboring time slots could interfere with the decisions made in those slots—this is referred to as intersymbol interference. Thus the equalizer design should take into account both variations in sampling time and intersymbol interference caused by adjacent pulses.

The performance criterion for equalizer design should reflect the influence of all factors that might interfere with the correct regeneration of the transmitted information. The design criterion selected was error

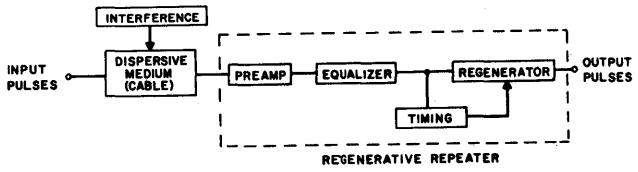


Figure 1—Pulse repeater system

rate, the rate at which errors are made in regeneration. The error rate calculation includes intersymbol interference, sampling jitter, thermal noise, and crosstalk due to neighboring transmission paths. Since the design requirement for each regenerator is less than one error in 10^7 transmitted pulses, the errors cannot be accumulated in evaluating the performance, instead the probability of occurrence is calculated. The probability of error assuming that thermal noise is normally distributed is:

$$P_E = [(2\pi\sigma_N)^{1/2}]^{-1} \int_{V_1}^{V_2} \exp[-V^2/2\sigma_N^2]\, dV \qquad (1)$$

where σ_N^2 is the variance of the noise and the limits indicate the region in which an unambiguous decision can be made.

It is in the specification of the decision region that all degradations other than noise are taken into account. The separation between adjacent signal levels can be maximized by correct sampling. This separation decreases if samples are offset due to jitter in the timing circuit. Furthermore, intersymbol interference reduces the separation because the effect of pulses in adjacent time slots is to modify the isolated pulse height. Crosstalk from adjacent signals influences the separation in the same manner as does the intersymbol interference. It is the need to include all these factors in the criterion that makes the error rate calculation complicated.

The optimization of equalizer design based on minimizing the error rate leads to output pulse shapes that allow correct regeneration with maximum separation between the repeaters. Increased spacing between regenerators, which lowers system cost, also reduces the signal level; thus it is necessary to keep all interfering effects small. The optimized equalizers have limited bandwidth to restrict the thermal noise, and yet they provide pulses that have little intersymbol interference and are relatively insensitive to timing variations. Since an equalizer cannot perform satisfactorily when installed in a regenerator unless all the degradations are small, the manufacturing decision on acceptability of each equalizer should be based on a criterion like error rate.

MANUFACTURING TEST

Programming the test computation

This discussion is based on realizing the entire error rate criterion used in the design phase. As experience in testing is acquired, it may prove satisfactory to use a subset of these operations as a criterion, thus further reducing the computation time. There were two basic resources for the implementation of error rate as a manufacturing criterion. First, there existed a general-purpose, digital Pulse Transmission Program (PTP) that could be used to calculate system error rate using measured equalizer data. Second, the standard programs on the **C**omputer **O**perated **T**ransmission **M**easuring **S**et (COTMS)[2] will automatically measure the gain and phase of the equalizer at a predetermined set of frequencies. Thus the primary task was to extract from the general-purpose program a subset of instructions that could calculate the error rate in a reasonable length of time on the PDP–9 computer contained in COTMS. It was imperative that the new program be short and efficient because the execution times for the instructions are relatively long and the storage capacity limited on the PDP–9.

The programming task was divided into two sections; the first was to reduce the run time of the PTP program by restricting its generality, and the second was to rewrite this program to account for the limitations of the PDP-9. Let us review the steps taken in PTP to calculate error rate so that the advantages of the modified algorithms can be demonstrated.

First, a Fourier transform of the input pulse is calculated. In order to consider the pulse to be isolated in subsequent calculations, it is necessary to separate the pulses by 80 time slots. This separation, for the bipolar code used, requires that a fundamental frequency of 39 KHz be used for the transform. The frequency domain representation of the pulse can then be combined with frequency domain representations of the cable and equalizer. Measured gain and phase data can be loaded into the program to represent the equalizer. The noise power is calculated and then an inverse

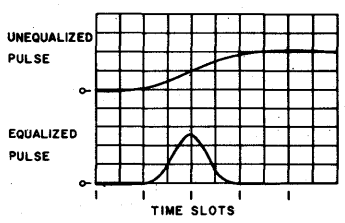


Figure 2—Equalizer response

transform is taken to find the equalized pulse shape over 80 time slots with 20 gross points per slot. The subsequent calculations are then performed in the time domain.

Before searching for the peak of the isolated pulse, the temporal resolution was improved by interpolating additional points to provide 80 fine time points per time slot. The peak is considered to be the correct sampling point and time slots are marked off with respect to it in both directions. Since the error rate depends on the intersymbol interference caused by preceding and following pulses, it is necessary to consider all sequences of a preselected length that can occur in the input code. There existed only 40 sequences five pulses long in the ternary[1] code used—bipolar with six zero substitution. An error rate is calculated for each sequence and then an average error rate is calculated using the probability of occurrence of each sequence.

To account for timing jitter, a sampling distribution is assumed; then the actual error rate for each sequence is found by averaging over the allowable timing variations. For each sequence, a separate decision region is obtained at the peak and at each allowed timing off-set by calculating the intersymbol interference. The intersymbol interference for each fine time point (t) is:

$$I(t) = \sum_{i=-1,\, i\neq 0}^{3} M_i f(t+iT) \qquad (2)$$

where M_i is the amplitude of the transmitted pulse in time slot i,
f is the isolated pulse response, and
T is the width of a time slot.

To account for crosstalk and any miscellaneous degradations, a fixed percentage of the remaining decision region is subtracted. The probability of error is then calculated for each time offset using equation (1). Then the error rates are averaged first over the allowable sequences and then over the timing variation to find the performance of an equalizer.

The time required to compute an average error rate by this method on the CDC 3300 computer was approximately four minutes.* This time was reduced to less than four seconds on the 3300 by two modifications to the program. Thus, a program that could be executed in a reasonable length of time on the PDP-9 was obtained without sacrificing accuracy or eliminating any factors that influence the error rate. This reduction was accomplished by using a precomputed set of test conditions and calculating only the data absolutely needed to determine error rate. This implies that test conditions that would yield a reliable indication of performance, such as worst case, would be preselected for each equalizer. These conditions include cable type, length and temperature, pulse shape, and any repeater circuitry not present in the test equalizers. The test conditions would then be combined using the standard PTP program into a data set suitable for loading into the modified program. Much of the generality required in PTP to accomplish these operations can thus be eliminated since the system simulation is stored as data.

The other major departure from the PTP program is that the entire pulse shape is not computed but only those time points that influence the error rate calculation. Much of the time in the PTP program is expended calculating the pulse response from the frequency domain representation of the system by inverse Fourier transform. These time point calculations are kept to a minimum by first searching for the peak of the pulse with widely spaced time points, then refining its location, and finally computing with respect to the peak the other points needed in the error rate calculation. A simple search algorithm can be implemented because the response of a correctly constructed equalizer can be computed. For a manufactured equalizer to be acceptable, it must possess approximately the same pulse shape. Thus, an approximate location for the peak can be used as the starting point in the search. Furthermore, small amplitude ripples in the pulse shape can be ignored in the search by selecting a decision threshold that is an appreciable percentage of the approximate pulse height. Initially, one time slot wide steps are taken alternately forward and backward from the starting location. After a value exceeding the threshold is found, the search continues in the same direction until the slope reverses. The location is then refined by taking smaller steps and finally by a quadratic interpolation. It was the drastic reduction of the number of inverse transforms required, using this search procedure, that accounted for most of the reduction in execution time.

Test procedure

Let us now consider the steps necessary to measure the equalizer and compute its performance.

1. Measure loss and phase as a function of frequency on COTMS.
2. Combine measurements with precomputed data and calculate the noise power.

* This computation time made iterative optimization of the equalizers prohibitively expensive during the design phase. Instead, a hybrid simulation was employed so that the entire waveform could be computed in the time domain without requiring transforms to be evaluated.

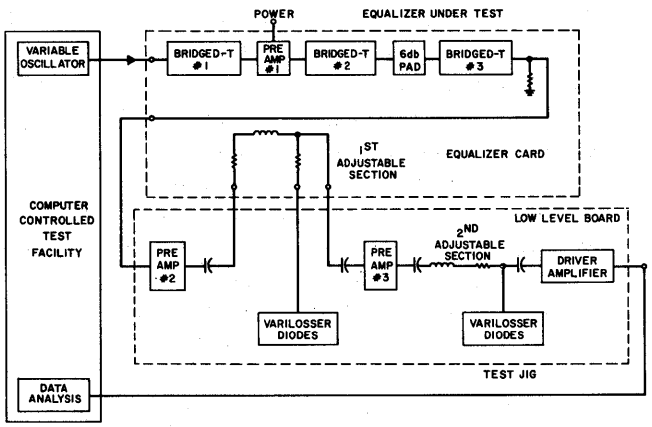


Figure 3—Equalizer test configuration No. 1

3. Compute pulse shape at required time points and determine the intersymbol interference.
4. For all allowable pulse patterns calculate an average error rate.

The entire equalizer can be measured by inserting the equalizer card into the low level board of a reference repeater, as shown in Figure 3. A separate reference repeater would be required for each cable type to be equalized because different circuitry is used for each type. Since the equalizer design includes a section that automatically adjusts for variations in cable length, this section must be biased to correspond to the cable length included in the program. The use of the repeater as a test jig insures that the influence of all spurious coupling paths will be included in the measurements. An alternate measurement approach that could be used is shown in Figure 4. This approach is not as close to field operation as is the first procedure but it is more easily implemented in the factory. A single test jig can be used for all equalizers and no bias adjustments are required. The software could be modified to make the fixed equalizer test identical to the first approach but the passive components in the adjustable would have to be tested separately. Either approach should be acceptable and the timing investigations would not be appreciably changed whichever one is used.

COTMS will provide a list of the gain and phase measured at a predetermined set of frequencies; it automatically removes systematic errors and minimizes the influence of noise by repeating the measurement and averaging. Since the digital program requires continuous phase rather than just the principal value provided by COTMS, the correction must be computed. To determine unambiguously the correction, the frequency range must be sampled to provide at least two frequencies per revolution. The measurement of these data at 109 frequencies required between 30 seconds and 3 minutes, depending on the noise present in the test configuration.

The data measured for the equalizer are used to compute the noise power of the system. Then the data are multiplied by the stored frequency data that represent the remainder of the system. The resulting frequency data are then a characterization of the entire system that precedes the regenerator, including the effects of cable and input pulse shape.

The output pulse shape at the time points required for the error rate calculation are obtained by inverse Fourier transform from the combined frequency data. Only three points about the peak and in each of the neighboring time slots had to be calculated by inverse transforms to account for the predetermined amount of sampling jitter. The finer resolution time points were then found by interpolation. The pulse data in the neighboring time slots is used to compute the intersymbol interference so that the average error rate can be determined exactly as described previously.

EQUALIZER TEST RESULTS

A test program that computes error rate from gain and phase measurements of an equalizer was extracted from the PTP program. The considerations in writing this program for solution on the CDC 3300 computer were to minimize both run time and the storage requirements. The execution time, using the measured data, to obtain the error rate is 3.873 seconds on the CDC computer. The calculation is identical to that performed with PTP; 109 harmonics are used in the transform and all pulse patterns that can occur using three preceding and one following pulses are evaluated in the error rate calculation.

The program has not been rewritten as yet for the PDP-9 computer but estimates of execution time have

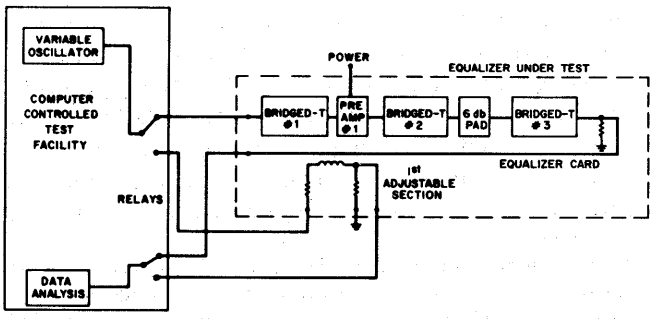


Figure 4—Equalizer test configuration No. 2

been made. These estimates are based on the time required to compute a time point by inverse transform since this is the most time consuming operation in the program. This basic calculation was isolated and then programmed for the PDP-9 computer. It was then run on the standard computer and on one with an extended arithmetic capability. The results are shown in Table 1.

Since a PDP-9 with extended arithmetic will be included in future test facilities, a crude estimate of total run time was made based on the extended arithmetic speed. The execution time on the CDC computer was multiplied by the ratio of transform time on the PDP-9 to transform time on the CDC and the result is less than 38 seconds. This estimate indicates that the time to compute error rate would be of the same magnitude as the time required to measure the data. A subset of the measurement program has been run on the PDP-9 computer and it was established that the time estimates are reasonably accurate.

In addition, the storage required for the test program has been reduced to 4761_{10} locations, which is small enough to be loaded into the PDP-9 at the same time as the standard measurement program.

TABLE I

Time to Compute Inverse Fourier Transform (109 Harmonics)

CDC 3300	73 milliseconds
PDP-9 (Standard)	2100 milliseconds
PDP-9 (Extended Arithmetic)	700 milliseconds

CONCLUSION

It is feasible to apply an error rate specification to the equalizers constructed by Western Electric. The computer contained in COTMS can be used to calculate error rate from the frequency domain measurements of an equalizer in a reasonable length of time. The additional time for calculation is estimated to be less than 40 seconds. Since the time is directly related to the number of frequencies used, it might be further reduced through analysis of the influence that the number of frequencies measured have on the error rate calculation.

Since the results were so favorable, the approach will be implemented on a PDP-9 computer. The next steps are the writing of a PDP-9 version of the complete measurement program and the determination of a set of test conditions for a specific equalizer. These test conditions, which include the entire system except for the equalizer, can be used to simulate worst case field conditions at the manufacturing level. Furthermore, any changes in channel characterization, such as transmitted pulse shape, can be included in the test by simple software modifications. It will then be possible to evaluate both the technical and economic advantages of this approach to equalizer testing.

As this example has demonstrated, it has become feasible to apply the same criterion in manufacture that was used in the design phase by using the full power of a computer operated test facility. Thus, simple tests can suffice to obtain a reliable indication of system performance.

ACKNOWLEDGMENT

The author gratefully acknowledges the advice and assistance he has received from J. Chernak, E. M. Butler, D. R. Smith, E. M. Underwood and R. G. Schleich. He is particularly indebted to Mrs. R. M. Allgair who wrote the programs and performed the timing evaluation.

REFERENCES

1 *Transmission Systems for Communication*
Bell Telephone Laboratories Incorporated Fourth Edition
Chapter 27 pp 626-676 1970

2 J H DAVIS
T2: a 6.2 Mb/S digital repeatered line
IEEE International Conference on Communication
Record pp 34.9-34.16 1969

3 W J GELDART G D HAYNIE R G SCHLEICH
A 50 Hz—250 MHz computer-operated transmission measuring set
Bell System Technical Journal Vol 48 No 5 May-June 1969

Finite state automation—Definition of data communication line control procedures

by DINES BJØRNER

IBM Research
San Jose, California

INTRODUCTION

The notions of finite state automata, state transition graphs and tables and the set of regular languages being accepted (generated) by such automata are well known. But for some reason these notions have not been rigorously applied in the definition of data communication line control procedures. It is the objective of this paper to do so and to show the naturalness of this approach. We claim that we thereby arrive at a complete, precise and unambiguous definition. Others have attempted this before us. They have, however, not used the descriptional tool of finite state automata.[1,2,3] Any one or all of these references thus form the basis on which we will compete and we shall use essentially the line control procedures which these documents set out to define.

It is shown how a multilevel hierarchical definition of data communication line control procedures by means of finite state automaton graphs leads to easily understandable communication standards, complete and unambiguous specifications, well-structured relations between the syntactic, semantic and pragmatic issues and embodies straightforward extensibility features. It is further shown how the definition is the basis for an automatic conversion into all conceivable schemes of implementation: active hardware logic, microprogram controls or software procedures. We specifically mention the duality of implementing the generator (transmitter) and acceptor (receiver). We finally show the suitability of our technique to that of specifying the proper interlocking of asynchronously operating full-duplex schemes, this latter scheme has applications to the well-behaved, rather than well-policed, control of systems of many parallel processes.

METHODOLOGY

We shall describe the definitional approach through an example and will use a "multilevel iteration, top-down technique"[4] whereby we first describe the total system, then its major subsystems and subsequently the components in each of these. We finally give the complete (character oriented) logic of each component. Exactly what we mean by this hierarchy will be apparent as we go along. We may also refer to this presentation technique as one demonstrating "layers of abstraction."[5] Needless to say, we would like to see such a methodology applied in other areas.

Our example is that of a TWO WAY ALTERNATE (or: half duplex) NONSWITCHED MULTIPOINT data communication system with CENTRALIZED OPERATION. (See Ref. 3 sections 5.5 and 6.4-5 and Ref. 1, page 20, transparent-text mode.)

SPECIFICATION HIERARCHY

First level

As our first level we show Graph 1 on Figure 1.

C denotes the *state* of the system in which the **c**entral device **c**, e.g., a computer data transmission control unit is *communicating* a *sequence* of *characters* u (a *word*, a *sentence*) to one or more **r**emote devices **r**, e.g., keyboards and displays. The communicated sequence u in turn *transfers* the system to state R. In state R zero or one remote device is communicating another sequence of characters v to the central device. This in turn brings the system back to state C. We note that while c is *sending* (or transmitting) u, r is *receiving* u and vice versa—r is sending v which is received by c.

GRAPH 1

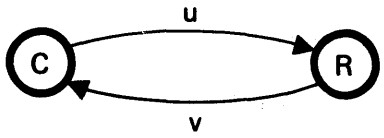


Figure 1—GRAPH 1
First level data communication system. C: Central- and R: Remote System State

Yet another equivalent characterization is that each of the two components, the central and the remote subsystem, can take on either of two mutually exclusive first level states or *modes*—receiving or transmitting. Note that no device can simultaneously be receiving and sending. This half duplex or two-way alternate scheme is truly reflected by the state transition graph approach.

A final note in this section concerns the nature (format, structure) of words $u \in U$ and $v \in V$, where U and V are *languages*. It is the precise definition of U and V that concerns us.

We start out by defining an *alphabet* Σ (vocabulary), a finite set of (*terminal*) symbols (where the use of the word terminal has no connection with the fact that we often call remote devices terminals). An n bit coding allows for up to 2^n symbols.

In our example the alphabet is:

$$\Sigma = \{\underline{\text{eot}}, \underline{\text{enq}}, \underline{\text{soh}}, \underline{\text{stx}}, \underline{\text{dle}}, \underline{\text{syn}}, \underline{\text{itb}}, \underline{\text{etx}}, \underline{\text{etb}}, \underline{\text{ack}}, \underline{\text{nak}}\}$$

$$U\Sigma_\alpha U\Sigma_{\bar{\alpha}} = \underline{\text{bcc}}$$

Underlined character triplets denote one symbol, and except for bcc (block check character set, which may take on any bit coding whatsoever), they are all *control* symbols as distinguished from Σ_α and $\Sigma_{\bar{\alpha}}$. Σ_α denote symbols α allowed in either prefix, header or text sequences. For reason of simplicity we omit a specific breakdown of Σ_α in sub-subalphabets. $\Sigma_{\bar{\alpha}}$ denote all other symbols of even or odd bit parity.

A string is a sequence of symbols. The (infinite) set of all finite length strings including the null string over Σ is denoted by Σ^*. U and V are true subsets of Σ^*.

$$U \subset \Sigma^*, \qquad V \subset \Sigma^*$$

From the next sections it will now be clear exactly how words in U and V are defined.

Second level (See Graph 2 on Figure 2)

The subsystem r denotes one or more (but a finite number of) remote devices. Transactions between c and r can be grouped in two major classes: Polling- and Select-transactions.

In *polling* one remote device r is asked by the central device c—through the use of a *polling sequence* p—to communicate to the central device whatever *polling message* pm or *termination sequence* t it might wish to transfer. p brings the system from state Ci to state Rp. The system goes to state Cp upon c receiving pm and c sends back appropriate *polling message replies* pmr. If these are erroneously received (i) by the system in state Rp the remote device sequences a *request reply* rr to c. The system then transfers to state Cp. Each of the words p, pm, pmr, t and rr may range within corresponding well defined languages: Lp, Lpm, Lpmr, Lt and Lrr.

In *selection* one or more remote devices are requested by the central device to enable themselves for subsequent *selection messages* sm communicated to them, either one-by-one or on a broadcast basis, from the central device. The selection process consists of one or more *selection sequences* s each in turn followed by an *initial selection reply* isr. These sequences bring the system back and forth between states Rs and Ci with the last such isr bringing the system to state Cs; whereas sm and their replies, the *selection message replies*

GRAPH 2

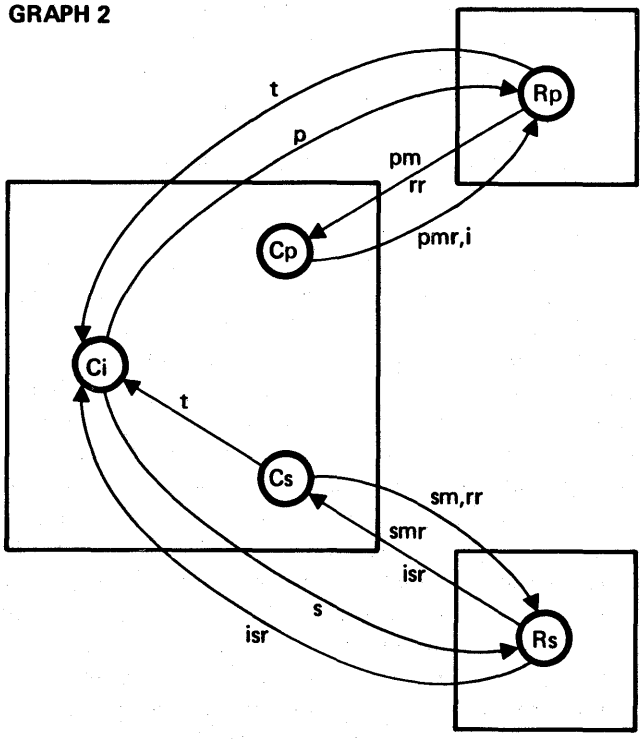


Figure 2—GRAPH 2
Second level data communication system p: polling- and s: selection state

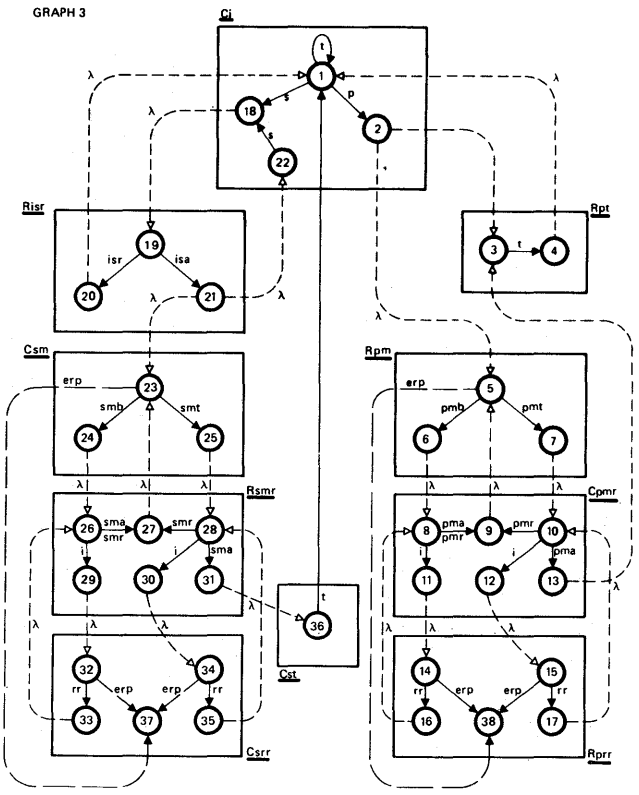


Figure 3—GRAPH 3
Third level data communication system. Transition from error-recovery states 37 and 38 to initial system state 1 not shown

smr, cause transitions between Cs and Rs. The system in Cs finally terminates selection mode by sending a t word to the remote devices causing a transition from Cs to Ci. Like in polling we may speak of well defined languages Ls, Lisr, Lsm, Lsmr, Lrr and Lt.

Third level (See graph 3 on Figure 3)

We now proceed to explode substates Ci, Cs, Cp, Rs and Rp into submachines. In Graph 3 we have clearly marked these and their relation to Graph 2.

First we cover some omissions from Graph 2 and subsequently we subdivide further the Second Level Languages.

1. When e.g., s in Graph 2 causes a transition from Ci to Rs, then we actually mean to show: an s sequence of transitions in Ci followed by a line-turnaround transition λ from Ci to Rs. We shall later explore this latter type of transition in depth.
2. States Rsir (substate of Rs). isr (isa) denote an *initial selection reply* sequence indicating a rejection (acceptance). After isa, a λ will bring us to either state 22 or 23 depending on whether or not the central device wishes to select more than one device. The (2) nondeterministic λ transitions from state 21 are thus resolved by conditions in c which are combined with λ.
3. States Rpm (Csm). Polling (selection) *messages* may be blocked or nonblocked. A series of blocked submessages, pmb (smb), will cycle us around the 4 state loop 5-6-8-9-5 . . . (23-24-26-27-23 . . .). The last blocked submessage or a nonblocked message is here referred to as pmt (smt), t denoting text.
4. pmb's and pmt's (smb's and smt's) are either (1) accepted pma (sma), (2) rejected pmr (smr), or either (3) the message transmitting (remote or central) device cannot interpret the reply (i) or (4) the message receiving device has not been able to detect the end of the message block thus causing a timeout (i).
5. In case of an i-type reply the message transmitting device may choose either to communicate a *request reply* rr or enter some *error recovery procedure* erp. This latter choice may also result from repeated message reject replies. After suitable error recovery procedures the system goes to state 1.
6. After the last message block has been acknowledged—or if in polling the polled device has no message to send—a *termination* sequence t brings the system to state 1.

Fourth level

We now first expand the message generating/recognizing super state Csm and Rpm, which like Cpmr and Rsmr or Csrr and Rprr can be treated by

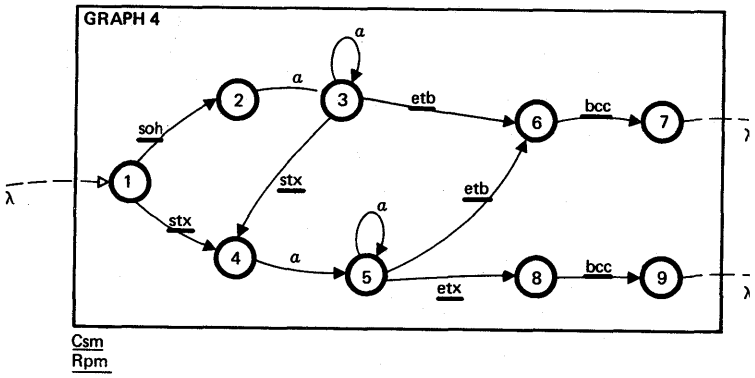


Figure 4—GRAPH 4
Fourth level message transfer subsystem

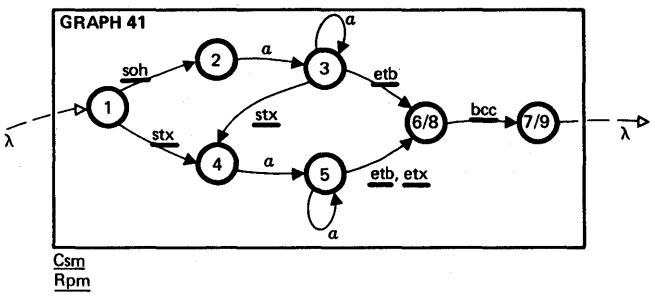


Figure 5—GRAPH 41
Fourth level message transfer subsystem

showing one graph. One then substitutes these graphs into the less detailed third level graph. The latter and remaining superstates are subsequently treated in depth.

In Graph 4 (Figure 4) the fourth level state 1 (4.1) equals either 3.23 or 3.5, 3.24≡4.7≡3.6 and 3.25≡ 4.9≡3.7.

Graph 4 is identical to the regular expression:

$$(\underline{soh}\ \alpha^+(\underline{etb} \cup \underline{stx}\ \alpha^+(\underline{etb} \cup \underline{etx}))\ \cup \underline{stx}\ \alpha^+(\underline{etb} \cup \underline{etx}))\underline{bcc}.$$

Graph 4 shows that our messages may be of either of five constructions.

(pmb/smb)	soh α^+ etb bcc	or
(pmb/smb)	soh α^+ stx α^+ etb bcc	or
(pmt/smt)	soh α^+ stx α^+ etx bcc	or
(pmb/smb)	stx α^+ etb bcc	or
(pmt/smt)	stx α^+ etx bcc.	

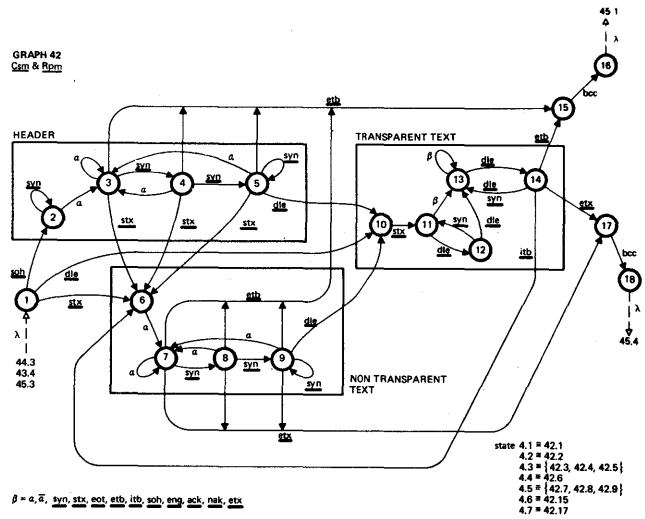


Figure 6—GRAPH 42
Fourth level message transfer subsystem. Detailed version distinguishes between transparent and non-transparent texts

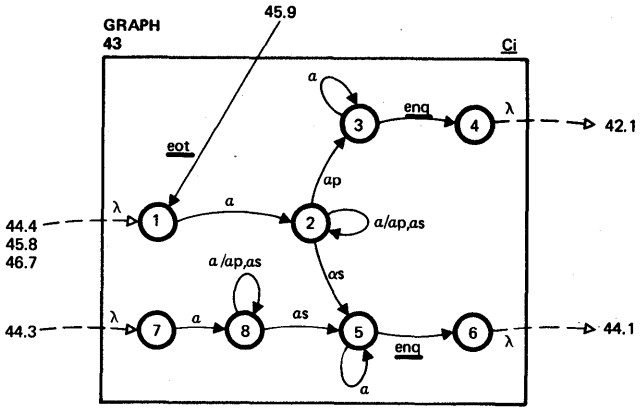


Figure 7—GRAPH 43
Fourth level establishment selection and polling subsystem

α^+ denotes a finite length string of one or more α's. Enclosed between soh and stx or etb an α^+ string denotes a header; in front of some enq, ack, nak α^+ denote a prefix; otherwise α^+ denotes text data.

The (minimal) finite state automaton (*fsa*) just recognizing either of these 5 types of strings is shown in Graph 41 (Figure 5).

We have chosen to split into *separate final states*, states 4.7 and 4.9 due to the *different* subsequent *actions* on affirmative replies.

We shall subsequently show a rather complete fourth level graph [42 (Figure 6)] denoting the generation of messages possibly involving *transparent text data* and messages using intermediate synchronization characters, syn. Basically we use the sequence dle stx to denote the start of transparent data, dle syn to denote one synchronization character in transparent text data, dle dle to denote one transparent text data dle character and dle ext or dle etb to denote end of transmission text or block. The first dle stx must by convention be preceded by at least two syn's. In transparent

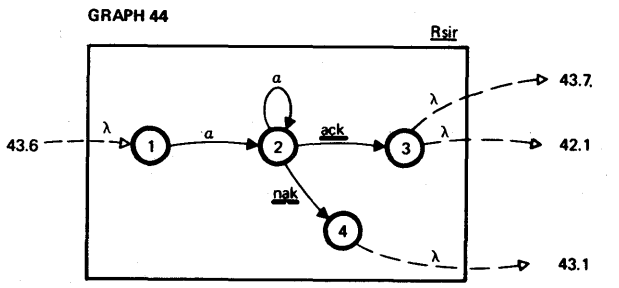


Figure 8—GRAPH 44
Fourth level establishment reply subsystem

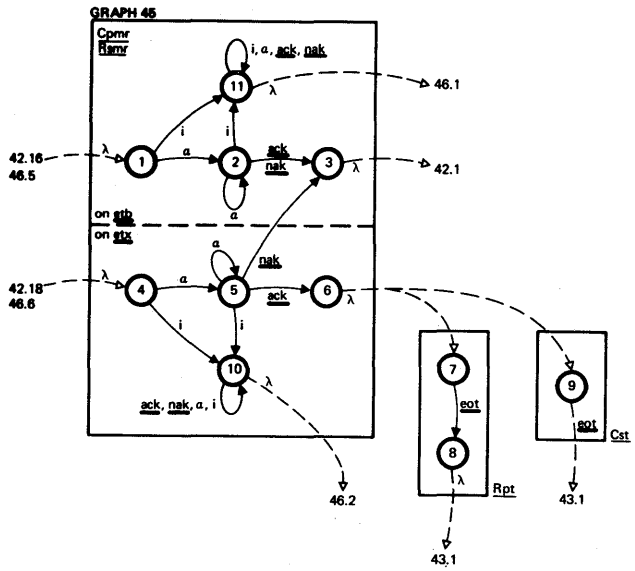


Figure 9—GRAPH 45
Fourth level delivery termination and terminate subsystem

text mode a dle itb signals a return to nontransparent text mode (dle≡data link escape).

For the sake of completeness we next show the detailed content of each of the remaining superstates: Ci, Rsir, Cpmr or Rsmr and Csrr or Rprr.

$\alpha/\alpha p$, αs denote the set $\Sigma\alpha$ excluding both αp and αs, αs (αp) is the character in an α^+ enq sequence which designates it as a selection (polling) establishment sequence (request). A specific installation will use individual α symbols in the string α^+ αp α^* enq (α^+ αs α^* enq) to select one (one or all) remote device(s)—and to further select which subdevice is being addressed—such as either card or paper tape reader

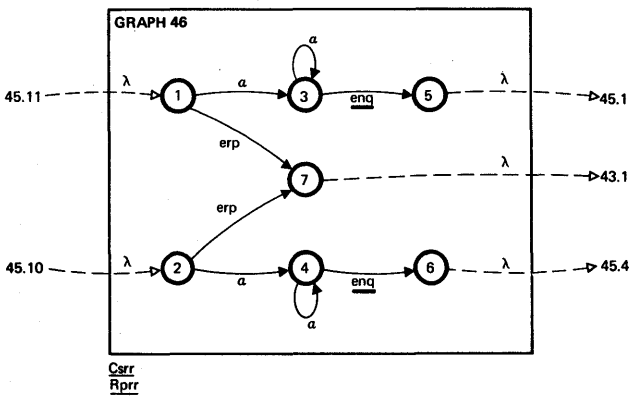


Figure 10—GRAPH 46
Fourth level invalid reply subsystem

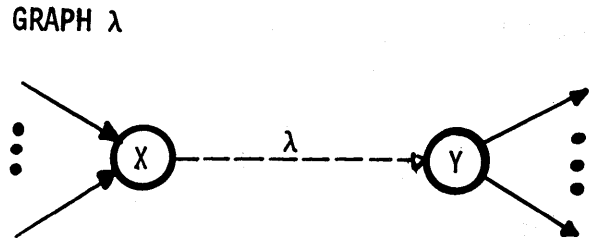


Figure 11—GRAPH λ
Fourth level line turnaround subsystem idealized

(punch) or keyboard (printer) within a 1050 system (enq=enquiry).

LINE TURNAROUND

We have so far shown the line turnaround function just by the transition λ, see Graph λ, Figure 11.

If we address ourselves to a specific system implementation we may then show λ far more explicitly, e.g., by either of two half-duplex extremes shown in Graphs λ1 and λ2 (Figure 12).

In the first scheme (Graph λ1) the transmitting device terminates each communication by 2 or more syn patterns subsequently followed by a *mark* condition; after at least two received syn patterns the receiving device now becomes the transmitting device and first sends at least 2 syn patterns. While this device was in receive mode it constantly held its send line in mark condition.

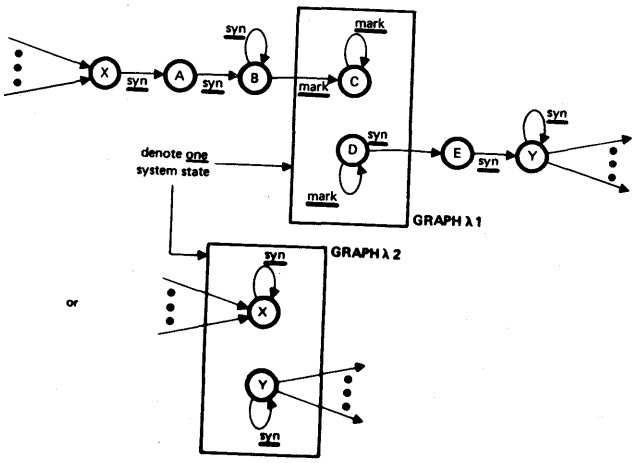
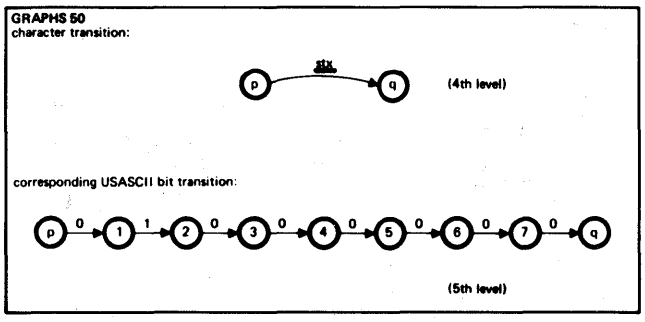


Figure 12—Fourth level line turnaround subsystem
Graph λ1: Conventional half duplex
Graph λ2: Synchronous transmit/receive



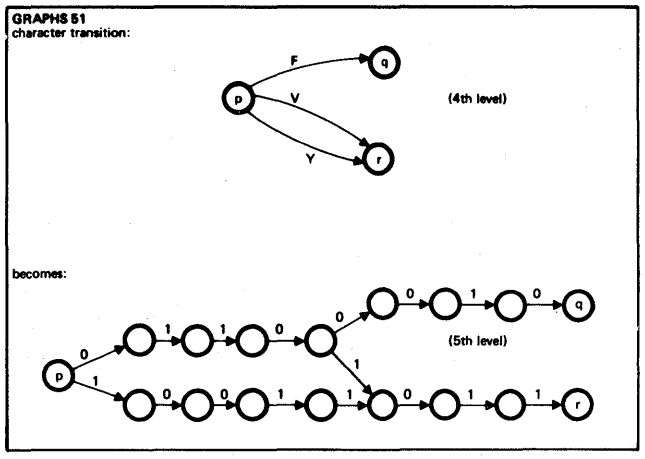


Figure 13—GRAPHS 50 and 51
Fifth level character subsystem

In the latter scheme (Graph λ2) a device in receive mode always transmits syn patterns and a shift to transmit mode can thus be effected immediately after receipt of the last character before syn.

CHARACTERIZATIONS

We now pause to characterize the type of transition graph that we have so far produced.

Scope of definition

The graph(s) on level four define all valid (correct) sequences which we may encounter on a data communication link under control of a TWO WAY ALTERNATE NONSWITCHED MULTIPOINT system WITH CENTRALIZED OPERATION. The definition uses a line character as its 'smallest' indivisible token.

Bit versus byte oriented definition

We may break the level four graphs down even further; our next more detailed level may take a bit as the smallest indivisible token (see Graphs 50 and 51, Figure 13).

We choose however not to go to the fifth level for several reasons. One is that we tend to think of line control procedures as character oriented. Another is that in our implementation we *decompose* the fifth level bit-to-character assembly out of the general graph and implement it as a simple shift register (serializer-deserializer) from which characters are read/decoded in parallel over all bits.

Alternating replies

In certain line control schemes text messages are *acknowledged* on an *alternating* basis, i.e., either ack0 for even-numbered blocks and ack1 for odd-numbered or ack*n* where *n* ranges over 0-9. We have not shown this in our graphs but will now do so using Graphs 42 and 45 (Cpmr) (See Graph 6, Figure 14).

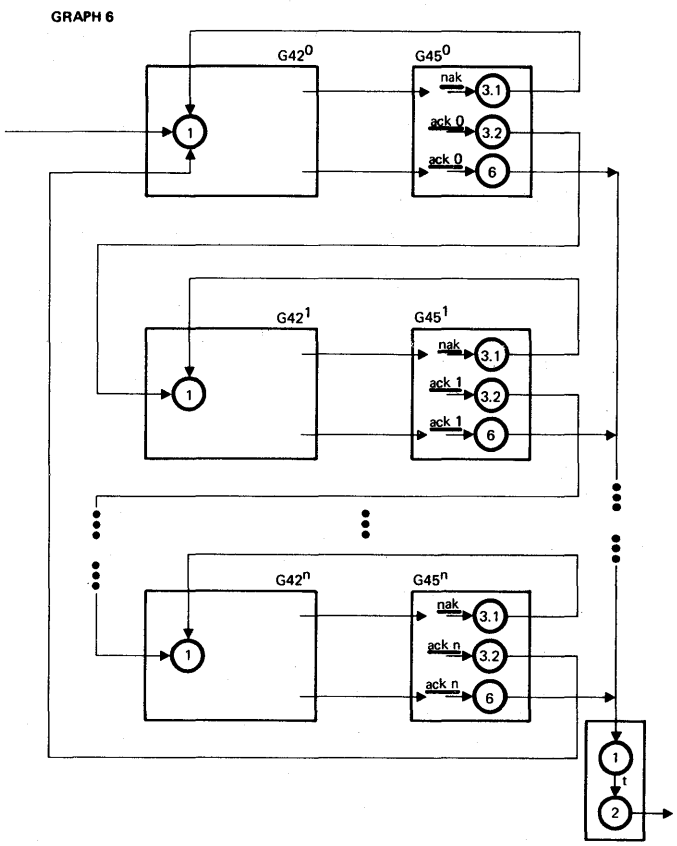


Figure 14—GRAPH 6
Simplified fourth level alternating acknowledgment subsystem

Instead of showing repeated instances of the same graph $G42^{0-n}$, $G45^{0-n}$) one can *decompose* Graph 6 into one single instance of Graphs 42 and 45 combined with a single counter (0-n). Again this is the way we would implement such a system. But even on Graph 6 or on the suggested decomposed graph we have not quite shown the fact that a nak response often requires the error-free *retransmission* of the same block, character-by-character as was erroneously communicated. Here perhaps an English worded definition is superior to a graph description. One could suggest that the latter would make use of a queue or buffer into which we put characters as we generate them. The buffer or queue content would be reused if a nak response was received. Otherwise it would be overwritten during the next transmission.

Half duplex

The *graph definition* truly *depicts* the *half duplex* nature of our *line usage*. We shall return later with thoughts on how a transition graph definition might be used to define full duplex line control procedures.

Definition, implementation, syntax, semantics and pragmatics

The graph definition is the implementation

It is well known that the *transition graphs* we have shown so far can be interpreted in a mechanical way as either *hardware* implementable *sequential circuits* or *soft*-ware implementable *tables*.

First, however, some words on *syntax*, *semantics* and *pragmatics*. The graphs represent the syntax or the grammar defining all valid character sequences. In the implementation one is also interested in:

- What to do with invalid character sequences?
- How to react to any sequence, valid or invalid?

So far our graphs show only generative capability but we can easily augment the graph or its corresponding tabular representation to show the answers to both of the above questions.

Take for example Graph 42 shown on Table 1 (Figure 15).

To give informally stated examples of what we mean and imply by actions/semantics we list a few examples:

a. *Start-of-header;* set up suitable (hardware) buffer, initialize (hardware) variables.

≡ bcc

State: \ Input:	soh	stx	α	dle	syn	etb	etx	β	itb	bcc
1	2 a	6 b		10 c						
2			3 d		2					
3		6 f	3 e		4	15 k				
4		6 f	3 e		5	15 k				
5		6 f	3 e	10 g	5	15 k				
6			7 h							
7			7 j		8	15 L	17 m			
8			7 j		9	15 L	17 m			
9			7 j	10 n1	9	15 L	17 m			
10		11 n2								
11	13 p	13 p	13 p	12	13 p	13 p	13 p	13 p	13 p	
12				13 p	11					
13	13 q	13 q	13 q	14	13 q	13 q	13 q	13 q	13 q	
14				13 q	13	15 s	17 t		6 r	
15										16
F 16					45.1					
17										18
F 18					45.4					
19						15	17			

The letters denote appropriate actions, i.e. semantics.

Unspecified (blank) entries all have next state transition to the error state 19 with some appropriate action, suitably depending on the present state.

Figure 15—TABLE I
Tabular representation of state transition graph 42. The letters denote appropriate actions, i.e., semantics. Unspecified (blank) entries all have next state transition to the error state 19 with some appropriate action, suitably depending on the present state

b. *Start-of-text;* no header, set up *out* device/text buffer, initialize text variable.
c. *Start-of-transparent-text;* no header, no non-transparent text, set up *out* device/transparent text buffer, initialize transparent text variable.
d. *First header character*, hardware count: = 1.
e. *Subsequent header character*, hardware count: = hardware count +1; buffer full? or output *in* character to *out* device.
f. *End-of-header*, store hardware count, interpret header; *start-of-text*, set up text buffer and initialize text variable.
g. *End-of-header*, ..., *start-of-transparent-text*, so far no nontransparent text, set up *out* device/text buffer and initialize variable, disable parity check.
h. *First text character;* text character: = 1; output *in* character to *out* device/buffer.
j. *Subsequent text character;* text character: = text character +1; ....

k. *End-of-transmission-block,* no text, interpret header, wait for block check character, disable parity check.
q. *Transparent text data,* use as such.
r. *End-of-transparent-text-data, start-of-*(nontransparent)*-text* reinitialize variables.

We have and shall not define the pragmatics of line control procedures rigorously (formally).

Some of the pragmatics here are concerned with the length of prefixes, headers, and text data and with the principles of retransmission and error recovery procedures.

Through this augmentation we have converted our graph from a *generator* into a *transducer:* a machine accepting and transducing (controlling the translation of) the generated sequences. This machine can be implemented either strictly in wired logic, as a micro program in a general purpose transmission control unit or as a software table switching/look-up mechanism. We shall comment further on this.

Completeness

We wish to demonstrate the COMPLETENESS of our definition. That is, we can *prove* that we have considered all possible character sequences and rigorously defined syntactically and semantically which are correct and what to do with valid as well as invalid sequences. Our definition and our implementation is therefore *unambiguous,* each taken separately or both taken together. The argument goes somewhat like this: (1) There is a finite number of states (nodes) in our graph. (2) In each state only a finite number of valid input (transducer/recognizer) or output (generator) characters is possible. Thus we only have to make a finite number of decisions in building our graph. The graph *is* the proof.

The design style and structure that the graph approach imposes on us is appealing.

Specification changes

Changes in the definition of line control procedures can be readily effected. The graph approach immediately reflects all nodes which may be affected by a change. This is true for generators as well as for recognizers/transducers.

Generator and recognizer duality

Now that we have seen that the graphs represent both the generation *and* the transduction of data communication transactions we can mention the possibility of letting both phases be software or hardware controlled through the same graph in both the master (c) and remote (r) systems. That is, the total collection of state transition Graphs 42-46 connected as shown by Graph 3 is present in either type of device. Subgraphs Ci, Csm, Csrr, Cst and Cpmr will be used by system c as generators whereas any device in system r will use them as transducers and vice versa—subgraphs Rsir, Rsmr, Rpt, Rpm and Rprr will be used by system c as transducers whereas any one device in system r will use them as generators.

To use a graph as a generator may not seem to have a practical meaning in any particular instance of a transaction. The generator generates all correct transactions, not a particular one. But somehow, someone (be it a human being at a terminal or the automatic program in a CPU at the central system) generates a particular instance based on a few parameters (v) such as (1) poll or select, (2) device address, (3) header/no header, text/no text just header, (4) possible embedded transparent text segments, (5) blocked/nonblocked messages, (6) length of prefixes, headers and text segments and (7) pointers to prefix (p), header (h), and text (t) segment buffers. What we mean by the generator generating a particular instance should now be more clear.

Let us visualize a system (Figure 16) where c selects r and sends messages to r.

Gc is the generator {Ci, Csm, Csrr, Cst}, *Tc* is the transducer {Rsir, Rsmr}; *Tr* is the transducer {Ci, Csm, Csrr, Cst} and *Gr* is the generator {Rsir, Rsmr}.

Gc uses parameters v and subsequently also link parameters from *Tr* while stepping through the appropriate transition graphs (sequential machines, micro programs or software tables).

Gr uses link parameters from *Tr* in its stepping through appropriate graphs.

By link parameters we mean the information shown in Graphs 42-46 relating the outcome (final state) of some graph (say 42.16) to one or more initial states in connected graphs (here 45.1).

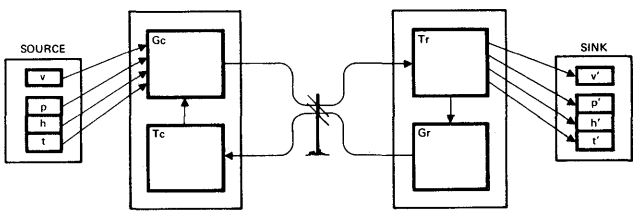


Figure 16—Two way alternate non-switched multipoint system with centralized operation of subsystems r from subsystem c

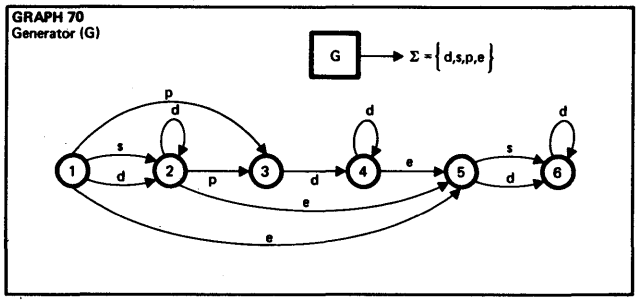


Figure 17—GRAPH 70
ALGOL 60 number g enerate graph. Fourth level

The stepping process thus uses the parameters in order to resolve next generator output in cases where more than one next output character is correct. For example, in Graph 4, state 1, if the v parameter indicates 'header,' then soh is output and the generator graph action directs the control to fetch pointer to and length of header buffer from v and to start outputting header information. The header characters are matched against the α's, i.e., a state transition from 4.2 to 4.3 and then recursively to 4.3 occurs unless the header buffer contains invalid information, i.e., characters different from, say, α. When the header buffer is exhausted the control again looks at v to see whether any text transmission is required, etcetera.

Note here that we use the generator graphs in two ways. In between prefix, header and text segments as a generator, otherwise as a check (recognizer) on the correctness of these segments. If, e.g., a header segment contained 'stx' then a transition to state 4.4 would occur. Depending on implementation we could either

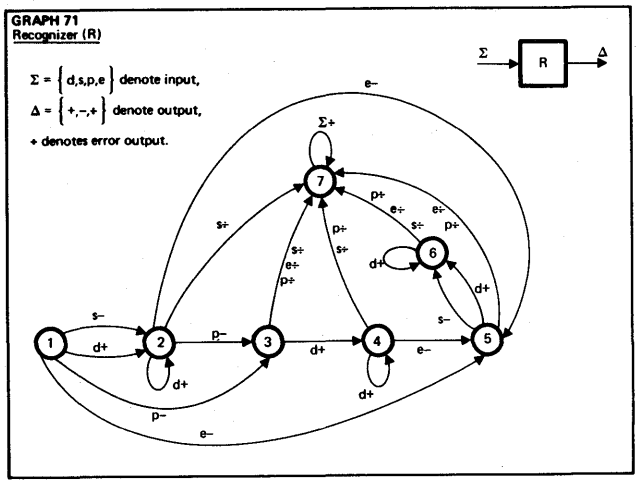


Figure 18—GRAPH 71
ALGOL 60 number recognize graph. Fourth level

Input:

State:	d	s	p	e
i: 1	2+	2-	3-	5-
2	2+	7÷	3-	5-
3	4+	7÷	7÷	7÷
4	4+	7÷	7÷	5-
5	6+	6-	7÷	7÷
6	6+	7÷	7÷	7÷
E: 7	7÷	7÷	7÷	7÷

next state
output
present state
S={1,2,3,4,5,6,7}

Figure 19—TABLE II
Fourth level tabular representation of Graph 71, Figure 18

use this transition to signal error, or the graph, when the header buffer subsequently became exhausted, would signal error because of the repeated attempt to output stx (from state 4.5).

THREE BASIC IMPLEMENTATION SCHEMES

As pointed out in above, the generator definition equals the transducer implementation. In this section we show three such. The first is based on Graphs 70, (Figure 17) and 71 (Figure 18).

The equivalent tabular representation is shown in Table 2, Figure 19 (Table 2 defines ALGOL-60 ⟨numbers⟩'s).

The three schemes are:

1. Hardware sequential machine or automaton using logic gates and flip-flops.
2. Micro program in read-only/writable control store.
3. Software program in main core.

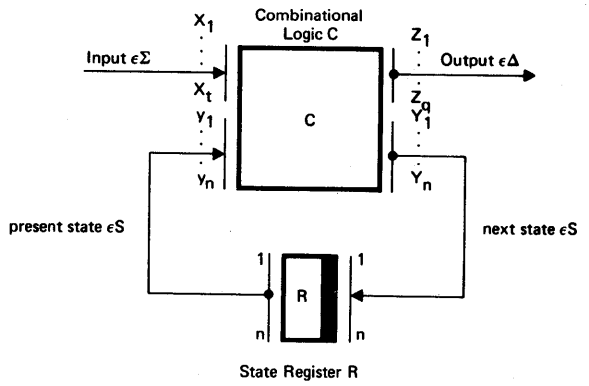


Figure 20—First Niveau sequential machine model

Scheme 1: A Finite State Machine (FSM)

The model of a hardware implemented fsm is given in Figure 20.

C maps $\Sigma \times S$ into $\Delta \times S$. The mapping is given in Table 2. The number of flip-flops needed in R is $n \geq \text{Log}_2$ (the number of states in S), n suitably being the smallest integer larger than or equal to $\text{Log}_2 (\mid S \mid)$. We choose $n=3$; and let $\div \equiv -$, i.e., we do not distinguish between these two symbols. In Table 3, Figure

34

Input: d=00	s=01	p=11	e=10	State:
000				
001	0101	0100	0110	1010	1
011	1001	1110	1110	1110	3
010	0101	1110	0110	1010	2
110	1101	1110	1110	1110	6
111	1110	1110	1110	1110	7
101	1101	1100	1110	1110	5
100	1001	1110	1110	1010	4

Figure 21—TABLE III
Fifth level binary tabular representation of Table II, Figure 19

State:	Input: 00	01	11	10
000	0	0		
001	0	0	1	1
011	0	1	1	1
010	0	1	1	1
110	0	1	1	1
111	1	1	1	1
101	0	0	1	1
100	0	1	1	1

$$\bar{Y}_3 = \bar{X}_1\bar{y}_1\bar{y}_2 \vee \bar{X}_1\bar{X}_2\bar{y}_1 \vee \bar{X}_1\bar{X}_2y_2\bar{y}_3 \vee \bar{X}_1\bar{X}_2\bar{y}_2 \vee \bar{X}_1y_1\bar{y}_2y_3$$

$$= \bar{X}_1\left(\bar{X}_2\,(\bar{y}_1 \vee \bar{y}_2 \vee \bar{y}_3) \vee \bar{y}_2\,(\bar{y}_1 \vee \bar{y}_3)\right)$$

Figure 22—TABLE IV
Binary tabular representation of variable Y_3 in Table III, Figure 21

21, we have coded $s \in S$ as the binary number shown in Table 2.

(This happens to be a 'not so good' internal state coding.) Output '+' is coded as logical '1,' '−' and '÷' both as '0.' Input d='00,' s='01,' p='11' and e='10.'

Note that in Table 3, row 000, state 0 is undefined. We can fill entries in row 000 with whatever $Y1$ $Y2$

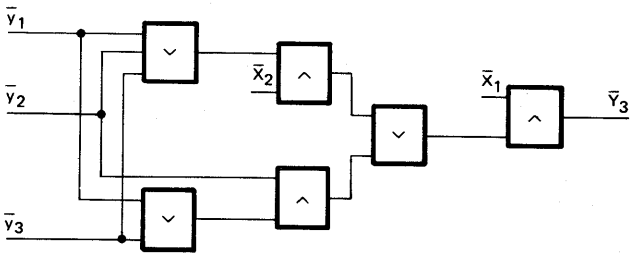


Figure 23—Third Niveau logic primitive implementation of variable Y_3 in Table IV, Figure 22

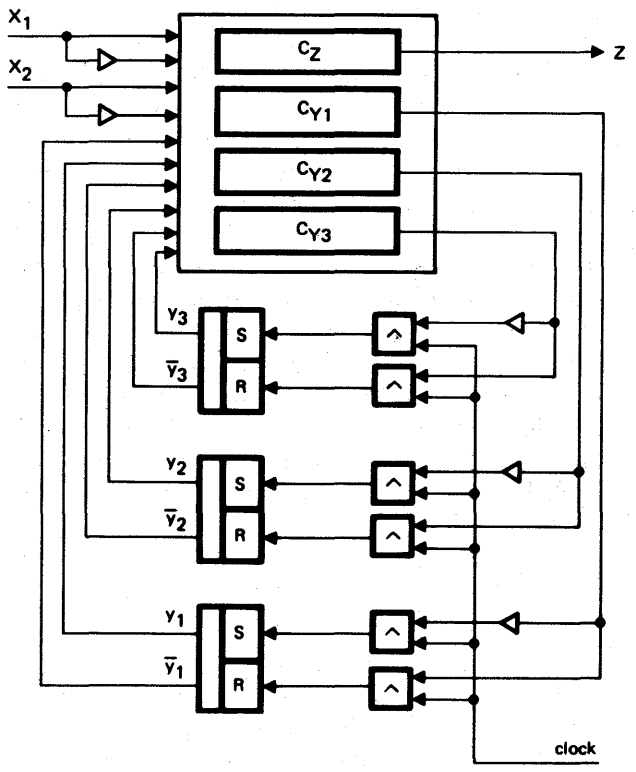


Figure 24—Second Niveau logic block implementation of sequential machine model of Figure 20

$Y3$ Z values we might find useful. From Table 3 we then compute Yi $(i=1,2,3)$ and Z as a Boolean function: $f(x1, x2, y1, y2, y3)$. Take for example $Y3$ or $\bar{Y}3$ shown in Table 4, Figure 22.

The logic for $\bar{Y}3$ (C_{Y3}) is indicated in Figure 23, and a typical implementation from Figure 20 is shown in Figure 24.

Scheme 2: A microprogrammed transmission control unit (μTCU)

The μTCU is to serve many independent lines sharing the same microprogram for their identical line control procedures.

In Figure 25 we show only the principles of the line *input* data flow: An input character is de-Serialized in *SdS*, transferred in parallel to *BF* and when serviceable by the writable control store (*WCS*), the input $(\sigma \in \Sigma)$ and the present state $(s \in S)$ which is kept in *Sreg* are transferred to the *WCS* address decoder.

If the *WCS* is to act only as a recognizer then just Table 2 (or some other appropriate table, e.g., Table 1) is stored in *WCS*. The *WCS* word addressed by (s, σ) has value (s', δ) where s' is the next state and δ is the output ('controls to *DF*') (e.g. $\{+, -, \div\}$), s' goes back to *Sreg* and the *WCS* is ready to serve another line.

If the *WCS* is to manipulate the input (i.e., possibly store σ in a *TCU* line buffer (*lbf*), changing code structure, incrementing *lbf* address, decrementing character count) then (s, σ) initiates a several microword subroutine via the $(s\sigma)$ loop. Branches in this routine are determined by conditions in the *DF*. The last micro instruction contains s' and directs s' to *Sreg* whereafter the *WCS* is again ready to serve any line.

All this is very trivial. What we should note however is that the basic flow of the microprogram, i.e., the proper sequencing of subroutines, is completely derived from (e.g.) Graphs 42-46. In other words if we change the definition of line control procedures then we can mechanically relate such changes to the microprogramming of the *TCU*.

Scheme 3: Software controlled recognition and transduction

Let Table 2 or 3 be stored as a three dimensional array $T(1{:}\,|\,S\,|, 1{:}\,|\,\Sigma\,|, 1{:}2)$ where $|\,X\,|$ denotes the number of elements in the set X. The next state part of the table is stored in $T(*, *, 1)$ whereas the output or action label part is stored in $T(*, *, 2)$. Let INDEX (CHARACTERSET, SYMBOL) map classes of 8 bit

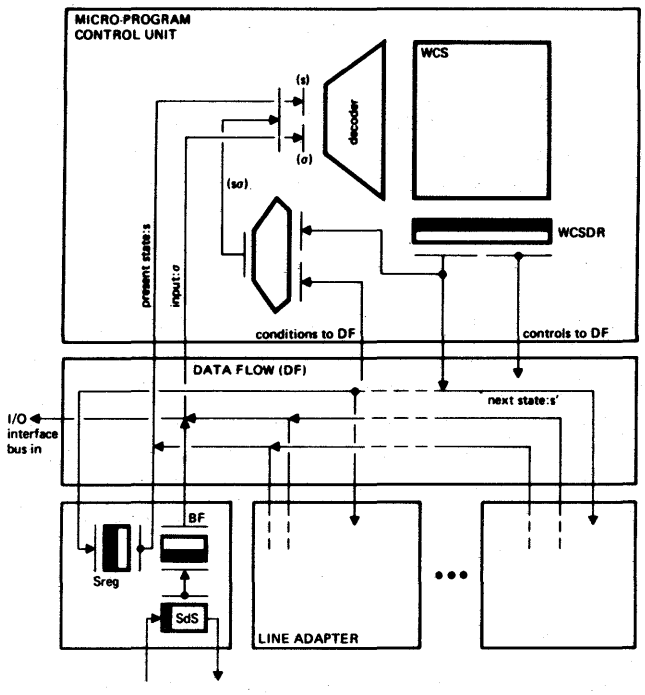


Figure 25—Micro-programmed data communication control unit. Only the input (read) data flow is shown

```
Program 1:
PROGRAM:   PROCEDURE OPTIONS (MAIN);
           DECLARE
                SYMBOL CHARACTER (1),
                CHARACTERSET CHARACTER (60),          /*INITIALIZE PROPERLY*/
                INPUT_CLASS (0:60) FIXED BINARY (7),  /*INITIALIZE PROPERLY*/
                T (1:S, 1:Σ, 1:2) FIXED BINARY (15),  /*INITIALIZE PROPERLY*/
                (INPUT, PRESENTSTATE) FIXED BINARY (15);

                     PRESENTSTATE = 1;
           READ:     GET EDIT (SYMBOL) (A(1));
                     INPUT = INPUT_CLASS (INDEX(CHARACTERSET,SYMBOL));
                     CALL ACTION (T (PRESENTSTATE, INPUT, 2));
                     PRESENTSTATE = T (PRESENTSTATE, INPUT, 1);
                     GO TO READ;

ACTION:    PROCEDURE (L);
           DECLARE SWITCH (N) LABEL INITIAL
                (ACT1, ACT2,...,ACTN);
           GO TO SWITCH (L);
ACT1:      •••
           •
           •
           RETURN;
ACT2:      •••
           •
           •                          actions associated with
           RETURN;                    valid input sequences
    •
    •
ACTM:      •••
           •
           •
           RETURN;
ACTM1:     •••
           •
           •
           GO TO EXIT;
ACTM2:     •••
           •
           •                          actions associated with
           GO TO EXIT;                invalid input sequences
    •
    •
ACTN:      •••
           •
           •
           GO TO EXIT;

END ACTION;  EXIT:
END PROGRAM;
```

Figure 26—Pseudo PL/1 program

symbols into the set $\{1, 2, \ldots, |\Sigma|\}$. The pseudo *PL/1* program on Figure 26 shows the generalized software program control statements into and around which any finite state algorithm can be arranged.[6]

Conclusion

It may be argued that we have shown nothing spectacularly new in the three schemes—the main reason we include this section is to demonstrate the clear-cut relation between definition and implementation—a relation which can be carried out mechanically, i.e., automatically. One is therefore greatly aided when having to decide whether a given scheme is to be implemented using method 1, 2 or 3.

FULL DUPLEX

Before describing how we would go about defining (and implementing) a full duplex scheme, the following brief remark or observation may be appropriate: It appears (at least to this author) that all full duplex schemes hitherto proposed could more appropriately either be called (termed) doubly interleaved half duplex or even just ordinary double simplex. This situation is truly reflected in the scheme next discussed.

First we assume that the transmission is point-to-point or that if we have a multipoint ($N>2$ station-) system. Then we assume that 2 out of N stations have been selected for point-to-point communication, and that the remaining $N-2$ stations 'listen' to both of the two-way simultaneous communication. We shall not here elaborate on the selection problem.

Next we postulate that a definite full duplex data communication line control procedure exists. We finally assume that the text message, text-reply and reply-request scheme shown in Graphs 3, 42-46 is the one used but that we interleave these in a threaded manner when going full duplex.

The constraints put on (or rules defined by) the postulated procedures are therefore:

Rule 1: When one station (A) sends a 'packet' of *type:* 'text-block,' 'reply' or 'reply-request,' then the other station (B) either sends "nothing" ("null"), synchronization (syn) symbols or sends a "packet" of the same type. A termination sequence, eot, has type: text-block. 'Text-block' and 'reply-request' are packets of the same type. This corresponds to the logic of Graph 2, Figure 2.

Rule 2: The exact mutual character-by-character sequence need not necessarily match. That is, any time combination of A or B beginning or ending their "packet" transmission is allowed.

Rules 1 and 2 guarantee that the two way line will never carry packets of different types at any one time.

The example shown in Figure 27 is a particular instance of a set of line transactions. Note specifically the situation after text-blocks m_2 where B first sends ack1 reply to $m_2(A \rightarrow B)$, then reply-request rr in response to an erroneously received ack1$=i$ from A; not before B has received a correct reply (either ackn or nak) can B proceed to text-block m_3. Concurrently: A has received an affirmative answer to $m_2(A \rightarrow B)$, so A proceeds to $m_3(A \rightarrow B)$ before it has cleared the $m_2(B \rightarrow A)$ reply.

In order, therefore, to *synchronize* these two processes properly we employ the following two *indivisible* operations[7] which we define using a pseudo *PL/1* notation:

```
S: SET:   PROCEDURE (semaphore);
          semaphore: = 1;
          END SET;
```

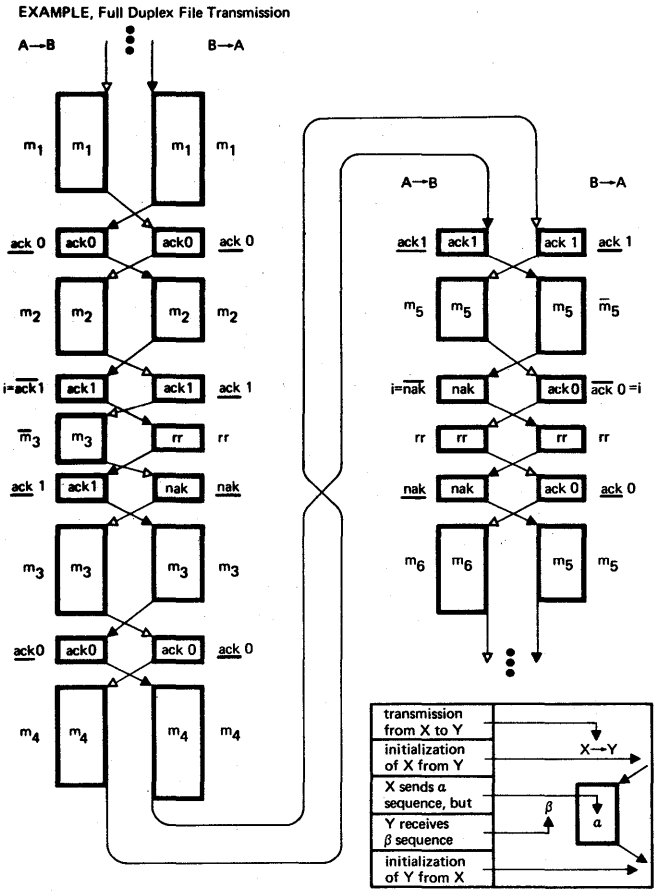


Figure 27—Example. Full duplex file transmission

```
H: HOLD: PROCEDURE (semaphore)
                                    RECURSIVE;
         IF semaphore = 1 THEN semaphore: = 0;
         ELSE HOLD (semaphore);
         END HOLD;
```

On Graph 8, Figure 28, we have shown the totality of generator $(-G)$ and recognizer $(-R)$ graphs in both stations and we have explicitly shown details concerning half of the otherwise symmetric channel traffic, in this case: text blocks $m_{B\to A}$.

1. While *text* block *generator TGb* in B transmits $m_{B\to A}$ the *text* block *recognizer TRa* in A is receiving that same message.
2. When *TRa* has received all of $m_{B\to A}$ then it *releases* the *reply generator* RGa. The release is accomplished through $S(\beta_A)$ and may precede, coincide with or succeed the time at which *TGa* ends transmission, i.e., goes to state $A8$.
3. When the *reply recognizer* RRb has received the reply then it: *either* releases the request-reply generator RRGb because an erroneous reply was received, this release is accomplished through $S(\gamma_B)$, *or* releases *TGb* through $S(\alpha_B)$.
4. If RRGb was initialized then a transition to state B6 will enable RRb to state B9. A completion of the request-reply recognition again releases β_A thus enabling RGa and the two corresponding components RRb and RGa are thus brought to synchronize. We point this out here, but could as well do it for the cases where TGb in state B2 enables RRb to state B9 simultaneously with *TRa* in state A2 ($\cong$B2) releasing β_A thus enabling *RGa* in state A9 ($\cong$B9).
5. In more general terms: The possibility that a generator G of type $x\in\{T, R, RR\}$ in station $u\in\{A, B\}$ enables a recognizer R of type $y\in\{\text{R, RR } or \text{ T, R}\}$ (taken in corresponding order) before the type x recognizer (xRu) has reached home condition (i.e., final- or end-state) should not bother us. yRu will not receive input before xGv has reached its home condition, $v\neq u$.

We thus see that the system of three semaphores $\{\alpha, \beta, \gamma\}$ in each of two communicating stations com-

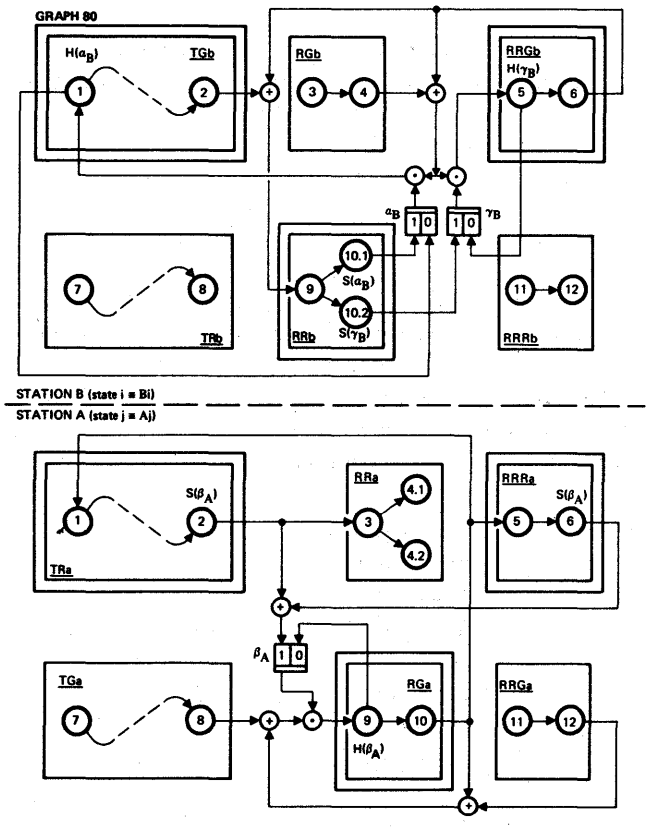


Figure 28—GRAPH 8
Third level full duplex file transmission. Generator and recognizer system

pletely *interlocks* proper sequences. The semaphore values are initially {1, 0, 0}, with respective TR's enabled.

6. One further note: observe that if xGu at time 't' goes to state $j(uj)$ then xRv will at time '$t+\Delta$' go to either the error state or vj. This is true for all state transitions. We might say that the recognizer is always one time step (Δ) behind its associated generator, Δ being the time it takes to serialize, transmit and deserialize a character.

In this specific example we do not require the indivisibility of the H and S semaphore operators. Thus we have not demonstrated the particular virtues of H and S. For the synchronization of more than two lines, however, one is greatly aided by such primitive operators.

Our example demonstrated the notion of (two) cooperating sequential processes. As such it conceptually belongs to the field of parallel computations. This is presently being studied intensely. In fact, it is proper here to refer to the pioneering work by Petri[8] and Holt.[9] Their work essentially embodies the research suggested by Lynch.[10]

In fact, Graph 8 is a somewhat transformed version of a *Petri net* as defined by Holt. Our graph not only shows the algorithmic parallelity but in addition indicates an implementation.

PRODUCTION SYSTEM DEFINITION

As a final (although non-graphical) way of completely and compactly defining a set of line transactions we now show how the language acceptable by an IBM 1070 process communication system[11] IBM 1071 terminal control unit in text mode can be specified through a top-down LL(1)[12] grammar G:

(1) $w \rightarrow$ eoa B
(2) $B \rightarrow$ sel M T eob
$T \rightarrow \mathbf{T}T$ $\quad$ $T \rightarrow$
(3) $\mathbf{T} \rightarrow$ sot S $\quad$ (4) $\mathbf{T} \rightarrow$ eoa A
(5) $S \rightarrow C\ X$ $\quad$ $S \rightarrow$
(6) $C \rightarrow$ char $\quad$ (7) $C \rightarrow$ sel
(8) $A \rightarrow$ char D
(9) $M \rightarrow$ char D $\quad$ $M \rightarrow$
(10) $D \rightarrow$ char H $\quad$ $D \rightarrow$
(11) $H \rightarrow$ char $\quad$ $H \rightarrow$

$$\Sigma = \{\text{eoa, sel, char, sot, eob}\}$$

where char denotes a larger alphabet.

State:	Input: eoa	sel	char	sot	eob
i: 1	2 (1)	3	3	3	3
2	3	4 (2)	3	3	3
f: e: 3	3	3	3	3	3
4	5 (4)	3	7 (9)	6 (3)	3+
5	3	3	7 (8)	3	3
6	5 (4)	6 (5,7)	6 (5,6)	6 (3)	3+
7	5 (4)	3	8 (10)	6 (3)	3+
8	5 (4)	3	9 (11)	6 (3)	3+
9	5 (4)	3	3	6 (3)	3+

Figure 29—TABLE V
Finite state transducer

The regular expression $R(G)$ is:

eoa sel(ϵ U char U char char U char char char)

(sot(char U sel)*U eoa(char U char char

U char char char))*eob

and the mechanically constructed finite state machine top-down transducing $L(G) = |R(G)|$ is shown in Table 5, Figure 29.

CONCLUSION

It is our hope that we will achieve the objective: that of data communication line control procedures being defined through some compact, complete and unambiguous methodology which lends itself to levels of abstraction and to well structured implementations. This results in *better designed communication subsystems*, be it software, hardware, firmware or all. The implications are:

1. *Better control of engineering changes* if any are needed.
2. A much more *generalized service engineering training.*
3. *Systems* which are *easier to diagnose*—visualize that we use all error state transition actions in our diagnostics, and we know *a priori* that we have dealt with all possibilities.
4. Much *better* chances of *implementing* new exotic line control schemes fast and efficient.

5. *Reliable techniques for evaluating* and comparing different implementations software, firmware or hardware, and through the hierarchical structuring, any mix of these.

We have chosen the model of finite state graphs, rather than that of a context-free grammar. We did so because the communications control processes are of a finite state nature. The context-free grammar model is too powerful for our purposes and the finite state graph model is claimed easier to grasp. One is in particular aided by the visuality rather than the formality.

The theory of (finite state) automata and formal languages has come a long way. There are scores of important results waiting to be used practically.

ACKNOWLEDGMENT

The author is indebted to Akira Saito of IBM Japan for pointing out a subtle error in a first draft.

REFERENCES

1 IBM CORP
General information, binary synchronous communication
IBM Systems Reference Library (SRL) Form No A27-3004

2 J EISENBIES
Conventions for data communication
IBM Systems Journal Volume 6 Number 4 pp 267-302 1967

3 ANSI
USA proposed standard data communication control procedures for USASCII
Communications of the ACM Volume 12 Number 3 pp 166-178 March 1969

4 B RANDELL F W ZURCHER
Iterative multilevel modeling, a methodology for computer systems design
IFIP Congress 68 Edinburgh August 5-10 1968
and
B RANDELL
Towards a methodology of computer system design
NATO science committee report: Software engineering pp 204-208 Garmisch Germany October 7-11 1968

5 E W DIJKSTRA
The structure of T.H.E. multiprogramming system
Communications of the ACM Volume 11 Number 5 pp 341-346 May 1968
and
E W DIJKSTRA
Complexity controlled by hierarchical ordering of function and and variability
NATO science committee report: Software engineering pp 181-185 Garmisch Germany October 7-11 1968

6 D BJØRNER
Flowchart machines
IBM research report RJ685 San Jose California April 1970

7 E W DIJKSTRA
Cooperating sequential processes
Programming Languages Ed F Genuys Academic Press Chapter 2 pp 43-112 New York 1968

8 C A PETRI
Kommunikation mit automaten
Schriften des Rheinisch-Westfalischen Institut für Instrumentelle Mathematik Universität Bonn Nummer 2 1962

9 A W HOLT
Information System theory project
Report Applied Data Research Inc Princeton New Jersey September 1968
and
A W HOLT F COMMONER
Events and conditions, An approach to the description and analysis of dynamic systems
Report Applied Data Research Inc Princeton New Jersey 1970

10 W C LYNCH
Commentary on the foregoing note
Communications of the ACM Volume 12 Number 5 pp 261 and 265 May 1969

11 IBM CORP
IBM 1070 process communication system
IBM System Reference Library (SRL) Form Number A26-5780 pp 14-18 1968

12 D BJØRNER
The theory of finite state syntax directed transductions
PhD Thesis Laboratory for Pulse- & Digital Techniques Technical University of Denmark Copenhagen January 1969

A strategy for detecting faults in sequential machines not possessing distinguishing sequences*

by D. E. FARMER

Clarkson College of Technology
Potsdam, New York

INTRODUCTION

The problem treated here is that of detecting faults in digital equipment by applying input sequences at the input terminals and observing output sequences at the output terminals. The checking of digital equipment by input/output tests applied at the terminals is motivated by current and future usage of large-scale integration techniques which make internal test points generally inaccessible for testing purposes. The modeling of digital equipment by finite-state sequential machines and then designing fault-detection tests based on the state table is a general approach. The difficulty is that it results in very long experiments for large state tables, particularly for the case in which the state table does not possess a *distinguishing sequence*, that is, an input sequence for which the response uniquely identifies the initial state. This paper presents a strategy for designing more efficient fault-detection tests for machines not possessing distinguishing sequences.

The fault-detection problem may be viewed as a special case of the machine identification problem. The general machine-identification problem consists of determining a state table for an unknown finite-state machine by applying inputs and observing outputs. The fault-detection problem consists of determining whether the possibly faulty machine is describable by the state table of the fault-free machine. The fault-detection test either verifies that this is the case or that it is not the case. If the test shows that the possibly faulty machine is not describable by the state table of the fault-free machine, it provides no further information concerning its state table.

The machine-identification problem was first treated by Moore.[1] He proposes setting an upper bound, n, on the number of states which the unknown machine may possess. He then forms a composite state table from all distinct state tables of n or fewer states. For this composite state table Moore finds a *homing sequence*, an input sequence for which the output uniquely identifies the final state. Application of the homing sequence to an unknown machine permits the determination of the final state of the composite machine (containing the unknown machine) and consequently of the machine containing that state. The difficulty with this approach is that the number of distinct n-state tables grows very rapidly for increasing n and consequently the test procedures become impractically long.

Further work on distinguishing sequences and homing sequences is contained in Gill[2] and Hibbard.[3] It is shown that every reduced n-state machine possesses a homing sequence and that the shortest such sequence contains at most $(\frac{1}{2})(n)(n-1)$ symbols. It is also shown that a machine need not possess a distinguishing sequence.

The design of test procedures based on knowledge of the state table of the fault-free machine was first treated by Hennie.[4] Hennie's tests are true fault-detection tests rather than machine-identification tests. Hennie treats both the case in which a machine possesses a distinguishing sequence and the case in which it does not. He demonstrates the feasibility of test procedures for both cases and presents the philosophic basis for the tests. He does not claim to present the optimum strategy for test design.

Kime[5] presents an alternate strategy for the distinguishing-sequence case, which retains Hennie's philosophy, but is somewhat more systematic and results in a lower upper bound on the length of tests.

---

* This paper is based on a dissertation submitted by the author in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering at the Polytechnic Institute of Brooklyn. This work was supported in part by the National Science Foundation Grant No. GK-10218.

Kohavi and Lavallee[6] treat the problem of designing machines so that they possess distinguishing sequences and also present a special test organization for machines which possess distinguishing sequences composed of repetitions of a single input symbol. Kohavi and Kohavi[7] present a test organization for the special case in which the machine possesses a variable-length distinguishing sequence. In this case some of the states may be identified by a prefix of a longer distinguishing sequence (required for other states).

The strategy presented in this paper includes the Kime organization and the Kohavi and Kohavi variable-length distinguishing sequence organization as special cases of a more general approach. Primary attention is focused on machines which do not possess distinguishing sequences, but the distinguishing-sequence case is also included.

For machines not possessing distinguishing sequences, an initial state must be identified by observing the set of responses to a set of input sequences, each element of the set being applied to the machine in the same initial state. Such a set of input sequences is called a *characterizing set.* It is possible to use different characterizing sets for the various states, some of lower order than others. The application of this strategy results in shorter test procedures. This is one of the principal features of this work.

Another feature is a general treatment of *locating sequences.* A locating sequence, as used here, is an input sequence for which the observation of a specified response permits the determination of the state of the machine at some specified point in the sequence. Hennie treats a special kind of locating sequence, one which not only locates a state, but also checks the response to a characterizing set for the located state. In this paper Hennie's locating sequence is called a *characterization-verification sequence.* The selective use of these sequences is an important feature of the testing strategy presented here.

DEFINITIONS AND NOTATION

The sequential machines considered in this paper are finite-state, synchronous, deterministic, strongly connected, and completely specified. The machines are of the Mealy model where $X=\{x_1, x_2, \ldots, x_m\}$ is a finite set of input symbols, $S=\{S_1, S_2, \ldots, S_n\}$ is a finite set of internal states, and $Z=\{z_1, z_2, \ldots, z_p\}$ is a finite set of output symbols. There is an output function, $\omega(S_i, x_j)$, which specifies an output symbol for every $S_i \in S, x_j \in X$. There is a next-state function, $\delta(S_i, x_j)$, which specifies a next state for every $S_i \in S$, $x_j \in X$.

Throughout this paper we shall be concerned with two machines: the fault-free machine, M, and the possibly faulty machine, M'. A fault-detection test must establish that machine M' is or is not indistinguishable from machine M, subject to certain assumed restrictions on M'. It is necessary to fix an upper bound for the number of states which M' may possess in order that the fault-detection test be finite. In this paper the upper bound is taken as n, the number of states of the fault-free machine. This is usually justifiable by knowledge of the internal structure of the machine being tested. It is also assumed that faults are logical, that is, M' is describable by a finite-state deterministic state table.

In order to distinguish between the states of the machines M and M', we denote the ith state of M' by S_i'. It is our goal to establish that the behavior of M' for initial state S_i' is indistinguishable from the behavior of M for initial state S_i. For convenience we choose the same subscript for corresponding indistinguishable states of M and M'.

In order to make these concepts precise, we make the following definitions.

Definition 1

A *fault-detection test for machine* M' is the application at the input terminals of M' of a sequence of input symbols such that the observation of the sequence of output symbols (response) produced at the output terminals is sufficient to determine whether or not M' is indistinguishable from M, subject to the assumptions that M' contains no more states than does M and that M' is representable by a finite-state deterministic state table.

Definition 2

A *locating sequence for state* S_i *of machine* M is an input sequence for which the observation of a specified response is sufficient to determine that M is in S_i at some known point in the application of the locating sequence.

The state S_i is *located* by the locating sequence. In accordance with whether S_i is located prior to, during, or after the application, the sequence is an *initial-state, intermediate-state,* or *present-state* locating sequence. An initial-state locating sequence and an intermediate-state locating sequence are also present-state locating sequences, since knowledge of the input sequence and the state table for M provides knowledge of a future state of M after some state is first located.

Throughout this paper the terms present-state or intermediate-state locating sequence are usually applied to sequences which are not also initial-state locating sequences. A reduced n-state sequential machine does not necessarily possess an initial-state locating sequence for any of its states.

Definition 3

A *distinguishing sequence for machine M* is an input sequence which is an initial-state locating sequence for all of the states of M.

A distinguishing sequence can also be defined as an input sequence for which the response of the machine is different for each initial state. This is equivalent to the preceding definition.

Definition 4

A *homing sequence for machine M* is an input sequence which is a present-state locating sequence for all states into which M can be transferred by the homing sequence. Any initial state is permitted.

For an input sequence to be a homing sequence, it must be possible to apply it to the machine in any initial state and have it locate the present state. The response is a function of the *initial* state, but serves to locate the *present* state. We are assured that a present-state locating sequence for some state exists for every reduced n-state machine because a homing sequence exists.

Definition 5

A *characterizing set for state S_i* of machine M is a set of input sequences for which the corresponding set of responses is unique to initial state S_i, M being in S_i prior to the application of each element of the characterizing set.

The characterizing set for S_i *characterizes* S_i.

Definition 6

A *characterizing set for machine M* is a set of input sequences which is a characterizing set for every state of M.

The characterizing set for M *characterizes* M. It is clear that a characterizing set for M can be obtained from the union of characterizing sets for all the states of M. A characterizing set for a state may thus be viewed as a subset of a characterizing set for a machine.

An initial-state locating sequence for state S_i is a characterizing set for S_i of order one. Likewise, a distinguishing sequence for M is a characterizing set for M of order one.

Definitions 2 through 6 apply to machine M, the fault-free machine. These concepts can also be applied to machine M', the possibly faulty machine, provided that we can devise a test procedure for verifying that M' possesses n distinct states which are characterized by the characterizing sets for the n states of M. The following definition provides us with a required tool for this purpose.

Definition 7

A *characterization-verification sequence for state S_i' of machine M'* is an input sequence applied to M' for which the observation of a specified response verifies that M' contains a state S_i' which is characterized by the characterizing set for state S_i of machine M.

It is incidental to the preceding definition that the way of forming characterization-verification sequences, presented in a subsequent section of this paper, results in their being intermediate-state locating sequences for S_i'.

Tables I, II, and III are the state tables for machines

TABLE I—State Table for Machine M_1

Present state	Next state, Output	
	input = 0	input = 1
A	A, 1	C, 0
B	A, 0	C, 0
C	B, 0	D, 0
D	A, 1	D, 0

TABLE II—State Table for Machine M_2

Present state	Next state, Output	
	input = 0	input = 1
A	A, 1	B, 0
B	C, 0	A, 0
C	B, 0	C, 1

TABLE III—State Table for Machine M_3

Present state	Next state, Output	
	input = 0	input = 1
A	A, 0	B, 0
B	A, 0	C, 0
C	A, 0	D, 0
D	A, 1	D, 0

M_1, M_2, and M_3. These machines provide examples of the types of input sequences just defined. The examples are given in Tables IV through IX.

TABLE IV—Example of an Initial-State Locating Sequence

Initial state	Response of M_1 to input sequence 00
A	11
B	01
C	00
D	11

Note: 00 is an initial-state locating sequence for states B and C as demonstrated by the unique responses for B and C.

TABLE V—Example of a Distinguishing Sequence

Initial state	Response of M_2 to input sequence 01
A	11
B	01
C	00

Note: 01 is a distinguishing sequence for M_2 as demonstrated by the unique responses for each state.

TABLE VI—Example of a Present-State Locating Sequence

Initial state	Response of M_1 to input sequence 0	State after application of input sequence 0
A	1	A
B	0	A
C	0	B
D	1	A

Note: 0 is a present-state locating sequence for state A, since a response of 1 verifies that the present state of M_1 is A.

TABLE VII—Example of a Homing Sequence

Initial state	Response of M_1 to input sequence 10	State after application of input sequence 10
A	00	B
B	00	B
C	01	A
D	01	A

Note: 10 is a homing sequence for M_1, since a response of 00 verifies present state B, while a response of 01 verifies present state A. Also, 10 serves this function for all initial states.

TABLE VIII—Example of a Characterizing Set for a State

Initial state	Response of M_3 to input set {0, 10}
A	{0, 00}
B	{0, 00}
C	{0, 01}
D	{1, 01}

Note: The set {0, 10} is a characterizing set for state C, since the response set {0, 01} is unique to initial state C.

TABLE IX—Example of a Characterizing Set for a Machine

Initial state	Response of M_3 to input set {0, 10, 110}
A	{0, 00, 000}
B	{0, 00, 001}
C	{0, 01, 001}
D	{1, 01, 001}

Note: The set {0, 10, 110} is a characterizing set for machine M_3 as demonstrated by the unique response sets for each state.

SUBDIVISION OF FAULT-DETECTION TESTS

At the outset of the test procedure we must place machine M' into some reference condition in order that we may then apply the fault-detection test proper, which is designed for this specific reference condition. This consists of placing M' in some state, S_0', by applying a homing sequence for machine M, followed by a sequence designed to transfer M to S_0 from its state at the conclusion of the homing sequence. This procedure is adaptive in nature since the transfer sequence used depends on the observed response to the homing sequence. Although this procedure treats M' as if it were M, if M' is not in fact transferred to S_0', this will be discovered in the fault-detection test proper and it is then established that M' is faulty.

The fault-detection test is preset and is designed for application to machine M' in state S_0'. The test may be subdivided (at least conceptually) into two parts. Part (1) establishes that M' possesses n distinct states, S_1', S_2', . . . , S_n', which are characterized by the characterizing sets for the n states of machine M. It also establishes that a locating sequence (either initial-state or present-state) exists for some state of M'. Part (2) establishes that the next-state and output functions for M' correspond to the next-state and output functions for M. We call part (1) the characterizing portion and part (2) the transition and output checking portion. In applying the tests the two parts may occur in any order and may in fact overlap one

another. It is convenient for the purpose of this discussion to treat the transition and output checking portion first, remembering that it depends for its validity on the characterizing portion.

TRANSITION AND OUTPUT CHECKING TESTS

The following notation is introduced for the description of this portion of the fault-detection test.

Let S_i, $1 \leqq i \leqq n$, denote the ith state of M.
x_j, $1 \leqq j \leqq m$, denote the jth input symbol.
$S_{ij} = \delta(S_i, x_j)$, denote the x_j successor state of S_i.
k_i denote the order of the characterizing set for S_i.
$\{X_{i1}, X_{i2}, \ldots, X_{il}, \ldots, X_{ik_i}\}$, $1 \leqq l \leqq k_i$, denote the characterizing set for S_i.
k_{ij} denote the order of the characterizing set for S_{ij}.
$\{X_{ij1}, X_{ij2}, \ldots, X_{ijl}, \ldots, X_{ijk_{ij}}\}$, $1 \leqq l \leqq k_{ij}$, denote the characterizing set for S_{ij}.

Let Q_{il} be the state in which M is left after application of X_{il}, M being initially in S_i.
Q_{ijl} be the state in which M is left after application of X_{ijl}, M being initially in S_{ij}.
$T(S_a, S_b)$ denote an input sequence which transfers M from state S_a to state S_b.
S_0 denote the reference state for the test.
X_0 denote a sequence which takes M from S_0 to a state Q_0 and which is also a present-state locating sequence for Q_0.
X_0 may also be an initial-state locating sequence for S_0 and is selected as such if machine M possesses an initial-state locating sequence for S_0.

The following operation symbols are defined.

1. Sequence inclusion is denoted by $+$. X_1+X_2 means any sequence which includes both X_1 and X_2 as subsequences. It is convenient to call this "summation."
2. Multiple inclusion is denoted by $\sum$. $\sum_{i=1}^{n} X_i$ means any sequence which includes all n X_i's as subsequences. These sequences are not necessarily disjoint, but may overlap.
3. XY denotes sequence X followed by sequence Y.

It is claimed that the following input sequence, *TOCP*, constitutes a transition and output checking portion of a fault-detection test.

$$TOCP = \sum_{i=1}^{n} \sum_{l=1}^{k_i} X_0 T(Q_0, S_i) X_{il} T(Q_{il}, S_0) + \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{l=1}^{k_{ij}} X_0 T(Q_0, S_i) x_j X_{ijl} T(Q_{ijl}, S_0) \quad (1)$$

That the inclusion of these subsequences is a sufficient condition for the sequence, *TCOP*, to constitute a transition and output checking portion of a fault-detection test is seen as follows. The first summation verifies a transfer sequence from state Q_0 to every other state. The second summation, together with this established transfer sequence, verifies the x_j successors and x_j outputs for every state. In fact, the second summation includes the first. This is due to the assumed strongly connected property of the machine. The fact that every state is reached through some transition means that a transition check for one state also verifies a transfer from Q_0 to another state. Therefore, the second summation alone constitutes the transition and output checking portion.

THE TRANSITION AND OUTPUT CHECKING TREE

The various subsequences of the second summation of Equation (1) may overlap. The greatest overlap results in the most efficient fault-detection test. We now introduce the transition-checking tree as an aid to achieving maximum overlap. This tree is a form of state transition diagram with nodes representing states and directed branches representing transitions for specified input sequences. The tree consists of four portions as follows.

1. The *locating-sequence trunk* consists of a single branch originating at state S_0 and directed to state Q_0. This branch is labeled with the input sequence X_0.
2. The *transition-covering portion* consists of a state transition diagram originating from state Q_0 and developed tree-like through successive levels until all of the transitions of M have been covered. There are $(m)(n)$ branches in this part, each labeled with a single input symbol.
3. The *characterization-completion portion* carries the tree through an additional level. From each terminal node of the transition-covering portion

is directed a branch or branches labeled with the input sequence or sequences needed to ensure that *each node of the transition-covering portion* be succeeded by tree paths labeled with sequences covering every element of the characterizing set for the state associated with the node.

4. The *closing portion* of the tree carries the tree through an additional level so that each tree path ends at state S_0. There is a branch from each terminal node of the characterization-completion portion labeled with the necessary transfer sequence to return the machine to S_0. In the event that a terminal node of the characterization-completion portion corresponds to state S_0, a closing branch labeled with λ, the zero-length sequence, is included.

Figure 1 is the transition and output checking tree for machine M_4, the state table for which is given in Table X. Table XI gives the responses of M_4 to input sequences 0, 10, 010 for each initial state.

Specializing the general notation previously given for machine M_4, we obtain:

$$S_1=A \qquad \{X_{11}\}=\{010\}$$
$$S_2=B \qquad \{X_{21}\}=\{010\}$$
$$S_3=C \qquad \{X_{31}, X_{32}\}=\{0, 10\}$$
$$S_4=D \qquad \{X_{41}, X_{42}\}=\{0, 10\}$$

We select $S_0=A$

$X_0=010$, which is an initial state locating sequence for state A, having response 000 associated with it.

$Q_0=A$

TABLE X—State Table for Machine M_4

Present state	Next state, Output	
	input = 0	input = 1
A	B, 0	D, 0
B	A, 0	B, 0
C	D, 1	A, 0
D	D, 1	C, 0

TABLE XI—Response of M_4 to Inputs 0, 10, 010

Initial state	Response to input sequence		
	0	10	010
A	0	01	000
B	0	00	001
C	1	00	101
D	1	01	101

The transition and output checking portion of the fault-detection test is constructed directly from this tree. This is done by tracing all of the paths through the tree from the origin (S_0) back to the common destination (also S_0). The input sequences associated with these paths are then concatenated in any order to form the transition and output checking portion. That this is sufficient is seen from the fact that these paths correspond to the second summation of Equation (1), covering every transition (input/state combination), with each transition being followed by each element of the characterizing set for the successor state associated with the transition. The following is such a test for machine M_4, taken from the tree of Figure 1, in left to right path order:

(01000010) (010001011) (0100101011) (01010011)

(01010101) (010110011) (010110101) (010111010)

If the possibly faulty machine responds to the above sequence as would the fault-free machine, then one portion, in this case the longest portion, of the fault-detection test has been passed.

In the following section we present a 23-symbol characterizing portion for this machine. The above

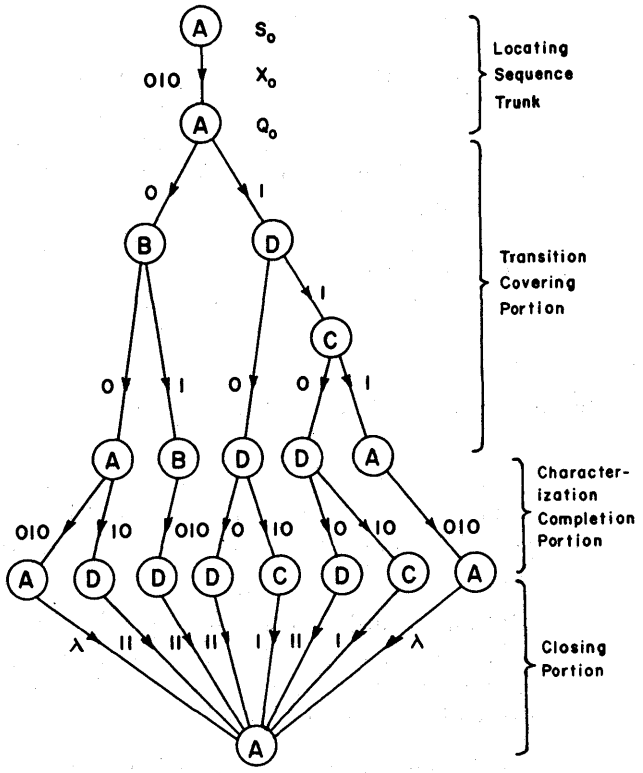


Figure 1—Transition and output checking tree for M_4

transition and output checking portion contains 70 symbols, making a total of 93 symbols for the length of a complete fault-detection test for M_4'. Hennie designs a fault-detection test of length 152 symbols for this same machine, although he explicitly states that this is undoubtedly not optimal. The chief saving over Hennie's test is due to the following:

1. A tree-organized overlapping procedure is utilized together with characterizing sets of order one for states A and B. Hennie uses the characterizing set $\{0, 10\}$ for all the states.
2. The sequence 010 is used for the locating sequence trunk of the transition and output checking portion. This contrasts with Hennie's corresponding use of a characterization-verification sequence for state D containing six symbols.

The preceding example was for a machine which possessed an initial-state locating sequence for some state. Such a class of machines may be thought of as intermediate between the class of machines possessing distinguishing sequences and the most general class, possessing no initial-state locating sequence. For the most general case the transition and output checking tree is still applicable, but a present-state locating sequence must be used for the locating sequence trunk. The characterizing portion of the fault-detection test is much longer for such machines. In this case S_0 is replaced by the set of initial states for which X_0 takes the machine to Q_0. The closing portion of the tree may return the machine to any member of this set.

THE CHARACTERIZING PORTION OF THE FAULT-DETECTION TEST

The characterizing portion of a fault-detection test is composed of characterization-verification sequences for each state linked together by appropriate transfer sequences. We now consider the organization of characterization-verification sequences. The strategy used is based on the behavior of the finite-state machines for repetitive input patterns. We require the following lemma for proving the desired result.

Lemma 1

Let x^n denote n consecutive repetitions of the input sequence, X. If X^n is applied to the n-state sequential machine, M', the state in which the machine remains is the same as a state in which it resided prior to at least one of the applications of X.

Proof

A specialized form of transition graph for a machine is obtained by letting directed branches between nodes represent the transitions between states which occur *only* for the input sequence, X. The application of X^n to the machine corresponds to tracing an n-branch path on this graph. Since M' is assumed to contain no more than n states, this path necessarily contains a loop. The lemma follows.

We now proceed to the organization of a characterization-verification sequence. The following notation is required:

Let $\{X_1, X_2, \ldots, X_l, \ldots, X_k\}$ denote a characterizing set for machine M. $(1 \leqq l \leqq k)$

$Q_{i1}, Q_{i2}, \ldots, Q_{il}, \ldots, Q_{ik}$ denote the set of states to which M is transferred by the application of X_{il}, $1 \leqq l \leqq k$, M being initially in S_i.

$T(S_a, S_b)$ denote an input sequence which transfers M from S_a to S_b.

Define $\bar{X}_{il} = X_l T(Q_{il}, S_i)$

Then, $\{\bar{X}_{i1}, \bar{X}_{i2}, \ldots, \bar{X}_{il}, \ldots, \bar{X}_{ik}\}$ is another characterizing set for M (and for S_i). This is called the *modified characterizing* set for S_i, each element being modified so as to return M to S_i when applied to M initially in S_i.

Let $\{\bar{Z}_{i1}, \bar{Z}_{i2}, \ldots, \bar{Z}_{il}, \ldots, \bar{Z}_{ik}\}$ denote the set of responses of M initially in S_i to the elements $\bar{X}_{il}$.

Let
$$
\begin{aligned}
Y_{i1} &= \bar{X}_{i1} \\
Y_{i2} &= Y_{i1}{}^n \bar{X}_{i2} \\
Y_{i3} &= Y_{i2}{}^n Y_{i1}{}^n \bar{X}_{i3} \\
&\cdots\cdots\cdots\cdots \\
Y_{il} &= Y_{i}{}^n{}_{(l-1)} Y_{i}{}^n{}_{(l-2)} \cdots Y_{i1}{}^n \bar{X}_{il} \\
&\cdots\cdots\cdots\cdots\cdots\cdots \\
Y_{i(k-1)} &= Y_{i}{}^n{}_{(k-2)} Y_{i}{}^n{}_{(k-3)} \cdots Y_{i1}{}^n \bar{X}_{i(k-1)} \\
Y_{ik} &= Y_{i}{}^n{}_{(k-1)} Y_{i}{}^n{}_{(k-2)} \cdots Y_{i1}{}^n \bar{X}_{ik} \qquad (2)
\end{aligned}
$$

Let W_{ik} denote the response of M initially in S_i to Y_{ik}.

Theorem 1

Y_{ik}, defined by Equation 2, is a characterization-verification sequence for state S_i', that is, its applica-

tion to M' and observation of response W_{ik} verifies that M' has a state characterized by the characterizing set for state S_i of machine M.

Proof

Note that $Y_{i(k-1)}$ concludes with $\bar{X}_{i(k-1)}$. The sequence $\bar{X}_{i(k-1)}Y_i{}^n{}_{(k-2)}Y_i{}^n{}_{(k-3)}\cdots Y_{i1}{}^n$ is repeated n times just prior to $\bar{X}_{ik}$ in the sequence Y_{ik}. By Lemma 1 the state of M' just prior to the application of $\bar{X}_{ik}$ must be the same as some state just prior to an application of $\bar{X}_{i(k-1)}Y_i{}^n{}_{(k-2)}Y_i{}^n{}_{(k-3)}\cdots Y_{i1}{}^n$. If the correct output sequence, W_{ik}, is observed, this state is one for which M' responds to $\bar{X}_{i(k-1)}$ with $\bar{Z}_{i(k-1)}$ *and* to $\bar{X}_{ik}$ with $\bar{Z}_{ik}$. Similar reasoning based on repetitive patterns shows that the state of M' just prior to the application of $\bar{X}_{ik}$ is a state for which M' responds to the input set, $\{\bar{X}_{i1}, \bar{X}_{i2}, \ldots, \bar{X}_{ik}\}$, with the response set,

$$\{\bar{Z}_{i1}, \bar{Z}_{i2}, \ldots, \bar{Z}_{ik}\}.$$

A characterization-verification sequence for state B' of the four-state machine, M_3' (Table III), is now constructed.

A characterizing set for M_3, $\{X_1, X_2, X_3\}$, is the set $\{0, 10, 110\}$ as seen from Table IX.

The modified characterizing set for state B is $\{01, 101, 1101\} = \{\bar{X}_{B1}, \bar{X}_{B2}, \bar{X}_{B3}\}$.

The corresponding response set is $\{00, 000, 0010\} = \{\bar{Z}_{B1}, \bar{Z}_{B2}, \bar{Z}_{B3}\}$.

Therefore $Y_{B1} = \bar{X}_{B1} = 01$

$$Y_{B2} = Y^n{}_{B1}\bar{X}_{B2} = (01)^4101$$

$$Y_{B3} = Y^n{}_{B2}Y^n{}_{B3}\bar{X}_{B3} = [(01)^4101]^4(01)^41101$$

$$W_{B3} = [(00)^4000]^4(00)^40010$$

There exist more efficient ways of designing characterization-verification sequences, taking advantage of the existence of a number of distinct responses to some elements of the characterizing set in order to reduce the number of repetitions. The detailed treatment is omitted here.

Frequently there are informal ways of designing the characterizing portion of a fault-detection test. An example is provided by the 23-symbol characterizing portion for machine M_4 which has previously been mentioned. The following is a sufficient sequence for characterizing the states of M_4' and verifying that 010 is an initial-state locating sequence for state A'.

Input	01	01	0	0^4	10	1	$(01)^4$	10	0	
State of M':	A'	B'	D'	D'	D'	D'	C'	C'	C'	A'
Output:	00	00	1	1^4	01	0	$(10)^4$	00	0	

The first five symbols show that there are at least three states, because there are at least three distinct responses to input sequence 010. State A' is located by response 000 to input 010 and B' is located by response 001 to input 010. The fifth symbol shows that there is at least one state for which the response to input 010 begins with 1. States C' and D' are then characterized by means of the characterization-verification sequences, 0^410 and $(01)^410$, respectively, and it is established that the response set to input set $\{0, 10\}$ is $\{1, 00\}$ for C' and $\{1, 01\}$ for D'.

VERIFICATION OF LOCATING SEQUENCES

An initial-state locating sequence is readily verified by selecting it as one of the elements of the characterizing set. The characterization-verification sequences designed as indicated here then verify that it is in fact an initial-state locating sequence. A present-state locating sequence can also be verified by employing the characterization-verification sequences as intermediate-state locating sequences for the basis of a partial transition and output checking tree, similar to the tree described in this paper. The method is straightforward, but the explanation is lengthy and cannot be detailed here.

CONCLUSIONS

The strategy presented in this paper results in fault-detection tests which are more efficient than those described in previous work. The length is shortened by judicious choice of characterizing sets, advantageous use of overlapping sequences, and by use of the shortest locating sequence for establishing a reference.

Experience has shown that the transition and output checking portion of a fault-detection test is generally the longest part. The saving in length achieved in this part by using the shortest locating sequence at leats equals the difference in length between the shortest locating sequence and the shortest characterization-verification sequence, multiplied by the number of

transitions. The saving achieved by overlapping and judicious selection of characterizing sets is usually even more significant, since this results in tracing fewer paths through the tree and therefore reduces the total number of sequences required for the test.

REFERENCES

1 E F MOORE
Gedanken-experiments on sequential machines
Automata Studies pp 129-153
Princeton University Press Princeton New Jersey 1956

2 A GILL
Introduction to the theory of finite-state machines
McGraw-Hill Book Company New York 1962

3 T N HIBBARD
Least upper bounds on minimal terminal state experiments for two classes of sequential machines
J Assoc Comp Mach Vol 8 pp 601-612 October 1961

4 F C HENNIE
Fault detecting experiments for sequential circuits
Proc 5th Annual Symposium on Switching Circuit Theory and Logical Design pp 95-110 Princeton New Jersey November 1964

5 C R KIME
An organization for checking experiments on sequential circuits
IEEE Transactions on Electronic Computers (Short notes) Vol EC-15 pp 113-115 February 1966

6 Z KOHAVI P LAVALLEE
Design of sequential machines with fault-detection capabilities
IEEE Transactions on Electronic Computers Vol EC-16 pp 473-484 August 1967

7 I KOHAVI Z KOHAVI
Variable-length distinguishing sequences and their application to the design of fault-detection experiments
IEEE Transactions on Computers (Short notes) Vol C-17 pp 792-795 August 1968

Coding/decoding for data compression and error control on data links using digital computers

by H. M. GATES

Braddock, Dunn and McDonald, Inc.
Albuquerque, New Mexico

and

R. B. BLIZARD

Martin Marietta Corporation
Denver, Colorado

INTRODUCTION

Data compression and error control have, over the years, been treated as two separate disciplines. Data compression can substantially reduce the loading of communication channels and error control using coding methodology, can reduce the amount of errors in the messages being transmitted, or allow the system to operate with less power for a comparable uncoded information rate. This paper demonstrates that both functions can be combined into one operation by applying sequential decoding developed for error control to data compression. Because the same general method can be used to solve both problems, data compression and error control can be united in a single system and held accountable for the required theorems in information theory.

The principal incentive for the use of sequential decoding for both compression and error control is that it permits the use of the source redundancy with extremely simple encoding equipment. Higher speed computing systems and larger available memories make it more feasible to use various redundancy schemes. In photographs, for example, line-to-line redundancy can successfully be used.

The proposed process of combining data compression and error control in a sequential decoder is reversible because the original data is recovered intact with no compromising or loss of information and resolution. This is attributed to the sequential decoding process itself. Uncertainty about data modification and missing data does not exist. If the capacity of the channel is exceeded because of increased noise or information activity beyond the design of the system, the data output from the decoding process stops and delivers no further information until the channel improves. The transmitted data delivered to the user is thus creditable.

The combined process does not require two separate systems; one for compression and one for error control. Data compression designers rely heavily on error free channels and error control factions assume purely random information into their system and data link. Naturally, neither is completely true.

In simulating data channels, error control designers expend a great amount of effort in generating a pure random number sequence to test their system. Yet, the data compression specialist expends his time trying to identify patterns in what is sometimes the most obscure material. There should be little doubt that the two processes are very closely related.

Background

The means for decoding convolutional codes first became practical when R. M. Fano in 1963 introduced his now famous sequential decoding algorithm.[1,2] I. M. Jacobs[3] recently discussed sequential decoding as it applies to retrieving data from a space probe. He points out that this is a particularly good application because fading is relatively unimportant, channel noise is nearly Gaussian, and the information rate is limited by the available signal strength rather than by the bandwidth. Significant developments in the implementation of sequential decoders have occurred in the Pioneer IX space program;[4] at the MIT Lincoln Laboratory;[5] and by the Codex Corporation;[6] and new algorithms for decoding convolutional codes are being disclosed annually.[7]

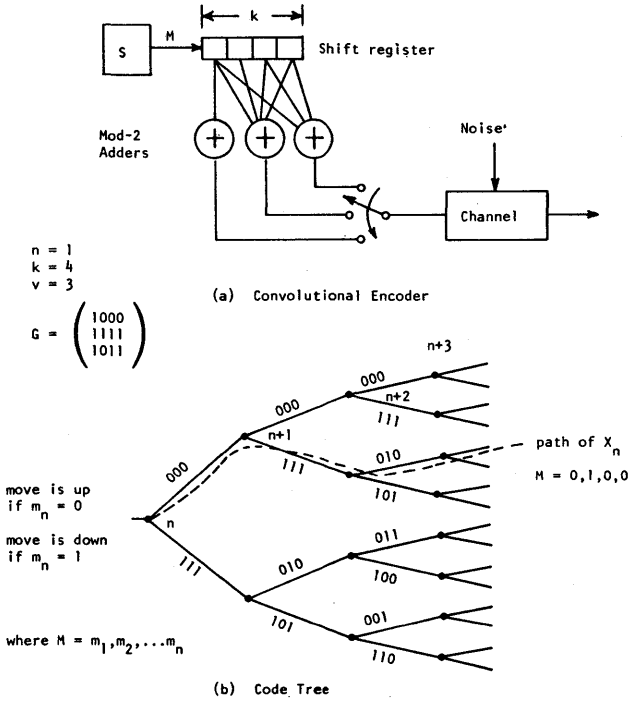


Figure 1—Binary convolutional encoder and its code tree for first 4 moves

The possibility of applying error control coding to data compression was first pointed out by Weiss[8] who showed that block codes work fairly well for compressing data streams consisting of a few 1's scattered at random among many 0's. However, this type of source can be very efficiently and easily encoded with a run-length encoder. More interesting sources are those in which the redundancy is of a more complicated type and is not easily utilized by a block decoder. Sequential decoding is ideally suited to this type of problem.

With block codes such as the orthogonal and BCH codes, the entire block is ordinarily decoded as a unit by computing correlations or by solving certain algebraic equations. Sequential decoding starts at the beginning of a message block and works toward the end by a trial-and-error process. A running account is kept of the reasonableness of the decoded message on the basis of received signal. If a wrong decision is made because of noise, the subsequent decoded message soon becomes unreasonable, and the decoder then searches back over its earlier choices until the correct message is found. It is relatively simple, in concept at least, to use what is known about the message probabilities to improve the likelihood criterion. This allows more data to be transmitted over a given link and accomplishes data compression.

Convolutional Codes

Figure 1a shows a convolutional encoder with a binary message source, M, assumed for the time being to be random information, and an error source, N, representing channel noise. Messages from the source S may be shifted in one or more bits at a time denoted as n. After each shift, v mod-2 adders (in this case three mod-2 adders) are sampled or commutated and the resultant bits passed into the transmitter equipment of the data channel. If four bits are shifted in before the encoder output is commutated, a convolutional data compression process occurs at rate $\frac{4}{3}$. The decoder must then use the *a posteriori* probability of the source S in defining the decoding parameters of the Fano algorithm. Normally, rates of $\frac{1}{2}$, $\frac{1}{3}$, and even $\frac{1}{4}$ are used in error control. That is, for each bit shift in, from two to four mod-2 adders are sampled. The number of mod-2 adders normally is fixed or hardwired for a particular mission or channel requirement. The connections from these mod-2 adders to the shift register, G, represents the codes themselves and a great deal has been written about this subject particularly if the shift register length, called the constraint length, k, is below 12 to 15 bits. Random selection of these codes for large constraint lengths produces adequate, if not completely satisfactory, results.[9] Two rates of 2/1 and 6/4 are demonstrated here. Constraint lengths of 60 bits are used as well.

The basic channel

The channel is important in both error control and compression since it represents the sole source of noise and thus errors into the system, that is, receiver front-end noise, electronic noise in the lines, and so forth, are combined into one noise error source. By definition then, the channel referred to here contains the system modulator, amplifiers, transmitters, antennas or cables, repeaters, receivers, and demodulators. The noise of this channel is assumed to be Gaussian or white noise. The channel is assumed to be stationary in the sense that its properties do not change with time. And the channel is assumed to be memoryless. The channel is described as binary antipodal in which the transmitted waveforms in the absence of noise are a sequence of 180 degree, phase-modulated waveforms, or binary data represented by plus and minus voltages for ones and zeros respectively.

The restrictions imposed on the channel are to this point, fairly realistic, depending of course on the method of transmission. In passing, if the received

binary sequence is quantized into say four or eight levels for each bit received, the compression and error control performance can be increased. This is a well known fact in information theory. This quantization does not change the basic operation of this system or any of the convolutional decoders that this author is aware of.

The decoder

Convolutional codes generated by the encoder shown in Figure 1a can be decoded in one of several ways. The oldest practical method is the use of the Fano Algorithm which is well described in several references.[2] The Fano Algorithm sequentially decodes the data stream as it is received and thus the reference to sequential decoding. Actually, there are several new algorithms for decoding convolutional codes which are extremely clever and add promise to the combination of error control and data compression.

Without going into great detail, a sequential decoding process will be described next. Any code produced by the convolutional encoder may be represented by a binary tree. The encoder input $M=M_1, M_2, \ldots$, corresponds to an infinite path through the tree, and labels on the branches indicate the encoder output. Figure 1b demonstrates this point for a given convolutional encoder. Since more bits are being generated than received in the encoder, i.e., $n<v$, redundancy is introduced to give the code error detecting and correcting capabilities. The received infinite sequence for the channel must similarly be compared with a receiver encoder replica to find the path which maximizes the conditional probability of the received sequence versus the absolute values of the code tree. Each move from one node to another represents a bit shift of information. Sequential decoding is then based on searching the most probable branches of a code tree. Whenever this path becomes too unlikely, a search is initiated for a better path. The sequential decoder is able to determine the correct path while examining only a fraction of the total set of possible paths. The number of searches depends on the errors introduced by the channel in the case of error control. If the channel is very noisy, the number of branches which must be searched before finding the most probable path becomes super exponential. The decoder is limited by the operating speed of the computer and the decoder input rate. The means of scoring which branch or path is best is referred to as the branch metric which is based on the conditional probabilities of the received versus the transmitted signal.

THE COMPUTATIONAL LIMIT

This section contains a discussion of the limits of performance that can be expected for data compression with sequential decoding.

If the convolutional code is fairly long, a sequential decoder has a very small probability of making an undetected error. Instead, when the signal-to-noise ratio gets too high, it runs out of computational capacity and gives up until a new block is started by filling the encoder shift register with zeros. The amount of computation required for each node of the decoding tree is measured by the average number of branches that must be tried before the decoder can advance to the next node. It is easily seen that, as the signal gets noisier (or, if compression is used, as the data become more active), the decoder will make the wrong choice more often, and more trials must be made before the right path is found.

At a certain point, called the computational limit, the average number of trials per node becomes infinite, and the decoder is bound to break down at least occasionally. Work can progress beyond this limit if some losses in the message can be accepted—e.g., when it is possible to obtain a repetition of the parts that were lost. In general, however, the computational limit is a good criterion for the capacity of a channel using sequential decoding. Operation is very good to within a few percent of the limit, and is poor when the limit is exceeded.

The computational limit is also equal to the union bound on the exponential error parameter for block coding.[10] Consider a code block that contains N successive transmitted signals. These may be biphase pulses, or they may be the tones of a frequency-shift-keying transmitter or any other discrete modulation scheme. This code block will be used to convey the information contained in N' successive message symbols. These may be the gray levels of N' picture elements, or perhaps N' letters of English text. It will be shown that there are codes (i.e., rules for mapping the N' symbols into the N signal elements) for which the probability of one or more errors in the block P_e is bounded by

$$P_e \leq 2^{-NE_0(\rho,Q)+N'D(\rho)} \tag{1}$$

where $E(\rho, Q)$ is a function of the signal-to-noise ratio in the channel and $D(\rho)$ depends on the message statistics. It is convenient to define $R'=N'/N$ as the decoding rate. Equation (1) can be rewritten as

$$P_e \leq 2^{-N\{E_0(\rho,Q)+R'D(\rho)\}} \tag{2}$$

to show that for a fixed R', the probability of error decreases exponentially with increasing N so long as $R'D(\rho)<E_0(\rho, Q)$. This defines the limiting rate $R_c=$

$E_0(\rho, Q)/D(\rho)$ that is also the computational limit for sequential decoders.

The proof of Equations (1) and (2) can be shown using Gallager's joint source and channel coding theorem.[9] Gallager leaves the proof of a maximum *a posteriori* probability decoder required for this system to the reader in problem 5.16 which I should like to do as well, although a complete proof does exist on request.[11] The results of Gallager's joint source and channel coding theorem for a discrete memoryless channel with equiprobable symbols is

$$P_e \leq 2^{-N\{E_0(\rho,Q)+\rho R\}}$$

as opposed to the extended results of Equations (1) and (2) above. The units of $E_0(\rho, Q)$ are in bits/channel symbol. The equivalence between R and $R'D(\rho)$ exists so their units are the same. The ratio R' has units of data symbols/channel symbol and $D(\rho)$ in bits/data symbol. This agrees with ρR so ρ is not dimensional and R is in bits/channel symbol. The term ρR may in this case be interpreted as entropy since the binary source digits are independent and equiprobable. Although $R'D(\rho)$ is not, in itself, entropy, it serves as a means of measuring system performance given the ideal case of ρR.

Because $E_0(\rho, Q)$ is determined by the communication link itself, exclusive of encoders and decoders, D is the quantity that determines how much a given message can be compressed. If $\rho=1$

$$D = \log_2[\sum_i (w_i)^{1/2}]^2 \qquad (3)$$

where the $\{w_i\}$ are the probabilities, based on all knowledge available to the decoder, of the possible message symbols. For instance, if the message is English text, each w_i would be the probability that the next symbol is a particular letter or punctuation mark. The decoder could be programmed to take into account the part of the message already decoded, word frequencies, and the idiosyncrasies of the particular writer.

D is the number of bits that would have to be transmitted on an errorless channel to convey the information in each message symbol. In the following sections, D will be evaluated for various types of source statistics.

Equiprobable symbols

Evaluating D in Equation (3) is particularly simple when all the symbol probabilities are the same. For the binary case, $w_0=w_1=\frac{1}{2}$ and $D=1$, indicating that the units of D are bits.

With m equiprobable symbols, $w_i=1/m$ for $i=1, 2, 3, \ldots m$, and $D=\log_2 m$. This is exactly equal to the entropy H for this case and indicates that the computational limit is the same as the Shannon bound for discrete noiseless systems. Of course, this is not a very interesting kind of source for data compression, but if $m=3$, for instance, the message could be coded into about 1.6 bits/symbol instead of going to 2 bits as would be required for a strictly binary system. Blizard, et al.,[11] have shown interesting values for D based on some common distributions several of which are given in this paper.

Binary source with unequal probabilities

A binary source emits only two different symbols. These may be called one and zero. If the probability of one symbol is greater than that of the other, the data stream can be compressed.

If the probability of one symbol is p, and that of the other $1-p$, Equation (3) becomes

$$D = \log_2[(p)^{1/2}+(1-p)^{1/2}]^2$$
$$= \log_2\{1+2[p(1-p)]^{1/2}\}.$$

The corresponding expression for the entropy is

$$-H = p\log_2 p+(1-p)\log_2(1-p).$$

Figure 2 shows the amount of compression, in dB, that can be accomplished with sequential decoding and also the theoretical limit based on the entropy H. It is seen that sequential decoding will give slightly more than half the available improvement measured in dB.

A line drawing is an example of a binary source. In this case, there is a large amount of redundancy in the two-dimensional picture that is available by extrapo-

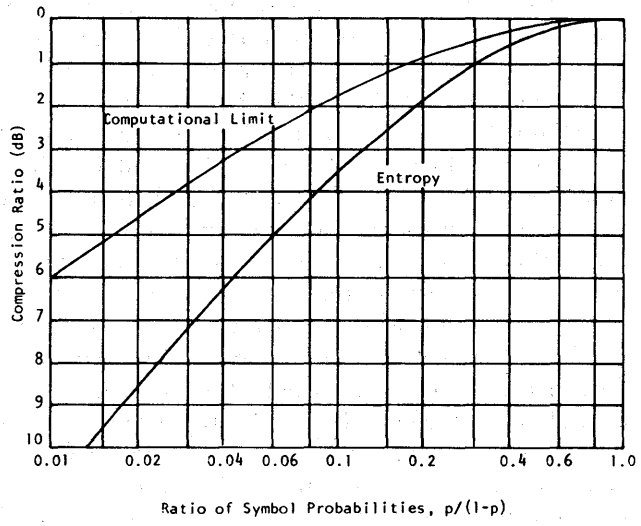


Figure 2—Compression limits for a binary source

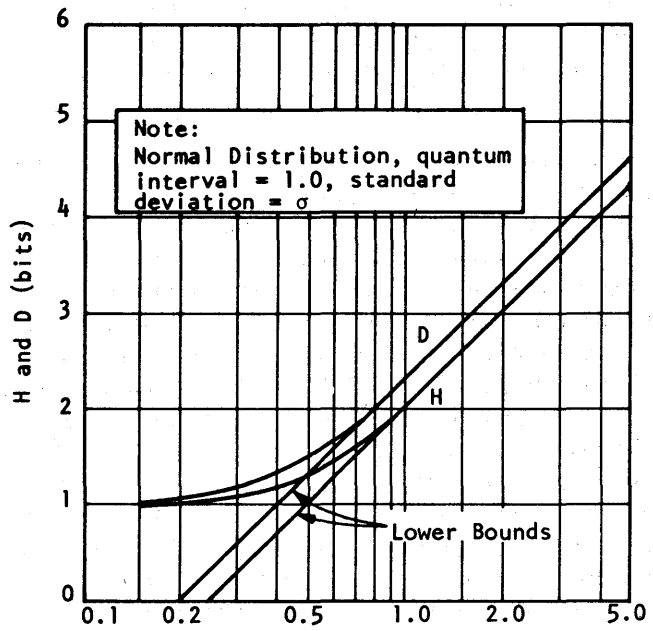


Figure 3—H and D for normal distribution

lating the picture lines from the part of the image that has already been decoded into the region that is being decoded. The probability of "black" for an image element will strongly depend on its position relative to lines in the part of the image that has already been decoded.

Normal distribution

The prevalence of the normal (or Gaussian) distribution in nature is generally exaggerated, but it is mathematically tractable and is a reasonable approximation in many practical cases.

Suppose that the message consists of successive values of a variable v. At any point in the decoding, the next value of v can be predicted on the basis of past values and any other pertinent information. Assume that the actual value differs from the predicted by an amount that is normally distributed with rms deviation equal to σ. For simplicity in the notation, let the quantizing interval be 1. At each sampling time, the expected value of v is $v_0(t)$ based on a knowledge of all the previous values of v. The actual value of v is

$$v(t) = v_0(t) + x(t),$$

where x is the difference between the true value and the predicted value.

Let x have a normal distribution:

$$p(x) = [\sigma(2\pi)^{1/2}]^{-1} \exp(-x^2/2\sigma^2).$$

With unity as the quantization interval, the probabilities of the output symbols are

$$w_i \triangleq p[i<v<i+1] = \int_{i-v_0}^{1+i-v_0} p(x)\, dx.$$

The entropy per symbol is

$$H[\{w_i\}] = \sum_i w_i \log_2 w_i.$$

This equation can be evaluated directly for any values of σ and v_0, but it is more convenient to obtain a lower bound as follows. Note that w_i is the average value of $p(x)$ over the unit interval from $i-v_0$ to $i+1-v_0$. $H[w_i]$ is then the entropy function of this average value. If the function is performed within the integral, this provides the average of $H[p(x)]$. Since the entropy function $H[\{w_i\}]$ is concave down, the average of the function is never greater than the function of the average. When the summation is performed over all i, the integral is extended over all x, and the inequality can be written as

$$H[\{w_i\}] \geq \int_{-\infty}^{\infty} p(x) \log_2 p(x)\, dx. \tag{4}$$

Similarly, for the lower bound on required channel capacity with sequential decoding,

$$D[\{w_i\}] \triangleq 2 \log_2 [\sum_i (w_i)^{1/2}].$$

Again, to get a lower bound, the square root can be performed under the integral, giving

$$D[\{w_i\}] \geq 2 \log_2 \int_{-\infty}^{\infty} [p(x)]^{1/2}\, dx. \tag{5}$$

Integrating Equation (4) provides

$$H \geq \log_2 [\sigma(2\pi e)^{1/2}] = 2.05 + \log_2 \sigma; \tag{6}$$

and from Equation (5),

$$D \geq \log_2 \sigma(8\pi)^{1/2} = 2.33 + \log_2 \sigma. \tag{7}$$

The above bounds are good approximations for large σ. Exact results for $\sigma = 1$ and $\frac{1}{2}$ are tabulated in Table 1 with the expected value in the center of a quantum interval (A) and the expected value on the boundary between two quantum intervals (b).

TABLE I—The Compression and Entropy Values for a Normal Distribution with $\sigma = 1$ and $\frac{1}{2}$

	$\sigma = 1$		$\sigma = \frac{1}{2}$	
	A	B	A	B
H	2.11	2.09	1.24	1.36
D	2.38	2.37	1.43	1.50

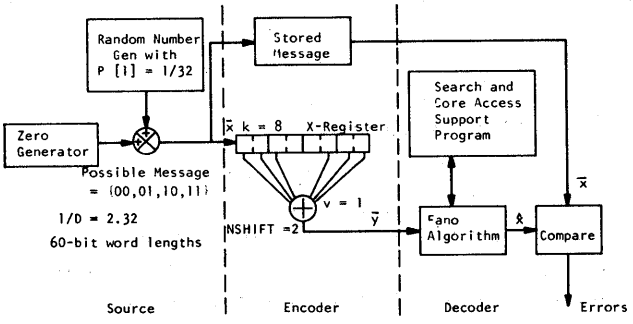


Figure 4—Model of a simple system

The deviations from the bounds are small for $\sigma=1$ and less than 0.31 bit for $\sigma=\frac{1}{2}$.

Figure 3 shows the bounds, which are good approximations for $\sigma>1$, and also the behavior for small σ when the quantization boundary falls exactly on the expected value of v.

D is the number of bits of channel capacity required to transmit each message element. Equations (6) and (7) are derived with the assumption that there is no limit on the range of v. Thus an infinite number of quantization levels are spaced at unit intervals. The amount of compression available at the computational limit depends on the actual number of quantization levels used. For example, suppose a picture is quantized to 64 gray levels (6 bits) and $\sigma=0.8$. From Figure 3, $D=2.0$ and $H=1.8$. The compression ratio for sequential decoding is $6/2=3.0$, and the maximum available is $6/1.8=3.3$.

MODELING A SIMPLE SYSTEM

A system example that used the *a posteriori* probabilities of the source is discussed. Despite the fact that this example is a simple system, it exemplifies all the problems and features of more complex codes and channels, if the nonstationary sources be excluded.

System description

Consider a rate 2/1 data compressor so the source encoding is binary and memoryless with probability of a binary one occurring as $p=p[1]=1/32$. Source encoding consists of shifting in two bits at a time into a shift register and forming a parity check per unit time; source decoding consists of sequential decoding the resulting tree with the Fano algorithm. Figure 4 illustrates the system.

Since the probabilities of two symbols forming the input elements are unequal, the actual information content (entropy) is less than 1 bit/symbol and some compression of the data is possible. As was shown in (3), the compression term may be found as

$$D=\log_2\left[(w_1)^{1/2}+(w_2)^{1/2}\right]^2=0.425.$$

The ratio of input to output symbols is $1/D=2.36$, and for this example a ratio of 2 was used.

Consider a second example pertaining to channel coding rather than source coding pointed out by G. D. Forney. Let the encoder use a nonsystematic rate $\frac{1}{2}$ binary constraint length code shown in Figure 5. The channel is a binary symmetric channel with error probability p. Syndromes that contain all information about the errors and are independent of data are formed at the decoder so that all-zero data can be assumed. A syndrome sequential decoder decodes the syndromes and determines the error locations; the error locations are used to make corrections on the data. It is clear that the syndrome form of Figure 5 corresponds to the encoder of Figure 4, and that the behavior of the Fano algorithm sequential decoder is identical in the two cases. In particular, since $R_{\text{comp}}<\frac{1}{2}$ for $p<4.6$ percent, it can be seen that computation is bounded for $p4<.6$ percent and can be determined for all p.

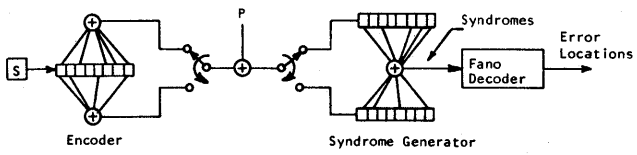


Figure 5—Correspondence of channel coding to source coding

the error probability is that for a rate $\frac{1}{2}$ constraint length 4 nonsystematic code, which would be comparable to J. Heller's results[7] at rate $\frac{1}{3}$ with constraint length 4 if Q were not hard decision. The optimum metric ratio at R_{comp} is $1/-9.1$ and the values used in simulation have been $(1, -4, -9)$ and $(1, -5, -11)$, which are Modulos 5 and 6, respectively.

Simulated channel

The system described in Figure 4 was programmed on the CDC 6400 computer in ascent. The system test consisted of the Fano algorithm decoder, encoder, and random number generator, which generated the binary information stream so there would be control over the number of binary ones. The system performs 12,000 calculations per second. The CDC 6400 has a 60-bit word, which represents the maximum constraint length possible for this system.

Simulations for this system were obtained for 8- and

48-bit constraint lengths at rate 2/1 convolutional codes and sequential decoding 300-bit blocks with a 12-bit tail. The decoder uses hard decision ($Q=2$) input. The code, in octal, used for $k=8$ is 716 as shown in Figure 4 and 7162610413051275 for $k=48$. The computer simulations were made for information content $P(w_1)=4$ percent, 4.6 percent, 5 percent, and 5.5 percent where the theoretical R_{comp} is at 4.6 percent. The results of this simulation are shown in Figure 6, where $P_r(C \geq L)$ is the probability that the number of computations C is greater than the number of computations per bit L when moving through a single branch. Theoretically,

$$P_r(C \geq L) = KL^{-\alpha}$$

where K is a constant and α the Pareto exponent. As the constraint length approaches 8 of this system, which is equivalent to 4 for the conventional convolutional code, the results approach those of J. Hiller[7] with a slight loss going from rate ⅓ to rate ½.

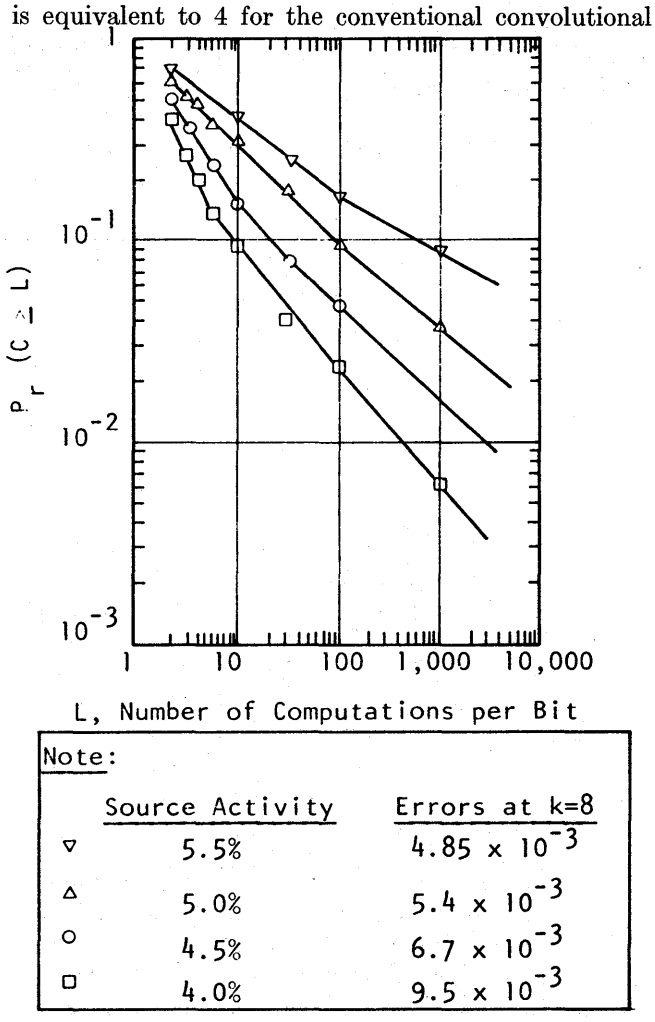


Figure 6—Simulation of convolutional data compression for binary symmetric channel

TABLE II—Summary of Picture Data

Picture	Entropy	Average Gray Level	Standard Deviation	D
1	1.90	50.3	2.48	2.3
2	2.60	54.9	9.22	2.85
3	3.06	34.8	9.84	3.4
4	3.33	50.8	12.45	3.65
5	3.65	41.5	10.08	3.9
6	4.00	48.5	11.79	4.35
7	4.67	41.2	11.61	4.95

Note: (1) Picture fields are 684 elements/line and no less than 598 lines/frame.
(2) Each pixel (picture element) is 6 bits.

APPLICATION OF THE SYSTEM TO A PRACTICAL PROBLEM

One obvious application of convolutional data compression is to work with picture data, say from a space probe. Statistics of the source can be applied to the sequential decoding operation. As an example of this technique, seven digitized pictures have been studied to determine how much compression can be theoretically achieved. The digitized pictures were supplied by R. Rice from JPL.[12] These pictures vary in activity from typical flyby data, which generally are inactive, to pictures taken from landers, which are quite detailed and very active, i.e., difficult to compress. Three typical photos are shown in Figure 7, which represent the data of pictures 1, 3, and 7 respectively. Once these theoretical values are established, the decoder can be implemented to decode similar data.

Gathering source statistics from picture data

The picture elements under study are quantized to 6-bit messages (64 array levels). They vary in activity from an entropy 1.9 to 4.67 bits/picture element on a first difference calculation of one picture element to the element immediately behind it. Each picture has a field of 684×600 elements. Picture data are summarized in Table II.

The key to the data compression decoding problem, whether using sequential decoding or block decoding, is to predict with some accuracy what a picture element value should be given the values of its surrounding neighbors. More explicitly, as the decoder moves through the picture from left to right and from

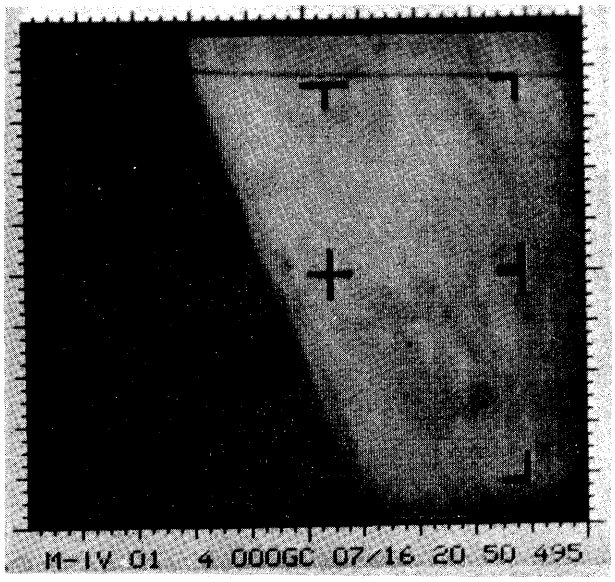


Figure 7(a)—Mariner IV Flyby of Mars. Low picture activity

top to bottom, we ask what is the value of the next picture element.

The correlation value determined for given base pixel x and pixel x_i located at coordinates (u, v) from x is given by

$$c(u, v) = 1 - [(x_i - x)^2 / 2\sigma^2] \qquad (8)$$

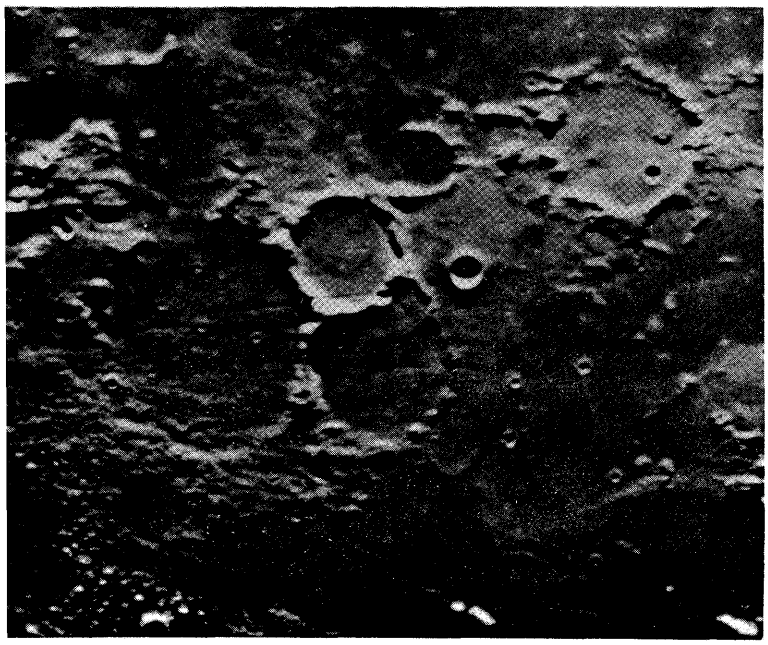


Figure 7(b)—Ranger picture of lunar surface. Medium picture activity

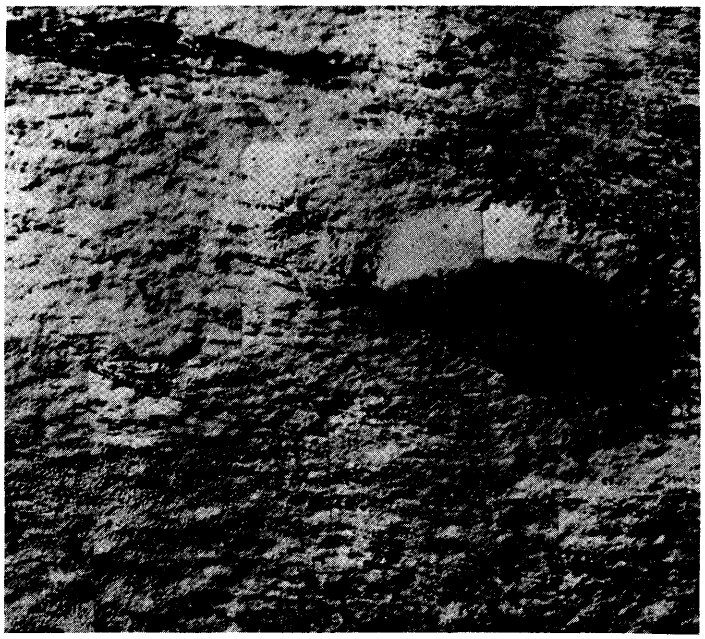


Figure 7(c)—Surveyor picture of lunar terrain. High picture activity

where σ^2 is the usual variance computed for all pixels in the entire picture field. That is

$$\sigma^2 = x^2 - (\bar{x})^2.$$

Obviously, $c(u, v) \leq 1$ with perfect correlation, i.e., $x_i = x$, yielding $c(u, v) = 1$.

Figure 8 shows an enlargement of a set of picture

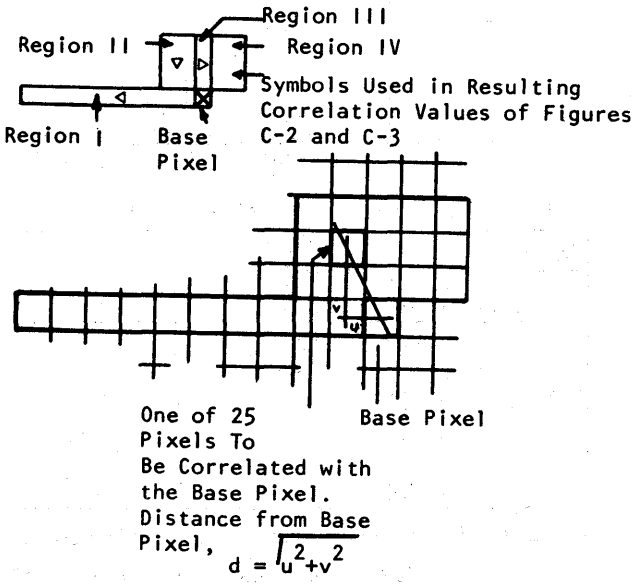


Figure 8—Correlation pattern composed of 25 picture elements (pixels) and one base element

elements. The x base pixel is correlated with each of the x_i elements within the pattern shown in the figure. When considering a pattern of more than one element, such as Figure 8, correlation becomes a function of distance, u and v, between the two pixels being correlated, where u is the horizontal distance and v the vertical distance. For the correlation pattern shown in Figure 8, 25 correlation values will be bound for one base pixel as a function of u and v.

After completing the 25 calculations, the base pixel is moved one unit to the right along the picture line and 25 more correlation coefficients are recomputed. These new results are averaged into the past results. Once the correlation calculations have been computed for one line, the base pixel is moved down by one line and the data again computed and averaged with all the previous work. This procedure was accomplished on a CDC 6400 digital computer.

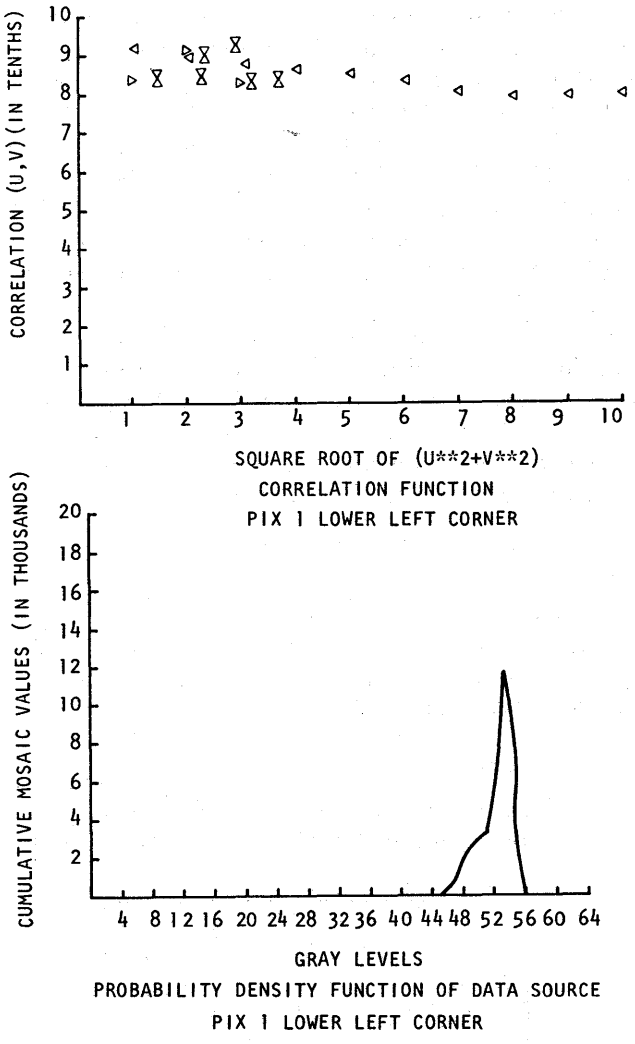


Figure 9—Autocorrelation of picture data using the pattern of Figure 8 (Probability density functions of the areas examined are included)

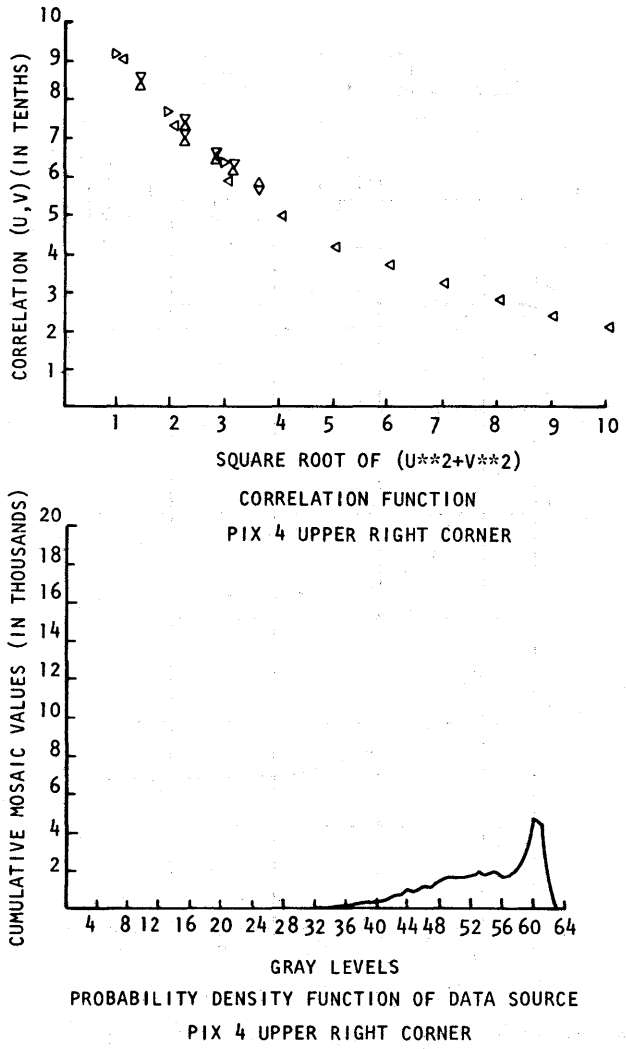


Figure 10—Autocorrelation of picture data using the pattern of Figure 8 (Probability density functions of the areas examined are included)

The correlation pattern was subdivided into four regions, as indicated in Figure 8, to investigate directional variation. Samples of the results of our correlation computations are shown in Figures 9 and 10 for two pictures. Each of the four symbols represents one of the four regions. The probability density functions are also shown for regions in which correlation data were established.

If the analyst wants to apply the correlation data to an efficient prediction program, he must evaluate the results of Figure 8 using a mean square estimation technique to determine how effective the correlation coefficients are in predicting results as a function of various patterns. The correlation patterns must be selected on the basis of which appears to be the most

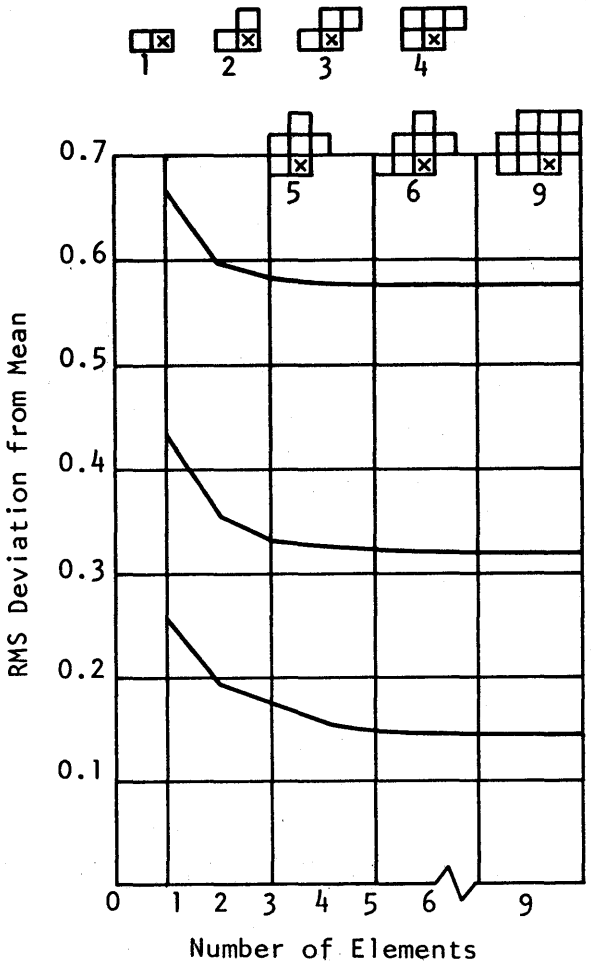


Figure 11—RMS deviation from mean given a set of patterns

efficient. The pattern shown in Figure 8 was used to gather data rather than trying to determine an efficient pattern. These results are then applied to the sequential decoder algorithm.

Eleven pixel patterns were tested. These patterns, along with their performance, are shown in Figure 11. The base pixel to be predicted is noted with an X.

The vertical axis of this figure is the rms deviation from the mean of the base pixel to the surrounding values. High rms deviations represent large errors in estimating the base pixel values. The horizontal axis represents the number of elements used to evaluate the gray levels of the base pixels. The correlation coefficients were used for low, average, and highly correlated pictures. Thus, the top curve represents the rms deviation from mean for low correlated pictures using the different pattern structures shown. The middle curve and the bottom curve represent the average and highly correlated rms deviations from mean, respectively.

From this set of curves, it can be seen that after four or five picture elements there is little need for the additional data supplied by more picture elements to estimate the value of the base picture element.

Extending the model

The example of the preceding section may be extended to handle the picture data discussed here. Knowledge of the picture statistics aids in the decoding process just as it did for the binary case. The only difference in the source is the complexity of the data. For this simulation, use was made of the statistics associated with Picture 2 (Table II), which are highly correlated. The standard deviation of the base pixels of this picture with their adjacent elements was computed at the same time the correlation data were collected. These deviations are shown in their respective locations associated with the base pixel in Figure 12. These values represent an average over all base pixels in Picture 2 and are measured in terms of gray levels. The base pixel was predicted to within $\sigma=2$ or so using the results of Reference 8, which were programmed on a digital computer. The term D from Reference 7 was solved for a Gaussian distribution, which is a function of S. This calculation yields $D=2.33+\log_2\sigma=3.4$ if $\sigma=2$. The system is designed to run at rate $\frac{3}{4}$. The simulation used a convolution encoder with a 6-bit input shift, 4 mod-2 adder outputs, and a constraint length of 60 bits.

The branch metric for the Fano algorithm is selected according to the distance the decoded message was from the guess as a function of σ and the Gaussian distribution with mean of the distribution at $\hat{x}$. A branch metric lookup table may be computed before the decoding operation is started so

$$B_M=\log_2 P=\log_2 f(y)$$

where

$$f(x)=[\sigma(2\pi)^{1/2}]^{-1}\exp-\{(x-\hat{x})^2/2\sigma^2\}.$$

With a rate $\frac{3}{4}$ sequential decoder, four choices are available (if noise is not present) to generate the four branches of the code tree.

System simulation

The pictures were read from a 7-track magnetic tape. The simulation included the encloder, branch metric

	1.80	1.60	1.54
1.70	1.34	base pixel	

Figure 12—Standard deviations of adjacent pixels to a base pixel of picture 2

lookup table, Fano algorithm sequential decoder, comparator, and the appropriate control algorithms. The block diagram is shown in Figure 13. The pattern used for prediction is pattern 4 of Figure 11 with the appropriate coefficients. To avoid startup problems, the first line and first column of each picture was read in and assumed to have been correctly decoded. This, in fact, identifies the single largest problem of systems like this one. It can be overcome, however, as discussed later.

Again the system was programmed on the CDC 6400, which provided sufficient core storage to store the data fields required to make proper evaluation of the base pixel. Searching time, however, is slow (100 searches per second) due in part to the evaluation of the branch metrics. The results of the simulation are shown in Figure 14 when the average number of searches per line is shown as a function of first difference picture entropy. Each picture contains 684 elements per line and 683 coded elements are being simulated. The programs have been limited to 40,000 searching operations. An attempt was made to decode Picture 6, first difference entropy = 4.00, with some interesting results. The system started decoding properly, but after 50 or so lines the maximum number of searches was exceeded and an erasure occurred. The decoding of successive lines deteriorated rapidly where erasures occurred sooner on each line than the line before. Complete erasures occurred 7 to 8 lines after the first erasure was detected.

The system was forced to restart halfway down, and the same phenomena occurred after several lines. The decoder simulation of Picture 7 would exceed the 40,000 calculation limit with every line. Only a small portion of Picture 7 was simulated because of the long run times encountered.

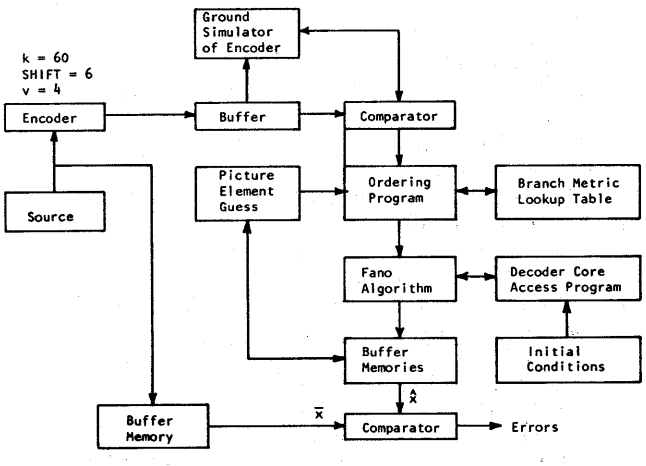


Figure 13—Rate 3/2 simulation model for picture data

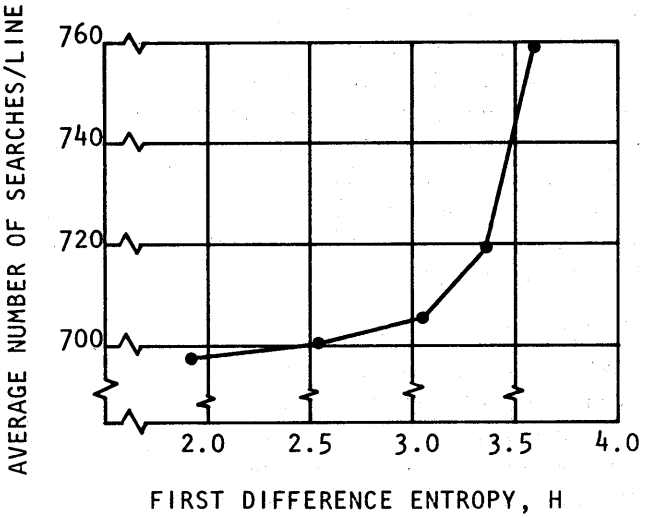


Figure 14—Performance of the 6/4 decoder-compressor

The results imply that once an entropy of a value in the neighborhood of 3.65 is exceeded, then R_{comp} is exceeded. From Figure 3, it can be seen that for $H = 3.65$, $\sigma = 3$. For $\sigma = 3$, $D = 3.9$, which is very near the output value of the convolutional encoder of 4. Theoretically, at least, activity any higher than $H = 3.6$ or so should be difficult to decode. This was verified by Pictures 6 and 7. The simulation uses a pattern of 5 elements but the entropy was computed on two patterns (first differences between the preceding element and the base pixel). Thus, some information should be decoded above the first difference entropy of 3.65.

SPECIAL PROBLEMS

Certain anticipated problems and some possible solutions are discussed next.

Variable activity

Any data compression scheme (such as this one) that maintains a constant rate in terms of data points per unit time must be designed to operate with the most active data expected; consequently, it will achieve substantially less compression than is possible in the dull regions. If the regions of high activity are considered to be analogous to bursts of noise, the analyst immediately thinks of interleaving as a way to even out the data statistics. In interleaving, the encoder would be equivalent to j separate convolutional encoders, each accepting one out of every j consecutive data points.

If there is an interval of active data m data points long, the decoder will only have to search through m/j branch points to get over the active region. Furthermore, if the decoder fails on one of the j channels but succeeds on the preceding and following ones, it can interpolate between these adjacent data values to im-

prove the probabilities for the channel on which it failed, and thus may be able to decode it.

It is also possible to treat regions of high activity by leaving off one or two of the least significant bits in each data word. Other types of processing can also be added to increase the compression. The decision to incorporate them will depend on an evaluation of their cost in complexity, power, and weight, and on the gain in performance they offer.

Startup

When a two-dimensional image is transmitted, the decoder will utilize previously decoded lines to improve the probability estimates for the elements of the line being decoded. Because this information is obviously not available for the first line, some special technique must be used on the first line (and possibly a few more) of each frame.

Perhaps the simplest method is to round off the data in the first few lines by forcing one or more of the least significant bits to be zero. In the course of a few lines, the rounding off would be gradually reduced and finally eliminated. For instance, suppose that 64 gray levels are encoded into 6 bits. The first line might be rounded off to 4 bits and the second to 5. In the third line, every other picture element might be rounded to 5 bits with the alternate elements intact, and the fourth line could have complete data. This would result in a picture with less detail on the upper edge than elsewhere. If this degradation cannot be tolerated, the first line can be transmitted at a lower rate with each picture element being repeated. However, this latter method might seriously complicate the data gathering system.

A similar problem arises at the start and finish of each line because there are fewer neighboring picture elements available to help the prediction. It may be possible to solve this problem by making the ends of the coding blocks coincide with the ends of the lines. The decoder has an advantage at the start of a block because it never has to search back beyond the first node. Near the end of the block, it has a similar advantage because of the series of zeros that is injected to clear the encoding register.

CONCLUSIONS

If computation can be done cheaply at the transmitter, then conventional types of data compression are preferable. Large buffers at the transmitter can smooth out variations in data activity, and uninteresting data can be removed by editing before transmission.

The principal advantage of data compression using sequential decoding is that it requires no additional equipment at the transmitter. When transmitter costs are much greater than receiver cost, as in a space-to-earth or air-to-ground link or where there are many transmitters and a single receiver, this method is likely to be cost-effective and may be the only possible one.

For the space-to-earth link, the savings are in producing software for general-purpose computers on the ground rather than hardware in space. In addition to the obvious saving in reliability and in power and weight on the spacecraft, cost and development time can be saved by avoiding hardware design and qualification test. It is even possible to increase the information rate of vehicles already flying by modifying the decoding program to exploit data redundancy.

REFERENCES

1 R M FANO
A heuristic discussion of probabilistic decoding
IEEE Transactions on Information Theory Vol IT-9 pp 64-74 April 1963

2 J M WOZENCRAFT I M JACOBS
Principles of communication engineering
John Wiley and Sons Inc New York N Y pp 405-476 1965

3 I M JACOBS
Sequential decoding for effective communication for deep space
IEEE Transactions on Communication Technology Vol COM-15 pp 492-501 August 1967

4 D LUMB
Test and preliminary flight results on the sequential decoding of convolutional encoded data from pioneer IX
IEEE International Conference on Communications Boulder Colorado p 39-1 June 1969

5 I L LEBOW R G McHUGH
A sequential decoding technique and its realizational in the Lincoln experimental terminal
IEEE Transactions on Communications Technology Vol COM-15 pp 477-491 August 1967

6 G D FORNEY
A high-speed sequential decoder for satellite communication
IEEE International Conference on Communications Boulder Colorado p 39-9 June 1969

7 J A HELLER
Sequential decoding: Short constraint length convolutional codes
JPL Space Programs Summary 37-54 Vol III pp 171-177 1969

8 E WEISS
Compression and coding
IRE Transactions on Information Theory pp 256-257 April 1962

9 R G GALLAGER
Information theory and reliable communication
Wiley and Sons Inc New York 1968

10 J M WOZENCRAFT I M JACOBS
Principles of communication engineering
John Wiley and Sons Inc New York 1965

11 R BLIZARD H GATES J McKINNEY
Convolutional coding for data compression
Martin, Marietta Research Report R-69-17 Denver Colorado November 1969

12 R F RICE
The code tree wiggle: TV data compression
JPL Report 900-217 STM-324-27 Pasadena California October 1968

Minimizing computer cost for the solution of certain scientific problems

by GERALD N. PITTS and PAUL B. CRAWFORD

Texas A&M University
College Station, Texas

and

BARRY L. BATEMAN

University of Southwestern Louisiana
Lafayette, Louisiana

INTRODUCTION

Many scientific problems require solution of the Laplace, Poission or Fourier equation. These equations occur in heat flow, fluid flow, diffusion and structural problems. It is well known that these types of problems lead to large sets of simultaneous equations that frequently require a number of iterations consuming a lot of computer dollars before a solution is obtained. Frequently one must solve a few hundred to a few thousand simultaneous equations. Numerical methods likely to be used for solution include: (1) Liebmann,[1] an explicit method, (2) alternating direction implicit procedure[1,5] and (3) banded matrix inversion technique.[4]

The computer was first thought to be the salvation of the engineer or scientist who had to solve these types of problems because of the great speed of the machines. In early work with computers large computer appropriations were frequently made available to scientific researchers in this area. The engineer or research scientist could afford the luxury of experimenting with the many solution techniques that required considerable computer time. However, times have changed in terms of computer appropriations. Many groups are now being brought into the computer act, and the budget is divided and distributed. Net computer appropriations to some groups have been decreased or at most have been the same for the past few years. However, the computer is expected to perform more and more tasks.

Budget problems have caused the engineer and scientist to strive for the best solution at the least possible cost. Cost reduction can be a broad area; however, to get to the heart of the problem, this paper investigates the costs of different methods of solving the cited equations. After determining the most economical method, the method is then scrutinized for internal cost reduction.

It was found that as far as cost is concerned, the three methods fell into the following descending order; (1) Liebmann, (2) ADIP, (3) banded matrix inversion. The cost per solution using ADIP was less than 1 percent over the cost of the banded matrix method. The cost of Liebmann, however, was 200 percent larger than the smaller of the other two. ADIP was chosen for special study because of its versatility in solving both large or small transient or steady-state problems. The banded matrix method on the other hand is very limited by its required computer core space and becomes impractical for a large number of equations, that is, more than a few hundred.

MATHEMATICAL DEVELOPMENT FOR ADIP

The partial differential equation that governs the unsteady-state flow of fluid or heat in rectangular coordinates of two dimensions is:

$$k_x \cdot \frac{\partial^2 P}{\partial x^2} + k_y \cdot \frac{\partial^2 P}{\partial y^2} = C \cdot \frac{\partial P}{\partial t} \qquad (1)$$

The ADIP procedure requires that one of the second derivatives, i.e., $\partial^2 P/\partial x^2$, be replaced with a second difference evaluated in terms of the unknown values of P, while the second derivative, i.e., $\partial^2 P/\partial y^2$, is replaced by a second difference evaluated in terms of

known values of P. Thus, in the alternating direction implicit method two difference equations are used.

$$P_{x-\Delta x,y,2n+1}-(2+\rho)\cdot P_{x,y,2n+1}+P_{x+\Delta x,y,2n+1}$$
$$=-P_{x,y-\Delta y,2n}+(2-\rho)\cdot P_{x,y,2n}-P_{x,y+\Delta y,2n} \quad (2)$$

and

$$P_{x,y-\Delta y,2n+2}-(2+\rho)\cdot P_{x,y,2n+2}+P_{x,y+\Delta y,2n+2}$$
$$=-P_{x-\Delta x,y,2n+1}+(2-\rho)\cdot P_{x,y,2n+1}-P_{x+\Delta x,y,2n+1} \quad (3)$$

where

$$\Delta x=\Delta y, \qquad \rho=C\cdot(\Delta x)^2/\Delta t$$

The use of equations (2) and (3) alternately results in a set of tridigonal simultaneous equations that can be solved by a technique illustrated by Douglas[2] and Peaceman and Rachford.[3]

Use of equation (2) or (3) at each time step leads to N sets of N simultaneous equations of the form:

$$A_0+B_0P_0+C_0P_1=D_0$$
$$A_rP_{r-1}+B_rP_r+C_rP_{r+1}=D_r$$
$$A_{n-1}P_{n-2}+B_{n-1}P_{n-1}+C_n=D_{n-1} \quad (4)$$

where

$$1\leq r\leq n-2$$

The solution of these equations can be accomplished as follows:

$$\text{Let} \quad W_0=B_0, \qquad W_r=B_r-A_rb_{r-1}$$

where

$$1\leq r\leq n-1$$
$$b_r=C_r/W_r$$

where

$$0\leq r\leq n-2$$
$$g_0=D_0/W_0 \quad (5)$$
$$g_r=D_r-A_rg_{r-1}/W_r$$

where

$$1\leq r\leq n-1$$

The solution is

$$P_{n-1}=g_{n-1}$$
$$P_r=g_r-b_{r+1}P_{r+1}$$

where

$$0\leq r\leq n-2$$

The w, b, and g are computed in order of increasing r and P is computed in order of decreasing r.

PROCEDURE APPLICATION

Two specific problems are presented here to illustrate the wide differences in computer execution cost of the ADIP method. Both problems represent fluid flow in a reservoir, one in a homogeneous media, the other in a heterogeneous media. The reservoirs are scaled to a 10×10 net or grid size with a permeability coefficient located at each grid intersection. Note, a 10×10 is hardly of practical size, but was all our budget would allow for this study. The desired solution is a matrix of potential or pressure values at a steady-state condition of the system. Each time step of the solution represents one iteration. Each iteration is a small portion of the total computer expense, but becomes quite important in the costs.

Referring to the above equations one may ascertain that each iteration is a function of $\rho=C\cdot(\Delta x^2/\Delta t)$, and therefore the cost analysis will depend primarily upon the effects of this parameter. Table I shows the effect of varying this on the cost for the homogeneous case. Table II shows the effect of this parameter upon the costs for the heterogeneous case.

RESULTS FOR THE HOMOGENEOUS CASE

Table I shows the relative cost for different iteration parameters.

Cost differences can be determined by comparing the effects of different iteration parameters. The solution at the two points studied here are 48.35 and 51.33 respectively. The solutions shown in Table I were determined to be convergent by point material balances. A small iteration parameter implies a large time step, therefore indicating the prospects of obtaining a solution upon fewer iterations. However, this is not always the case because there is a limit to the size of the time-step. For example compare the number of iterations of the third entry to the fourth entry in Table I. It took 12 iterations for convergence for both solutions even though the increment was ten times smaller (meaning a larger time step) for the third entry than the fourth. However, on the whole for the homogeneous case the small iteration parameter (within some limits) will yield a less expensive solution. It was also discovered that the sequence in which the parameters are employed have a tremendous effect upon costs. Again, looking at Table I, the first entry requires six iterations while entry number four requires 12 iterations. They both use the same iteration parameters but one sequence is the reverse of the other. The direction then in this case could mean a 50 percent cost reduction to obtain a solution.

Comparing entry number six with entry number two in Table I shows that a tenth smaller parameter could yield a cost savings of over 50 percent. However when the use of this small and constant parameter (entry

TABLE I—Relative Computing Costs When Using Various Iteration Parameters
(Homogeneous Media)

Program No.	Iterations[1]*	ρ[2]	Increment[3]	$P(4.6)$[4]	$P(9.6)$[4]	Time[5] (Sec.)	Cost[6] $
1	6	.1→1.1	0.2	48.34	51.33	4.42	.44
2	8	.396	—	48.35	51.32	6.01	.60
3	12	.1→1.1	.02	48.35	51.33	8.91	.89
4	12	1.1→.1	.2	48.33	51.34	9.09	.91
5	16	0.01→1.1	.02	48.37	51.35	12.06	1.21
6	17	3.96	—	48.62	51.18	13.34	1.33

*See notes below:
(1) Number of Iterations to Convergence
(2) Iteration Parameter Sequence $\rho = \Delta x^2/\Delta t$
(3) Iteration Increment Within the Sequence
(4) Pressures (psi) at Points (x, y)
(5) Computer execution times in seconds
(6) Approximate cost per equations (IBM 360/65 WATFOR)

number two) is compared with a parameter sequence (entry number one in Table I) it is seen that a cost saving of about 25 percent is achieved by using the sequence.

The three cost saving observations that can be made from Table I are: (1) use as small a parameter as possible that yields convergence, (2) employ some sequence of parameters instead of a single value, and (3) utilize the direction of the sequence that gives the least number of iterations.

RESULTS FOR HETEROGENEOUS CASE

Table II shows the relation between cost and iteration parameter for the heterogeneous case. The heterogeneous case is quite different program-wise than the homogeneous case because it must utilize several computer library functions to generate and maintain the heterogeneous coefficients. The execution times shown in Table II are therefore much greater than those of Table I. This difference in cost magnitude is also considered a factor in determining the economic feasibility of simulating a heterogeneous system over a homogeneous one. The heterogeneous case requires the generation of reservoir geologic information (requiring considerably more subroutines and equations) therefore requiring more computer execution time.

The range of coefficients (k) for the heterogeneous case was (0.1−100). These coefficients were distributed randomly throughout the grid or net. Table II shows

TABLE II—Relative Computer Costs When Using Various Iteration Parameters
(Heterogeneous Media)

Program No.	Iterations[1]*	ρ[2]	Increment[3]	$P(4,6)$[4]	$P(9,6)$[4]	Time[5] (Sec.)	Cost[6] $
1	77	3.96	—	49.34	49.59	59.58	5.96
2	72	.396	—	45.61	45.86	57.05	5.71
3	6	0.1→1.1	0.2	49.37	49.62	32.30	3.23
4	4	0.01→1.1	0.2	49.11	49.54	30.94	3.09
5	100	.1→1.1	0.02	44.06	44.30	69.00	6.90
6	18	1.1→.1	0.2	49.06	49.36	36.31	3.63
7	11	1.1→0.01	0.02	49.21	49.41	34.29	3.43
8	74	1.1→.1	0.02	46.47	46.72	59.62	5.96

*See notes below:
(1) Number of Iterations to Convergence
(2) Iteration Parameters Sequence $\rho = \Delta x^2/\Delta t$
(3) Iteration Increment within the Sequence
(4) Pressures (psi) at points (x, y)
(5) Computer execution times in seconds
(6) Approximate cost per 100 equations (IBM 360/65 O.S.)

the results of several different iteration parameters upon the cost of solution. The exact solution at the designated points were 49.10 and 49.30, respectively. The approximate same generalities can be made about the heterogeneous case that were made about the homogeneous case.

The one-tenth reduction in magnitude of the single value iteration parameter (comparing entry number one to entry number two) results in only a small fraction of the cost reduction shown for the homogeneous case. However, when a sequence of parameters is employed instead of a single value (comparing entry number two with entry number three) a cost reduction of about 90 percent is obtained compared to about 50 percent for the homogeneous case. Again, as in the homogeneous case the direction of the sequence employed can result in considerable savings. For example when comparing entry number three with entry number six, one finds a savings of about 66 percent. For the heterogeneous case the cost was always greater than that for the homogeneous case. A greater number of iterations was required for convergence than for the homogeneous case.

SUMMARY AND CONCLUSIONS

Several factors should be considered before a large system of equations is solved. First, does the problem warrant a large computer expense? It is definitely more expensive to simulate a heterogeneous system than a homogeneous one. Can a homogeneous solution be used, if so, the cost may be only a fraction of the cost of the heterogeneous case. Can a few equations be used rather than a few hundred or thousand?

It was found that ADIP was superior in terms of breadth and cost, although it was slightly more expensive than the banded matrix method. Within the ADIP method itself several dollar-saving techniques may be used. It is better to use a sequence of iteration parameters than a single repetitive value. The direction in which this sequence is employed is very important.

For the homogeneous media it is better to use as small a parameter as possible. This appears to hold true for the heterogeneous media case with some limitations.

If the factors presented in this paper are considered, the savings in terms of dollars can be very substantial.

REFERENCES

1 G D SMITH
Numerical solution of partial differential equations
Oxford University Press pp 149-151 1965

2 J DOUGLAS JR
On the numerical integration of $\mu_{xx}+\mu_{yy}=\mu_t$ by implicit methods
J Soc Ind Appl Math Vol 3 pp 52-65 1955

3 D W PEACEMAN H H RACHFORD JR
The numerical solution of parabolic and elliptic differential equations
J Soc Ind Appl Math Vol 3 pp 28-44 1953

4 B R KOEHLER
Private Communication Texas A&M University
College Station Texas March 10 1969

5 B CARNAHAN H A LUTHER J O WILKES
The implicit alternating-direction method
Applied Numerical Methods Vol 2 pp 543-553 1964

Analytical techniques for the statistical evaluation of program running time

by BORIS BEIZER

Data Systems Analysts, Inc.
Pennsauken, New Jersey

INTRODUCTION

The design of large software systems or real-time systems imposes several constraints on the designer. Predominant among these are the running time of the programs, the amount of memory used by these programs, and the input/output channel utilization. A well considered design not only runs, but has optimum efficiency. Efficiency is often measured by the running time of the program.

If the designer must wait till the program is running to evaluate its running time, important design decisions will have been made, and cannot realistically be changed. Consequently, trades that could have improved the efficiency of the programs will not have been made. This will result in higher unit processing cost, increased hardware, or a reduction of available capacity. In real-time programs, the difference may be that of working or not working at all. For these reasons, the system analyst and programmer require techniques that allow the evaluation of such trades and the early estimation of running time.

Simulation is one method that has been used for timing analysis. The major blocks of the program are described in a simulation language that must be learned like any other programming language. The program simulator is run, statistics gathered, and the efficiency of the program judged thereby.

Analytical techniques, on the other hand, have not been extensively used for several reasons: the analysis has been too tedious for the value of the results obtained; such analyses have required a greater knowledge of mathematics than typical for a programmer; the solutions can be overly complicated (e.g., including transient behavior). In short, both analytical methods and simulation have been effectively inaccessible to the one who needs it most—the programmer. Yet this need not be, if we are willing to make a few analytical assumptions.

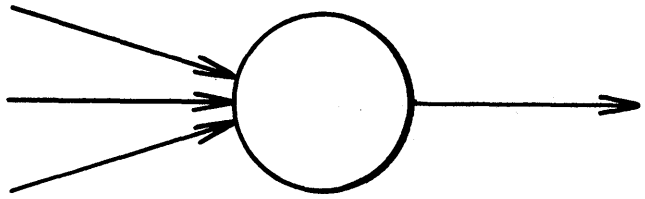
PROGRAM MODELS

Simulation or analysis both require a model. The model that we shall use for a program is based on the flow chart of the program. The model consists of junctions, decisions, and processes, as depicted in Figure 1. Associated with each process is an execution time, obtained by counting the instructions (with appropriate modification for indexing and indirect operations) within that process. Associated with each decision is a probability for each of the several exident branches. The sum of such probabilities must equal 1. Furthermore, the probabilities are assumed to be fixed and not to depend upon how the program got to the particular branch. This will be recognized as a Markov model assumption. Though this assumption is not always valid,* the number of cases in which it does not hold are sufficiently rare to allow us to ignore them. Furthermore, if we do not assume a Markov model, the resulting analysis is overly complex.

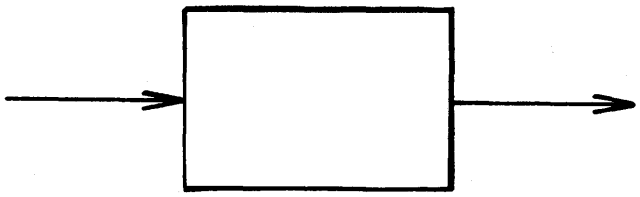
There is one difficulty with this model; the processing that takes place at a decision. It can be readily shown, that for analytical purposes, we can transform a decision to a new decision followed or preceded by processes, such that there is no work done at the decision itself. This is depicted in Figure 2.

Having done this, we can simplify the model further, by eliminating the distinction between junctions and decisions. The new model consists of *nodes* and *links*. That is, the model is a graph. Associated with the outways of each node, there is a probability that that outway will be taken. Associated with each link, there is work required for the execution of the instructions represented by that link. A link then, can represent a

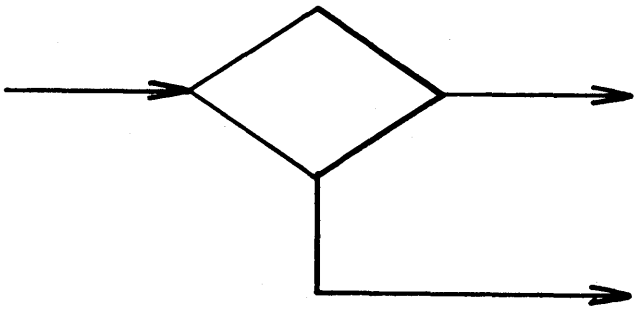
* As an example, consider a program switch within a loop whose position is determined by a count of the number of passages through the loop. While the mean is unaffected, the variance will depend on the way the program got to that point.



JUNCTIONS



PROCESSES



DECISIONS

Figure 1—Basic program elements

sequence of instructions, a subroutine, or a whole program, depending upon what level we do our analysis.

Having gone this far, we can introduce a further generalization into the model. Rather than assuming that the execution time for a link is fixed, we can assume that it is really the mean value of a distribution of running times for the link. We can characterize that distribution by its mean value (μ) and standard deviation (σ). In practice, we shall find it more convenient to use the variance ($\lambda = \sigma^2$) rather than the standard deviation. The resulting model is shown in Figure 3.

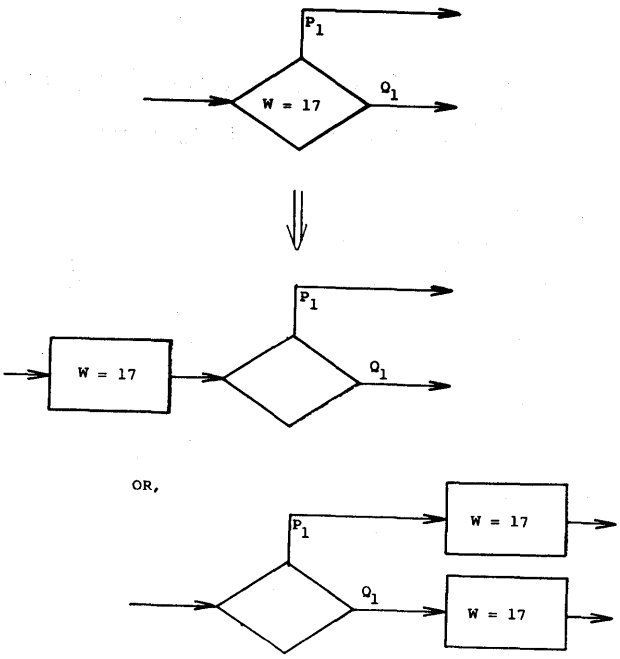


Figure 2—Equivalent decisions

It is clear that any reasonable flow chart and, hence, any reasonable program operating within a single computer at one priority level, can be readily modeled in this manner.

Our problem is then: given the graph corresponding to the flow chart of a program, properly annotated with the required link properties (μ, λ, p), determine the mean value, standard deviation, and the probability associated with every combination of flow chart entrance and exit; there is after all, no need to restrict ourselves to programs that have only a single entrance and exit—that would not be realistic.

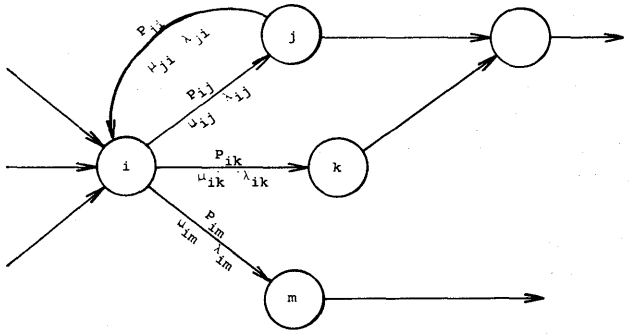


Figure 3—Final model

Before we do this, however, it pays to go into the question of how we obtain these numbers.

ESTIMATION

The running times for individual links are obtained by an estimated count of the instructions in that link. This can be done precisely without programming. The real program must run, but its estimated version need not. We need not take the meticulous care that is mandatory for a real program. Furthermore, since almost all instructions are equivalent, we can replace the real instructions by estimators of these instructions. For most problems, the repertoire can be cut down to about 10 different generic instruction types. Similarly in indexing and indirect operations, we need not be concerned with which index register is used, and so forth.

The standard deviation is either externally supplied or it results from an intermediate step in the analysis. In most computers, the variation in the running time of individual instructions is small and can be ignored.

The difficult part of the analysis is the evaluation of the probabilities associated with the links. Some of these probabilities are externally supplied—that is, they are inherent in the job mix for which the program is written. How these are estimated depends upon the application. Many other probabilities, while difficult to estimate can be ignored. Consider the example shown in Figure 4. We have shown a decision which is followed by two radically different processes that take almost the same amount of time. It is clear that the probability in question is not important. Therefore, a crude estimate will suffice. The third type of probabilities are those which are inherent in the structure of the program. Thus, switches which are set on the first pass through the program and reset on the next pass, the number of times a program will go through a certain loop, etc., fall into this category. These are also readily obtained. Our pragmatic experience has been that about half of the probabilities are data dependent and

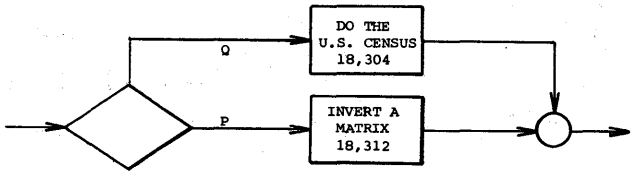


Figure 4—Non-critical probabilities

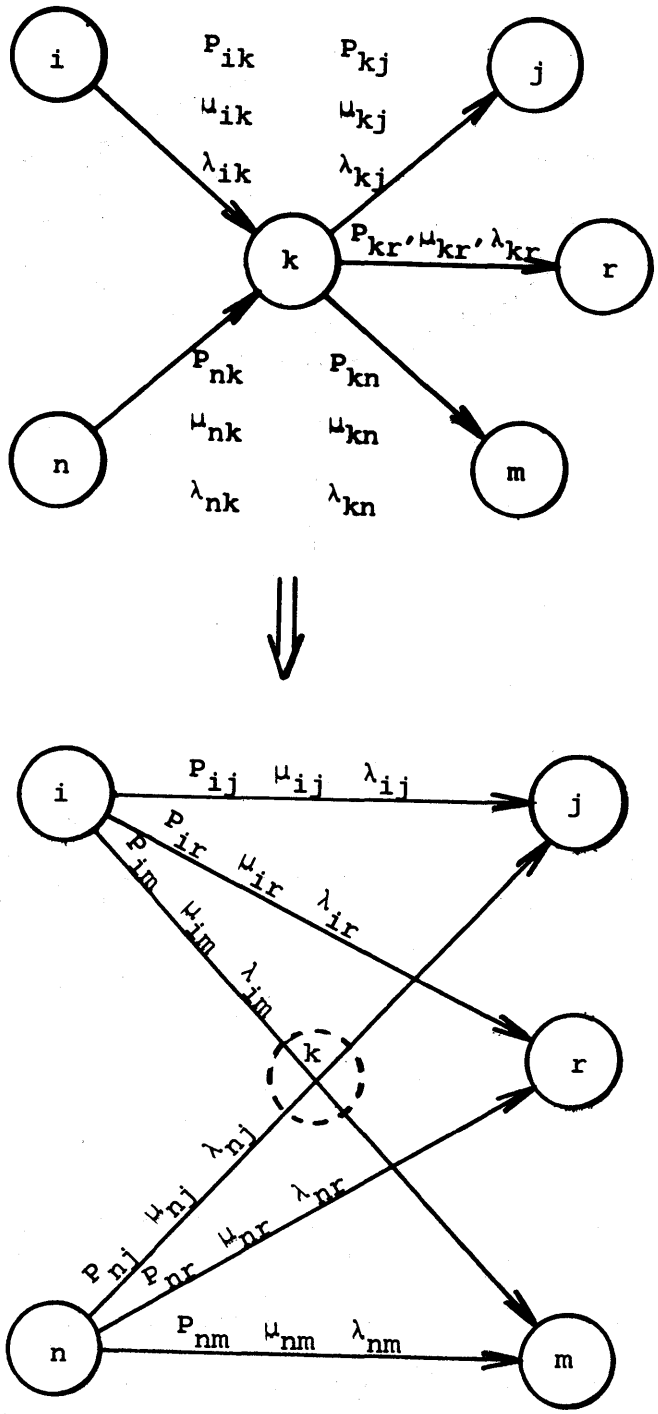


Figure 5—Series case

readily obtained, 20 percent are non-critical, 25 percent are readily obtained from the structure of the program. The remaining 5 percent are sweaty and can require much analysis to obtain. However, since the analytical technique is fast, we can by parametrically examining values of these difficult probabilities, find out if they are indeed critical.

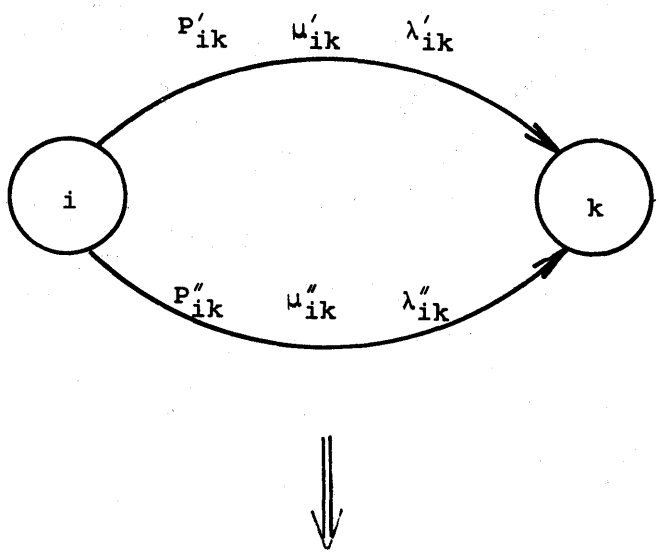


Figure 6—Parallel case

ANALYSIS

The analytical technique is a step-by-step node elimination based on what is sometimes called the "star-mesh transformation." We shall eliminate a node, and its associated incident and exident links and replace every combination of incident and exident link with equivalent links that bypass that node. There are three cases of importance—links in series, links in parallel, and loops. The situations for the series case is shown in Figure 5. The transformation equations for the series case are:[1]

$$P_{ij}=P_{ik}P_{kj}$$

$$\mu_{ij}=\mu_{ik}+\mu_{kj}$$

$$\lambda_{ij}=\lambda_{ik}+\lambda_{kj}$$

The transformation for the parallel case is shown in Figure 6. The equations are:

$$P_{ik}=P_{ik}'+P_{ik}''$$

$$\mu_{ik}=(P_{ik}'\mu_{ik}'+P_{ik}''\mu_{ik}'')/(P_{ik}'+P_{ik}'')$$

$$\lambda_{ik}=(P_{ik}'\lambda_{ik}'+P_{ik}''\lambda_{ik}'')/(P_{ik}'+P_{ik}'')+(\mu_{ik}''^2P_{ik}'+\mu_{ik}''^2P_{ik}'')/(P_{ik}'+P_{ik}'')-\mu_{ik}^2$$

The transformations for the loop is shown in Figure 7. The equations are:

$$P_{ik}=P_{ik}'/(1-P_{ii})$$

$$\mu_{ik}=\mu_{ik}'+P_{ii}\mu_{ii}/(1-P_{ii})$$

$$\lambda_{ik}=\lambda_{ik}'+\lambda_{ii}P_{ii}/(1-P_{ii})+\mu_{ii}^2P_{ii}/(1-P_{ii})^2$$

The algorithm proceeds as follows:

1. Select a node for removal—other than an entrance or an exit.
2. Apply the series equations to eliminate the node. This creates new links.
3. Combine parallel links into a single equivalent link by using the parallel equations.

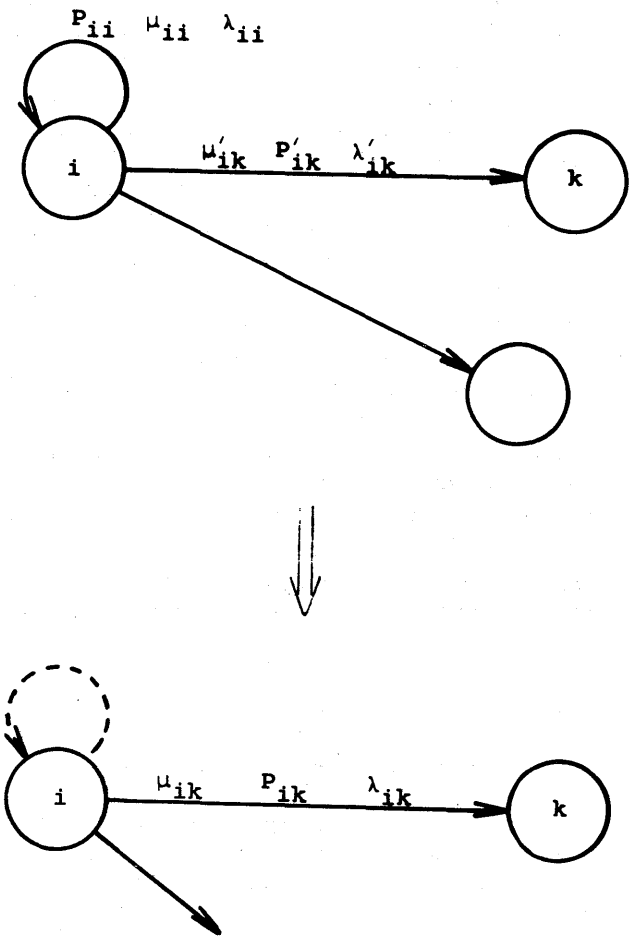


Figure 7—Loop case

4. Eliminate loops.
5. Repeat until only entrances and exits remain.

For manual calculations it is best to represent the flow chart by a matrix. The outmost column and row of the matrix is removed, reducing its rank. We have written a program that performs these calculations, a sample of which is shown in Figure 8. We have here a rather complicated model if we take into account all the possible loops and such. The links are described by the names of the nodes they span. The node names can correspond to the labels in the original flow chart. The special nodes "GILA" and "ZEND" are included as programming conveniences. The output shown has the expected probability of 1 and a mean value of 2263

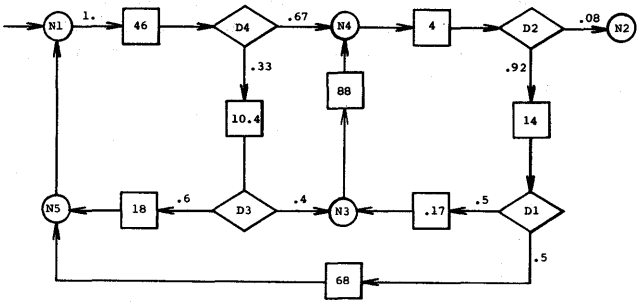


Figure 8—Single inway—single outway example

INPUT

SEQ.	CODE	ANODE	BNODE	PROBABILITY	MEAN	CONTROL
1	INWA	GILA*	N1	1.0000000*	0.0000000*	DIST.
2	LINK	N1	D4	1.0000000*	46.	DIST.
3	"	D4	D3	.33	10.4	DIST.
4	"	"	N4	0.6700000*	0.0000000*	DIST.
5	"	D3	N5	.60	18.	DIST.
6	"	"	N3	.40	0.0000000*	DIST.
7	"	N5	N1	1.0000000*	0.0000000*	DIST.
8	"	N3	N4	1.0000000*	88.	DIST.
9	"	N4	D2	1.0000000*	4.	DIST.
10	"	D2	N2	.08	0.0000000*	DIST.
11	"	"	D1	0.9200000*	14.	DIST.
12	"	D1	N3	.5	.17	DIST.
13	"	"	N5	0.5000000*	68.	DIST.
14	OUTW	N2	ZEND*	1.0000000*	0.0000000*	DIST.
15	ENDL					

END OF LINK ENTRIES

CREATE THE FILE NAME WHERE THE INPUT IS TO BE SAVED. := A39

OUTPUT

SEQ.	CODE	ANODE	BNODE	PROBABILITY	MEAN	CONTROL
1	INWA	GILA	N1	1.000000	0.0000	DIST.
2	LINK	N1	N2	1.000000	0.2263E+04	DIST.
3	OUTW	N2	ZEND	1.000000	0.0000	DIST.

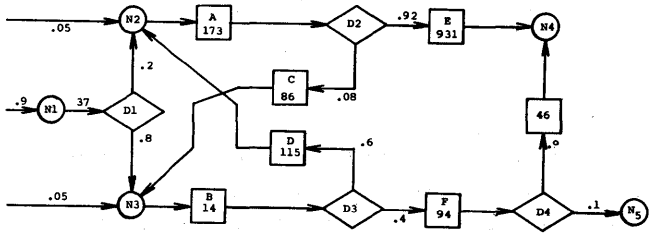


Figure 9—Multi-inway, multi-outway example

INPUT

SEQ.	CODE	ANODE	BNODE	PROBABILITY	MEAN	CONTROL
1	INWA	GILA*	N1	.90	0.0000000*	DIST.
2	"	GILA*	N2	.05	0.0000000*	DIST.
3	"	GILA*	N3	0.0500000*	0.0000000*	DIST.
4	LINK	N1	D1	1.0000000*	37.	DIST.
5	"	D1	N2	.2	0.0000000*	DIST.
6	"	"	N3	0.8000000*	0.0000000*	DIST.
7	"	N2	D2	1.0000000*	173.	DIST.
8	"	D2	N4	.92	931.	DIST.
9	"	"	N3	0.0800000*	86.	DIST.
10	"	N3	D3	1.0000000*	14.	DIST.
11	"	D3	N2	.6	115.	DIST.
12	"	"	D4	0.4000000*	94.	DIST.
13	"	D4	N5	.1	0.0000000*	DIST.
14	"	"	N4	0.9000000*	46.	DIST.
15	OUTW	N4	ZEND*	1.0000000*	0.0000000*	DIST.
16	"	N5	ZEND*	1.0000000*	0.0000000*	DIST.
17	ENDL					

OUTPUT

SEQ.	CODE	ANODE	BNODE	PROBABILITY	MEAN	CONTROL
1	INWA	GILA	N1	0.900000	0.0000	DIST.
2	INWA	GILA	N2	0.050000	0.0000	DIST.
3	INWA	GILA	N3	0.050000	0.0000	DIST.
4	LINK	N1	N2	0.200000	0.3700E+02	DIST.
5	LINK	N1	N3	0.800000	0.3700E+02	DIST.
6	LINK	N2	N3	0.080000	0.2590E+03	DIST.
7	LINK	N2	N4	0.920000	0.1104E+04	DIST.
8	LINK	N3	N2	0.600000	0.1290E+03	DIST.
9	LINK	N3	N4	0.360000	0.1540E+03	DIST.
10	LINK	N3	N5	0.040000	0.1080E+03	DIST.
11	OUTW	N4	ZEND	1.000000	0.0000	DIST.
12	OUTW	N5	ZEND	1.000000	0.0000	DIST.

microseconds. Another example, involving a subroutine with three entrances and two exits yields a somewhat more complicated result and is shown in Figure 9.

The transformation equations can be derived on the basis of very weak assumptions. We assume a Markov model. We assume further that the running time for a link does not depend upon the running time of other links. The only thing that has to be assumed about the distributions representing a link is that they exist and have a first and second moment.[1] Other than that, the equations are valid for any distribution. There are additional refinements to the process to distinguish between deterministic and probabilistic loops that will not be discussed here. Furthermore, the analytical method is also applicable to the determination of memory utilization and channel utilization.

To gauge the efficiency of the procedure; a 100 link program requires less than two seconds of 360/50

time. The running time of a 1000 link analysis is under 10 seconds.

REFERENCES

The algorithms described here were first developed by the author in 1964. The assumptions, however, were overly strong—i.e., required that all link distributions be Gaussian. A more formal derivation based on weak assumptions (i.e., the distribution exists and have first and second moments) is to be found in References 1, 2, as well as a more detailed discussion of non-Markov models in which the node probability depends on the previous history of the program. It is shown there that while the mean value is not affected by these assumptions, the standard deviation is. The algorithm as programmed, is accordingly modified. We have only presented the variance equations for the Markovian case. These equations can be readily shown to yield an upper bound for the variance.

1 P W ARMS JR
Derivation and verification of system 6403 mathematical formulas
Data Systems Analysts Inc 503-TR-3 December 15 1969

2 P W ARMS JR
Forthcoming MS thesis in computer and information science
University of Pennsylvania

3 W FELLER
An introduction to probability theory and its applications
John Wiley & Sons Inc Volume II Chapter XIV 1966

4 B BEIZER
Application manual, system 6403
Data Systems Analysts Inc 503-TR-2 August 22 1969

5 P W ARMS JR
Instruction manual, system 6403
Data Systems Analysts Inc 503-TR-1 August 22 1969

6 S E ELMAGHRABY
An algebra for the analysis of generalized activity networks
Management Science Volume 10 Number 3 April 1964

This paper represents a parallel development in a more general area. Elmaghraby treats the generalized network. It will be seen that the network treated here is his EXCLUSIVE-OR case.

Instrumenting computer systems and their programs*

by B. BUSSELL

UCLA Computer Science Department
Los Angeles, California

and

R. A. KOSTER

North American Rockwell Information Systems Company
Anaheim, California

INTRODUCTION

Considering the high cost and sophistication of data processing equipment, it is almost incredible that techniques for computer system measurement and evaluation have lagged so far behind. While it is true that computer technology has advanced at a rapid rate during the past 20 years, it is surprising that instrumentation for displaying system efficiency has only recently been given exposure in the literature; and even that literature is sparse.

This paper will discuss some recent research in measurement carried on in the UCLA computer instrumentation project. The project proposed by Estrin et al., at the 1967 Spring Joint Computer Conference, was to develop measurement tools and techniques for the purpose of evaluation of hardware and software systems, instead of the historically dominant purpose of measurement for fault location and prediction. The two tools described here have been developed for low level self-measurement, requiring no special hardware.

One tool is an efficient self-simulator that closely duplicates the operation of the machine it is running on. It can be probed at the subinstruction level by a data collection routine to determine detailed characteristics of the programs running on it. A simulator and two data collection routines for this simulator are described.

The other tool is the implementation of an algorithm for making high precision measurements of the time duration of events or activities in the computer. This instrument measures the time interval of an activity in an operating program to about 2 microseconds accuracy using only an 8 MHz clock.

* This research was sponsored by the Atomic Energy Commission [AT(11-1) Gen 10 Project 14].

BACKGROUND

One generally makes measurements on a computer system in order to evaluate it. The evaluation may be for the purpose of acquiring a new system, optimizing an existing system configuration or, perhaps, for specifying an improved system. The measurement may be made on the system itself or on some simulation model of the system. The input for the measurement may be real data or a model of the data, e.g., statistical data. It is to be expected that the most accurate information is derived from measurements made on the actual system, using real data.

Measurements may be made on resource utilization, be it memory space, peripherals or busses. Measurement may be made of computation time, event statistics or program structure. The choice of the measurement must be determined by the specific objective.

Each instrumentation for measurement is expected to introduce artifact; either space, time, program structure modification or event statistics. A measure of the quality of an instrumentation is determined from both the artifact that it introduces and how well the artifact can be separated from the actual measurement in the final evaluation.

Since most of the total running time involved arithmetic operations, early evaluations were based on the time to perform addition or "add time." If more involved arithmetic operations were available as primitive instructions, e.g., multiply, divide—these times were similarly considered in system comparisons. Input

and output times were ignored in computing the run time of a problem and only the actual computing or central processing unit (CPU) time was considered.

As more general purpose applications evolved, costs for commercial users became more important. The job profile for a commercial use is generally one which has large volumes of data to be fed to the system; movement of the data within the system for sorting, merging or extracting pertinent records of files; and finally, output of these updated records or files. Since most of the activity involved peripheral equipment, and since the input and output time was so long compared to the actual computing time, performance measures were generally determined for I/O equipment only. Thus the familiar criteria of "cards per minute," "lines per minute" and "characters per second" became the performance parameters in a commercial computational environment.

These early evaluation procedures can be classified as extremely crude data models applied to a crude system simulation.

Refinements in both the data and system models were made by including additional instructions, and broader classes of operations specific to particular users. For example, instruction-mix statistics have been gathered for scientific computations and for commercial users. These statistics are generally grouped into instruction classes, and weights are assigned to the classes dependent upon the frequency of occurrence as measured in large groups of programs. Examples of mixes can be found in Arbuckle[1] and Smith.[2] The mixes are used to compare computational efficiency between machines. For each operation, the manufacturer's listed operation speed is multiplied by the particular weight factor. A sum of all the products produces a "figure of merit" for comparison purposes. It must be noted that mix evaluations are only generally a measure of the speed of the logic hardware, instruction set.

Other major factors affecting computational efficiency are usually not considered by mix evaluation: special instructions in a particular machine may not be included in the mix; multiple registers with multiple uses, hierarchical memory, and parallel arithmetic units are among special features not usually considered by mixes; mixes are usually developed for the "average" user in some class; some users are not well represented by any average mix.

A refinement over the mix-statistics measure has been the kernel evaluation. In this exercise, small sample routines are coded for each machine under comparison. For scientific users problems such as solving two simultaneous equations in two unknowns, or evaluating a polynomial are typical. Commercial application kernels might be formatting output lines or searching a table for prescribed descriptors. In addition to comparison of computation times, this method may display special features or instructions which are useful (or hindering) in each machine under investigation. A great shortcoming of this method is that it depends greatly upon the experience and skill of the programmer with the particular machine.

Probably the best procedure for comparing systems is to program all of a user's jobs on all of the machines being considered. However, this not being practical, users have attempted to define small portions of typical work loads in order to run them on different machines. These problems are called "benchmark problems" and, of all of the procedures discussed so far, provide the best data and come closest to predicting a "best" system for a particular need. This is true not only because of the better data, but also because of our inability to generate accurate models of today's complex computer systems.

The above described measurement and evaluation procedures were all designed to compare existing systems for possible acquisition. With the advent of parallel processing capabilities, time sharing of resources, multiprocessing, and the inclusion of an increasing number of system functions, the earlier, cruder methods of measurement yielded increasingly poorer evaluations. Other measurement procedures were developed.

First to come upon the scene were channel analyzers. These hardware monitors essentially made measurements on the use of data channels and peripheral equipment from which it is possible to calculate the amount of overlap between the central processing unit and peripherals. Their prime use was in reconfiguring a system to increase efficiency. A later application of a similar device built by one of the large computer manufacturers was to demonstrate to potential customers the increased efficiency of a new product when running the customers' jobs. Recently complex hardware monitors have been described by Schulman[3] and Estrin.[4] Schulman's monitor can attach information in up to 48 signals which describe a maximum of 48 measurable parameters. Through suitable choice of connections significant measurements of system operation can be made during the running of a problem. A less sophisticated version has also been built and at least one manufacturer is marketing a small hardware monitor of this type for general application. Estrin's proposed monitor has, as one of its major variances with other systems, the requirement that the measuring device have the ability to control or interfere with ongoing computations in the object computer system. The purpose of this interference is twofold. First, to be able to capture all essential data by slowing the object computer if the data is being made available too quickly for retention. Second, when simultaneous analysis of the

data indicates more optimal program structure or resource allocation, the measurement system should be capable of making (or initiating) these alterations.

As a first step toward this ultimate goal, software tools were developed to facilitate the collection of data for post-processing. The particular instrumentation to be described in this paper makes measurements of both instruction use and event timing in an ongoing computation in its normal environment in a computer.

INSTRUCTION UTILIZATION

The instrumentation program for measurement of instruction utilization consists of two parts: a data gathering routine, and a control and simulation routine that can be used with many different data gathering routines. The data gathering routine collects instruction mix statistics. The control and simulation routine simulates the object computer on itself, providing a means for data to be collected for each instruction cycle. It is important that nearly every program that would run on the uninstrumented object computer be able to run identically on the simulator. This means that the instrumentation must occupy minimal space and be able to retain control in spite of error, interrupt, or other conditions which may arise from the object program. It is also desirable, for economic reasons, that the simulator operate as fast as possible.

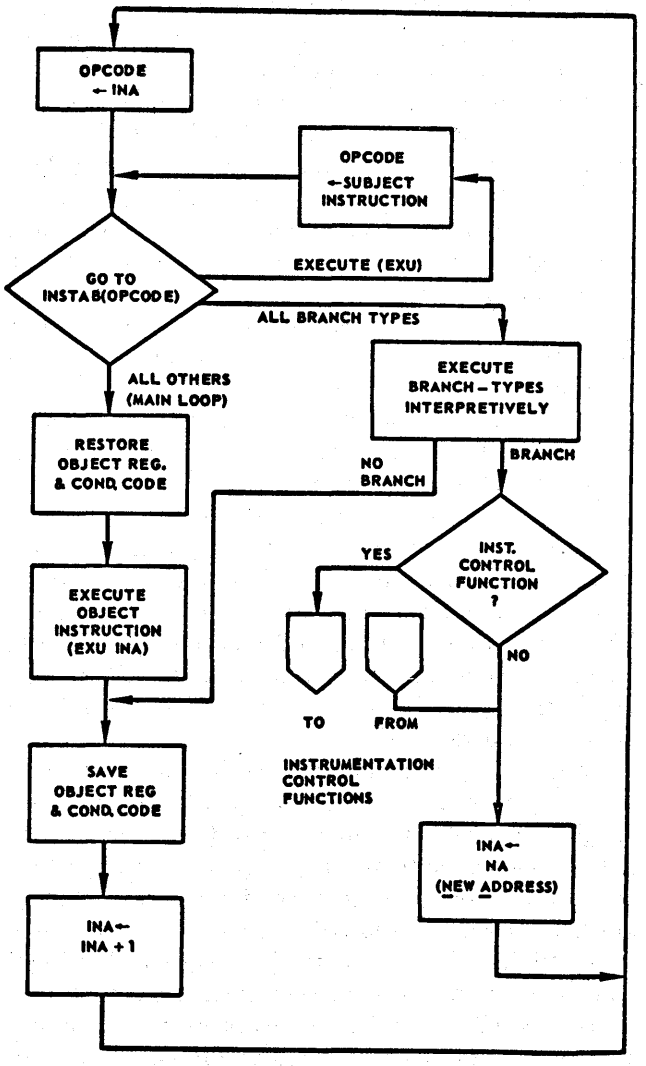


Figure 1—Sigma-7 self simulator flowchart

The central element in the simulator is the Execute instruction of both the Sigma-7 and IBM/360 instruction sets.[5,6] It causes the instruction residing at its operand address to be executed. In the main simulation loop the object program environment, its register and condition code values, are saved after and restored before the EXU instruction (Figure 1). The operand address of the EXU is incremented and the operation code of the next object instruction is examined before executing it. If the object instruction can change the sequence of execution, e.g., a branch instruction, it would cause the simulator to lose control if executed in the main loop. Therefore it is executed interpretively by other routines.

The data collection routine maintains a counter for each operation code that can occur in the object computer. Before executing each instruction, its counter is incremented. Additional counters for the conditional branch instructions record, for each condition tested, whether or not the branch actually occurred.

The dominant elements contributing to the time artifact are proportional to the number of instructions executed under simulation, so relative artifact is usually more interesting to the user. A statement about relative artifact must assume an average execution time for instructions in the object program (running normally, without simulation), which is heavily dependent upon its instruction mix. Register-to-register operations in the 360/75, for example, typically take about 0.5 μsec less than their register-to-storage equivalents. In the Sigma-7 the register-to-register operations typically take 1.5 μsec more than their register-to-storage equivalents. Analysis of a small sample of programs written in assembly language by one of the authors indicates an average execution time of about 1.1 μsec for the 360/75 and about 2.2 μsec for the Sigma-7. Neither sample included any floating point or decimal operations and the author is not necessarily a "typical" coder, so no attempt will be made to defend these numbers as applicable to the general case.

Using the listed "book" values of instruction times it was determined that the ratio of simulator time to book

A. SIGMA-7 SIMULATOR ARTIFACT

Mnemonic	Artifact	Natural Time	Ratio	Instruction (comment)
main loop	18.6	2.2 (av.)	8.2	all not named below
BCS/BCR(b)	41.8	1.0-2.1	42-20	Branch on Condition Set/Reset
BCS/BCR(nb)	39.8	1.9-3.0	21-13	(no branch)
BIR/BDR(b)	41.8	1.5-2.3	28-18	Branch on Increment/Decrement Reg.
BIR/BDR(nb)	39.8	2.4-3.3	17-12	(no branch)
BAL	47.4	2.2-2.8	22-17	Branch and Link
EXU	31.0	1.2-2.4	26-13	Execute (add subject instr. time)

B. SYSTEM/360 - 75 SIMULATOR ARTIFACT

Mnemonic	Artifact	Natural Time	Ratio	Instruction (comment)
main loop	45.0	1.1 (av.)	41	all not named below
BC(b)	36.1	1.0	36	Branch on Condition (RX type)
BCR(b)	28.2	1.0	28	Branch on Condition (RR type)
BC/BCR(nb)	15.5	1.0	16	(no branch)
BCT/BCTR(b)	41.6/33.7	1.0	42/34	Branch on Count (RX/RR types)
BCT/BCTR(nb)	21.0	1.0	21	(no branch)
BXH/BXLE(b)	45.3/37.4	1.1	41/34	Branch on Index High/Low-Equal
BXH/BXLE(nb)	24.7	1.1	22	(no branch)
BAL/BALR	26.6/26.8	1.0	27	Branch And Link (RX/RR types)
LM/STM	24.0 + 1.6r	1.4 + .26r	17 + 6r	Load/Store Multiple (r = no. of registers)
EX	28.5	3.2	9	Execute (add subject instr. time)

Figure 2—Simulator artifact

time for the Sigma-7 instruction repertoire ranged from about 8:1 for main loop instructions to 20:1 (on the average) for branch type instructions (See Figure 2). Since monitor functions, including I/O, are normally not executed under simulation, it is to be expected that the average artifact is lower.

The control routines allow insertion of calls in the object program to command the instrumentation to turn on or off or output its data through three Fortran CALL routines or assembly language branch and link instructions which turn on the instrumentation (INSTON), turn off the instrumentation (INSTOFF), or close the instrumentation and output the data (INSTCLOS). A call to INSTON starts executing the object program under simulation at the next instruction. A call to INSTOFF returns direct control to the object program but retains any data collected so that a later call to INSTON continues the accumulation of data. A call in INSTCLOS outputs and clears the data tables before doing the INSTOFF function. The INSTON, INSTOFF pair can be used to instrument a subroutine or a portion of a program. Since the object program runs at natural speed when the instrumentation is closed or off this kind of control permits significant reduction of artifact.

The infrequency with which calls are made to the control routines causes their relative contribution to overall artifact to be small. INSTON costs about 50 instructions, depending upon arguments it must handle, plus about 50 more if the monitor call is used. INSTOFF costs about 20 instructions plus 50 if the monitor call is used. INSTOUT is about as expensive as INSTOFF plus the cost of the data output routines, which in most cases will totally dominate it. The first call to INSTON performs a number of once-only initializing functions that cost about 500 instructions.

Artifact due to the simulation routines is shown in Figure 2. The Natural Time column gives the manufacturer's published times for the instruction explicitly named and the assumed average time for fixed-point binary and logical instructions executed in the main loop of the simulator. The Ratio columns are a measure of the relative artifact, the artifact divided by the natural time. The average Natural Time in the first lines will be greater, and the Ratio less, in programs having a significant number of floating point or decimal instructions or in programs compiled by a non-optimizing compiler. The range of natural times in Figure 2A assumes no indexing. Add 0.6 to the artifact for only the branch instructions if they are indexed in the Sigma-7. Figure 2B assumes single indexing; e.g., a nonzero base register field, in the artifact column. Add 2.1 μsec to the artifact for RX type instructions if both the base and index register fields are nonzero and subtract 2.1 μsec if they are both zero.

The most important general conclusion that can be drawn from Figure 2 is that the Execute instruction is a definite aid to simulation in the Sigma-7 but, at least as implemented, it is not much help, from a time standpoint, in the 360. In most cases, in fact, the instructions can be executed interpretively a little faster than they can in the main loop. To gain this speed advantage, however, it is necessary to treat practically each instruction as a special case, making the space artifact very large. A related observation is the fact that the EXU instruction in the Sigma-7 adds less than one Add time to its subject instruction whereas the EX instruction in the 360 adds more than three Add times to its subject instruction.

For many experiments it is desirable for the simulator to regain control automatically after an interrupt has given control to the operating system. In most cases this will happen automatically except when the return to the user program is to a point other than that at which the interrupt occurred (as in the implementation of a PL/I "ON" condition). Explicit INSTON commands can be put in all such entries, but that is an error-prone operation for the experimenter. It is also desirable to have an end of job condition, particularly an unplanned one, cause an implicit INSTCLOS function so that the data gathered so far can be recovered.

These goals were accomplished in the Sigma-7 monitor by adding only 25 words to it. It was estimated that a change of similar magnitude would be required on OS/360. They involve the use of a previously unassigned CALL (SVC) instruction to set a flag in the monitor that indicates that the instrumentation system is present. The flag can be explicitly reset and is automatically reset by the end-of-job processor. The routine that gets the address for a non-sequential return and the end-of-job processor check the flag and return instead to an address (in the flag word) that gives control back to the simulator if the flag is nonzero. The address that the monitor would have returned to or the fact that an end-of-job condition has occurred is passed to the simulator entry routine so it can take appropriate action.

The object program

In order to test the above described instrumentation software, an elementary problem was selected which utilized the I/O, performed arithmetic and which had a non-trivial looping structure. The results described below were obtained from tests run on the XDS Sigma-7 using the XDS Fortran compiler.

The object program was a short Fortran program which calculated and printed the amortization table for a loan (Figure 3). The output consisted of 140 lines of data and 3 headlines printed on 3 pages. All variables

```
      'FORTRAN
      INTEGER MONTH(12), PR, RATE, PMT, BAL, YR
      DATA MONTH/'JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG',
     1'SEP','OCT','NOV','DEC'/
      BAL = 800000
      RATE = 60
      PMT = 0
      M0 = 1
      YR = 64
      INT = 0
      PR = 0
      I = 0
      IR = RATE / 10
4     L = 0
      WRITE(6,5) IR, RATE
5     FORMAT('1    DATE', T17, 'AMT PAID', T30, 'INT(', I1, '.', I1,
     1'%)', T45, 'PRINCIPAL', T62, 'BALANCE'/ ' ')
6     IF (L .EQ. 48) GO TO 4
      IPMT = PMT / 100
      IINT = INT / 100
      IPR = PR / 100
      IBAL = BAL / 100
      L = L + 1
10    WRITE(6,11) MONTH(M0),YR,IPMT,PMT,IINT,INT,IPR,PR,IBAL,BAL
11    FORMAT (' ', A3, ' ', I, 19', I2, T14, 4(I6, '.', I2, 6X))
      PMT = 8000
      INT = (BAL * RATE / 12 + 500) / 1000
      PR = PMT - INT
      BAL = BAL - PR
      M0 = MOD(M0,12) + 1
13    IF (M0 .NE. 1) GO TO 15
14    YR = YR + 1
15    IF (BAL .GT. 0) GO TO 6
16    PMT = PMT + BAL
      PR = PR + BAL
      BAL = 0
      I = I + 1
      IF (I .LT. 2) GO TO 6
      WRITE(6,20)
20    FORMAT ('1')
      STOP
      END
```

Figure 3—Object program for instrumentation

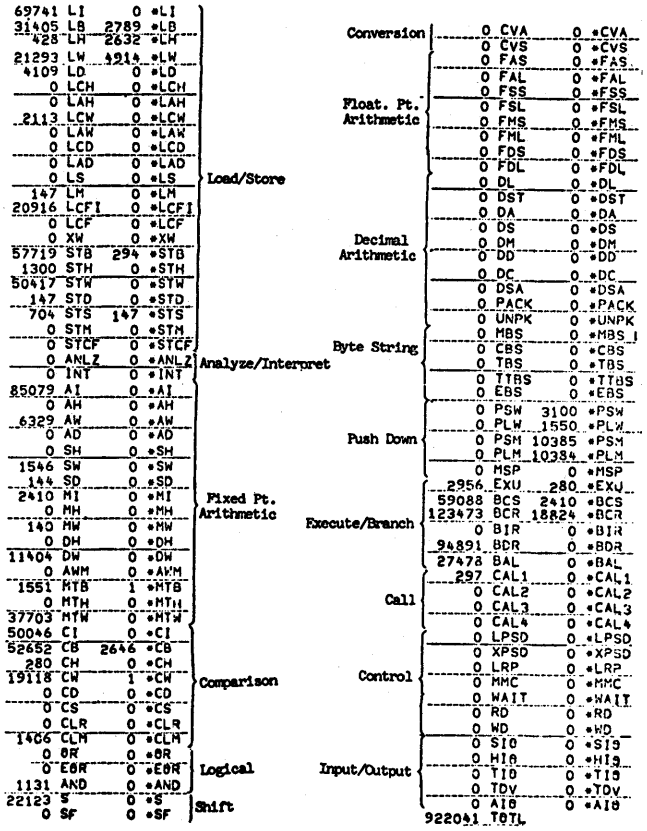


Figure 4—Mix statistics for object program

were integer variables, so there were no floating point operations attempted. Scaling and rounding were done explicitly and the output was formatted with the months spelled out.

The object program was instrumented as a whole and, in order to test the facilities of the instrumentation system, in parts. The latter was accomplished by turning the instrumentation on and off, in order to focus the instrumentation on program sections of interest. Additionally one measurement was made by turning the system on, off, and closing it when the calculations reached January of a specified year. This was done by inserting statements of the form IF(YR. EQ. 67) CALL INSTCLOS before statement 15.

Despite the simple nature of this test program, experiments described below reveal a good deal about the instrumentation process of its potential.

EXPERIMENTS WITH THE OBJECT PROGRAM

Instruction mix statistics

The object program described above was run fully instrumented in order to provide a base for further

experiments, by totalling the number of executed instructions and by displaying the instruction mix compiled for this Fortran program. Figure 4 displays the mix statistics from printout. An examination of this instruction mix indicates that byte instructions are used in compiling the formatted output in preference to byte string instructions. As a test on this hypothesis, a program was rerun with all WRITE instructions disabled. The mix statistics verified the hypothesis—no byte instructions were used. Additionally the total number of instructions used in the latter program was 10,900 (without WRITE instructions) compared to 922,041 for the full program—revealing that approximately 911,000 instructions were used for formatting the printing 143 lines. The program was rerun with different groups of the three WRITE statements suppressed and measurements showed that the headline statement required 5,285 instructions, data printout required (on the average) 6,383 instructions while the page skip control statement required 1,661 instructions.

As a test to see whether or not the method of formatting determined that the compiler used byte string instructions, the FORMAT statements were rewritten using Hollerith format (*xH . . .* ,) and run with the data printout and page skip suppressed. Again the mix statistics showed that these instructions were not used. Further investigation revealed that the compiler had been written to be usable by a smaller system without byte-string manipulation capability.

Program debugging

A debugging aid that was temporarily put into the original simulator has proven so useful that it provides the basis for a separate data collection program. This program records in a circular queue the instruction address, operand address, instruction, and operand for each instruction executed. A queue capacity of fifty instructions appears adequate, but a larger capacity is easily possible. The space required by this program is about 700 words including its data. An instruction being simulated runs about 25 times slower than it would normally. When a program presents a difficult debugging problem, this routine can be loaded with it and turned on just prior to the suspected problem areas. Intermediate and/or terminal (object program fatality) data dumps can be produced. Turning the simulator off during execution of debugged portions of the object program can reduce average artifact to reasonable levels even for long programs.

EVENT AND INSTRUCTION TIMING

Program accesible clocks in present day computing systems are included primarily for accounting purposes. Instruction and event timing require clock resolution which is several orders of magnitude finer than that for billing purposes. It is possible, however, to develop algorithms which can "see through" a low resolution process to the more accurate standard behind it. In this algorithm it is the timing precision of the individual "ticks" of the clock, not the time between them (clock resolution) that limits the accuracy of measurement. Its principle of operation is analogous to that of a vernier scale which proportionally divides the space between marks on a scale to permit a much higher measurement accuracy than would be indicated by the resolution of that scale. It is this type of vernier timing algorithm which has been implemented as a subroutine in the set of instrumentation programs.

The TIME Subroutine measures the actual elapsed time of events within a user program to an accuracy of about two microseconds. It uses an 8 MHz clock in the XDS Sigma-7 computer. It is fully compatible with Fortran and system protocol. The space and time artifact introduced is about 270 memory cells and an average of between 220 and 300 microseconds in most applications.

The TIME Subroutine has two entries, $TIMON and $TIME. A call to $TIME is normally used to start the timer. A call to $TIME is normally used to obtain the time (in ¼ microsecond units) since the most recent return from TIME. Optional arguments passed with the calls can be used to control auxiliary functions of the subroutine. These functions include calibration and hardware clock control.

The vernier clock algorithm has four basic hardware requirements: there must be (1) an accurate hardware clock, (2) minimal jitter, and (3) a very short program loop whose time is (4) accurately known and which can detect when the clock is incremented. The discussion of error analysis will show how each of these requirements enters into the determination of the accuracy of the vernier clock. In some systems (e.g., the XDS Sigma-7 computer) some of the critical times are dependent upon maintenance adjustments, temperature, and other factors that make long term calibration of the timing routine hard to maintain. An analysis of calibration procedure will show that the presence of two clocks makes self-calibration practical. Systems without a second clock may require occasional manual calibration with the aid of special hardware. Simple practical procedures involving attachment of an oscilloscope to a convenient point in the machine while a calibration routine is running and adjustments of a precision oscillator can be developed for most machines.

In some software systems access to the hardware clock by a user program involves high and frequently variable overhead. These effects act like additional clock

jitter and program loop time and may have a fatal effect on the utility of an algorithm to a user. Since the core of the algorithm can be programmed in less than 50 cells, it may be reasonable to make it a resident part of the system programs in some systems. It could then have direct access to the hardware clock. The control, calculation and calibration sections which would link to the core would then be brought in when needed as user or utility programs.

The XDS Sigma-7 computer has two interrupt cells associated with each clock, the count interrupt and the zero interrupt. If an interrupt cell contains the Modify And Test Word (MTW) instruction when that interrupt is activated (advanced to the active state), that instruction is executed and the interrupt is immediately cleared without changing the rest of the machine's state. Unless there are other interrupts waiting, control is immediately returned to the interrupt program. The MTW instruction increments the word at its target address by a specified amount between -8 and $+7$ and tests the result for "less than," "greater than," or "equal to" zero. If the MTW is in the clock count interrupt cell and the result is zero, the clock zero interrupt is triggered (advanced to the waiting state if armed and enabled).

The fast clock triggers the count interrupt every 125 microseconds. This action will be called the tick of the clock. Its timing is dependent upon a crystal oscillator which is probably stable to at least .01 percent. The interrupt will be activated when the currently executing instruction is complete. A similar slow clock triggers its count interrupt every 2 milliseconds at exactly the same time as every sixteenth fast clock trigger. Since the fast clock has higher priority, it will activate first. The time required for instruction execution in the Sigma-7 has been observed to differ from nominal values by 10 percent and from one cell to another, for the same instruction, by 1.2 percent. Short time (a few hours) stability for a fixed instruction in a fixed cell is better than 0.02 percent. No nominal value is given in the manuals for time to execute the interrupt and the MTW, but tests indicate a value of 6.0 μsec with drift and short-term stability similar to that for other instructions.

We can observe the content of a cell that is incremented by the clock before and after an event and calculate the number of whole 125 μsec intervals that elapsed during the event. Multiplying this by 199 (125 minus the 6 μsec required for the interrupt) gives the time of an event $\pm$ 199 μsec. The error is due to the fact that we do not know what part of the first and last intervals were occupied by the event. What is needed is a "vernier" timer that can divide the clock intervals into many small parts and estimate the number of those parts occupied by the object event.

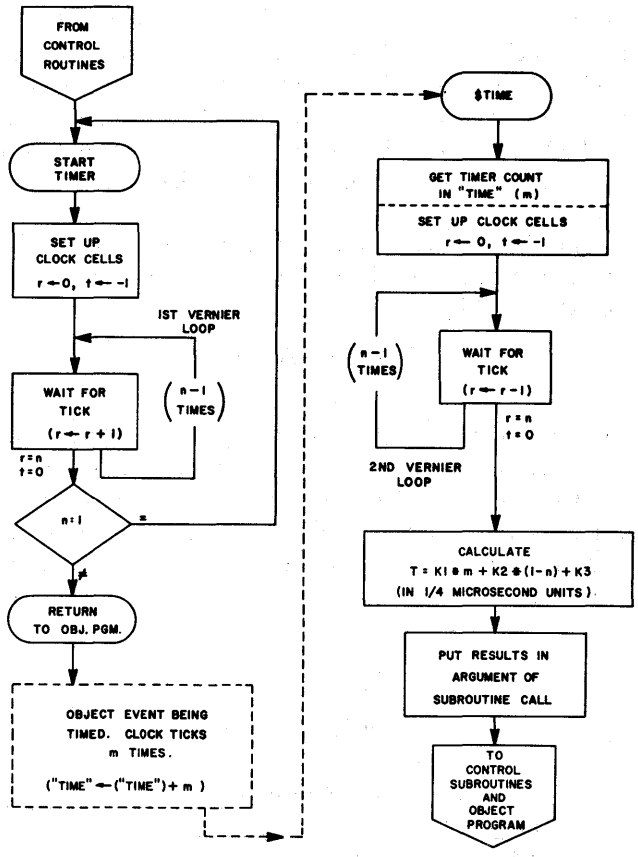


Figure 5—Time subroutine flowchart

The algorithm that makes the vernier measurement is shown in Figure 5. After performing the requested control functions (not shown in the flow chart), the object program's call to $TIMON results in the entry at START TIMER. After setting up the clock cells, the program enters a loop that cycles until the clock ticks. The loop is a single instruction that would infinitely branch to itself except that it is the object of the MTW in the clock interrupt cell. The clock tick causes the branch to go to the following instruction within one branch instruction time. When a tick is detected, control is returned to the object program event that is to be timed. The part of the first interval that is occupied by the object event is now known to a tolerance of the time it takes for one vernier loop. If the number of times through the vernier loop, (n) is one, the timing may be invalid so the timer starts again.

The object event is executed and the object program calls $TIME. After setting up the clock cells and recording the count of the number of times the clock ticked during the event (m), a second vernier loop is

entered. This loop counts the number of times it cycles until the next tick (n) (the BDR, Branch and Decrement Register, instruction is used for the loop to count cycles). The part of the last interval that is not occupied by the object event can now be calculated to a tolerance of the time of one of the second vernier cycles. With this information, the time spent in the object event can be calculated.

Knowing m, the number of clock ticks, and n, the number of vernier cycles in the second loop, the time can be calculated by

$$T = K_1 m - K_2 n + K_3$$

where K_1, K_2 and K_3 are functions of the time and resolution of the vernier loop, the system clock tick, and the system overhead. Variations in the timing in the hardware require that each of the K's be determined experimentally by a calibration routine. However, nominal values for these constants were, in microseconds,

$$K_1 = 119$$

$$K_2 = 1.5$$

$$K_3 = \begin{cases} 89.2 \pm 1.5 \text{ for } n > 1 \\ 89.75 \pm 2.05 \text{ for } n = 1 \end{cases}$$

Experiments

It is of qualitative interest to consider two simple experiments that were executed using the above timing routines. The first measured actual instruction timing in the computer. The second measured the time range and average value for real-time routine.

A program was written to construct a table of instruction times similar to that supplied by the manufacturer except that, instead of specifying memory overlap, it explicitly specified the memory banks used by the instruction. The program was driven by a set of data cards that specified the instruction combinations to be tested. This allowed a subset of the table to be generated or the sequence of entries to be permitted for particular applications. It also allowed unusual combinations; e.g., indirect address and operand both in a register, to be tested.

When operated in the normal mode, the Sigma-7 (with 32K memory) uses one memory bank for even addresses and the other one for odd addresses. The program set up 500 instruction pairs. The even or odd one of each pair contained the specified instruction. The other one in the pair contained a branch to the next instruction (the fastest null operation available). These 1000 cells were bracketed by calls to $TIMON and $TIME. When this sequence was executed, the time was $500I + 500B$, where I is the execution time for the test instruction and B is the execution time for the null operation. The same 1000 cells were then all filled with the null operation and the sequence executed again, giving $1000B$. The value of I was calculated, converted to microseconds, and printed. The first eleven bytes of the input card image were printed to the left of I and the comment field was printed to its right.

A sample output is illustrated in Figure 6 showing timing measurements made to validate manufacturer's book values for instruction times.

Column A gives the octal operation code whose mnemonics are in column I. Column B indicates indirect addressing if an asterisk is present. Column C describes an even address (if 0) or an odd address (if 1) corresponding to column J. If the instruction is not indirectly addressed, this corresponds to the effective address of the instruction. If indirectly addressed, this corresponds to the location of the cell pointed to by the address field of the instruction. Column D indicates indexing if set to 1. Column E describes whether the original instruction address is even if 0, or odd if 1, as indicated in column H. If the instruction is indirectly

A	B	C	D	E	F	G	H	I	J	K
4B	*0	1	0	0		3.00	EVEN	AND,6	*EVEN,1	(IND ADD EVEN)
4B	*0	1	0	1		2.69	EVEN	AND,6	*EVEN,1	(IND ADD ODD)
4B	*0	1	1	0		2.69	ODD	AND,6	*EVEN,1	(IND ADD EVEN)
4B	*0	1	1	1		3.04	ODD	AND,6	*EVEN,1	(IND ADD ODD)
4B	*1	1	0	0		3.04	EVEN	AND,6	*ODD,1	(IND ADD EVEN)
4B	*1	1	0	1		2.73	EVEN	AND,6	*ODD,1	(IND ADD ODD)
4B	*1	1	1	0		2.73	ODD	AND,6	*ODD,1	(IND ADD EVEN)
4B	*1	1	1	1		3.07	ODD	AND,6	*ODD,1	(IND ADD ODD)
30	0		0	0		1.80	EVEN	AW,6	EVEN	
30	0		1	0		1.48	ODD	AW,6	EVEN	
30	1		0	0		1.49	EVEN	AW,6	ODD	
30	1		1	0		1.83	ODD	AW,6	ODD	
30	0	1	0	0		2.42	EVEN	AW,6	EVEN,1	
30	0	1	1	0		2.11	ODD	AW,6	EVEN,1	
30	1	1	0	0		2.11	EVEN	AW,6	ODD,1	
30	1	1	1	0		2.46	ODD	AW,6	ODD,1	
30	*0		0	0		2.70	EVEN	AW,6	*EVEN	(IND ADD EVEN)
30	*0		0	1		2.39	EVEN	AW,6	*EVEN	(IND ADD ODD)
30	*0		1	0		2.39	ODD	AW,6	*EVEN	(IND ADD EVEN)
30	*0		1	1		2.73	ODD	AW,6	*EVEN	(IND ADD ODD)
30	*1		0	0		2.74	EVEN	AW,6	*ODD	(IND ADD EVEN)
30	*1		0	1		2.43	EVEN	AW,6	*ODD	(IND ADD ODD)
30	*1		1	0		2.43	ODD	AW,6	*ODD	(IND ADD EVEN)
30	*1		1	1		2.78	ODD	AW,6	*ODD	(IND ADD ODD)
30	*0	1	0	0		3.00	EVEN	AW,6	*EVEN,1	(IND ADD EVEN)
30	*0	1	0	1		2.69	EVEN	AW,6	*EVEN,1	(IND ADD ODD)
30	*0	1	1	0		2.69	ODD	AW,6	*EVEN,1	(IND ADD EVEN)
30	*0	1	1	1		3.03	ODD	AW,6	*EVEN,1	(IND ADD ODD)
30	*1	1	0	0		3.03	EVEN	AW,6	*ODD,1	(IND ADD EVEN)
30	*1	1	0	1		2.73	EVEN	AW,6	*ODD,1	(IND ADD ODD)
30	*1	1	1	0		2.73	ODD	AW,6	*ODD,1	(IND ADD EVEN)
30	*1	1	1	1		3.07	ODD	AW,6	*ODD,1	(IND ADD ODD)
66	0		0	0		2.73	EVEN	AWM,6	EVEN	
66	0		1	0		2.67	ODD	AWM,6	EVEN	
66	1		0	0		2.72	EVEN	AWM,6	ODD	
66	1		1	0		2.81	ODD	AWM,6	ODD	
66	0	1	0	0		3.34	EVEN	AWM,6	EVEN,1	
66	0	1	1	0		3.29	ODD	AWM,6	EVEN,1	
66	1	1	0	0		3.34	EVEN	AWM,6	ODD,1	
66	1	1	1	0		3.43	ODD	AWM,6	ODD,1	
66	*0		0	0		3.61	EVEN	AWM,6	*EVEN	(IND ADD EVEN)
66	*0		0	1		3.62	EVEN	AWM,6	*EVEN	(IND ADD ODD)
66	*0		1	0		3.57	ODD	AWM,6	*EVEN	(IND ADD EVEN)
66	*0		1	1		3.70	ODD	AWM,6	*EVEN	(IND ADD ODD)
66	*1		0	0		3.67	EVEN	AWM,6	*ODD	(IND ADD EVEN)
66	*1		0	1		3.67	EVEN	AWM,6	*ODD	(IND ADD ODD)
66	*1		1	0		3.61	ODD	AWM,6	*ODD	(IND ADD EVEN)
66	*1		1	1		3.75	ODD	AWM,6	*ODD	(IND ADD ODD)

Figure 6—Instruction timing test output

addressed then the effective address is even if 0 or odd if 1 as shown in column *K*. If the instruction is not indirectly addressed the field has no meaning. Finally, column *G* lists the calculated instruction times in microseconds. For all runs, the contents of register 1 is zero.

Results of these tests indicated that most instruction times were within ±0.10 μsec of the nominal times published by the manufacturer. The test data for identical experiments repeated to ±.01 μsec for four tests made over a two week period. A few instruction times, e.g., Branch on Incrementing/Decrementing Register (BIR/BDR), were as much as .16 μsec less than nominal under some conditions. The odd memory bank appeared to be consistently .04 μsec slower than the even bank for each reference the instruction had to make when there was no overlap (all references to the same bank). Comparison of the times for similar instructions on a condition-by-condition basis showed identical (±.01 μsec) times (examples: Add Word/Subtract Word, BIR/BDR, and Add Word to Memory/Exchange Word (both read-modify-write), Load Byte/Halfword/Word, Store Byte/Halfword (but notWord)).

A three dimensional graphic input device was developed at UCLA. It is similar in concept but different in many details from the Lincoln Wand developed by Roberts at Lincoln Laboratories.[7] The differences stem from efforts to make it lower in cost, eliminate some of the difficulties experienced with the M.I.T. wand, and interface it into the UCLA Sigma-7 computer and graphics system. It is described in detail by Makranczy.[8]

The wand is a hand-held pointer containing a microphone in its tip. Four sonic transmitters are located in the corners of the graphics area. The associated hardware determines the time it takes sound to travel between the transmitters and the microphone. These four times can be used to determine the location of the wand. The calculation to convert the four times into *X*, *Y*, and *Z* coordinates could be done either by special hardware or by the Sigma-7 computations. An embryonic form of the TIME subroutine was used to determine the average time required by the software conversion under simulated tracking conditions. Measurements indicated that 224 (±4) μsec were required to convert one set of data using the software. Since each set was available only once every 40 milliseconds, the computational load was less than 0.6 percent of the CPU time.

CONCLUSIONS

The work described above was concerned with two nearly independent tools for low level self-measurement in computers; self-simulators and timing programs. In each case a need was recognized, satisfactory existing tools were developed to work as well and efficiently as possible within the environment at UCLA (which is typical of many large computer user environments). The methods used are applicable to other environments.

Self-simulation

There is a class of experiments and environments for which self measurement via simulation is the best tool available. It has low initial cost, portability, and easy adaptability to the environment and to varying experiments, compared with external hardware measurement. It does not involve the system shutdown and reliability problems frequently associated with attachment of a special hardware. It can instrument a class of object programs that cannot be statically analyzed. It can collect a class of low level raw data that is not available to other software measurement techniques.

Conversely, measurement via simulation introduces, at best, a rather large speed artifact and a space and facility limitation which tend to increase the cost of operation and restrict the range of measurable systems.

The simulator programs that were described here have a lower artifact by at least an order of magnitude and a greater demonstrated adaptability than any other published simulators.

Timing

While much is said about fast machines, efficient programs, time-saving algorithms and devices, etc., very little has been done to provide the user with a means to measure the basic element of these statements —time. Gross time measurement devices are commonly provided for accounting and control purposes. Microscopic measures such as individual instruction times are provided. But it is the intermediate range of a few tens to a few thousands of instruction times that the user must work with when evaluating, comparing, or improving his programs, algorithms, and operating systems.

The timing program that was described here provides a convenient means for timing events in this range to an accuracy of about one instruction time. When the basic accuracy of the hardware clock is not good enough its short-term stability may still be good enough to allow comparative measurement to the desired precision.

This timing program is practical for a broad class of

time measurement tasks but it does place some important restrictions on the environment in which it can be used and it introduces an artifact that may, in some cases, be intolerable. A range of hardware improvements have been suggested[9] which would eliminate restrictions and/or reduce artifact for these timing functions.

REFERENCES

1 R A ARBUCKLE
Computer analysis and thruput evaluation
Computers and Automation pp 12-15 January 1966

2 J A SMITH
A review and comparison of certain methods of computer performance evaluation
The Computer Bulletin pp 13-18 May 1968

3 F D SCHULMAN
Hardware measurement device for IBM system/360 time sharing evaluation
Proceedings of 22nd National Conference of Association for Computing Machinery pp 103-109 1967

4 G ESTRIN D HOPKINS B COGGAN S D CROCKER
Snuper computer—a computer instrumentation automation
AFIPS Conference Proceedings Spring Joint Computer Conference 1967

5 *XDS Sigma-7 computer reference manual*
Scientific Data Systems Inc Santa Monica Calif
No 90 09 50

6 *IBM reference manual—IBM system/360 principles of operation*
International Business Machines Corp Poughkeepsie New York Form A22-6821

7 L G ROBERTS
The Lincoln wand
AFIPS Conference Proceedings Fall Joint Computer Conference Vol 29 pp 223-227 November 1966

8 T A MAZRANCZY
Logic design and interface of a Lincoln wand system
MS in Engineering University of California Los Angeles June 1969
Also published as UCLA Engineering Report No 69-11

9 R A KOSTER
Low level self-measurement in computers
PhD in Engineering University of California Los Angeles December 1969
Also published as UCLA Engineering Report No 69-57

SHOEBOX—A personal file handling system for textual data

by RICHARD S. GLANTZ

The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

The SHOEBOX system, a part of MITRE's long-term effort in the development of text-processing systems,[1] is designed to be the electronic analog of a personal desk file drawer. A desk drawer is conveniently at hand and readily accessible. It contains documents, reports, adversaria—probably most of which, if the work of others, haven't been thoroughly read, or if one's own work, remain unfinished. This material is organized under whatever whimsical scheme suits one's fancy; and as one's fancy changes, the file contents are variously combined or further segregated. (Or at least one would *like* to perform that kind of reorganization, at present a formidable undertaking.) Needless to add, the texts are set down in a variety of formats, the only common factor among them all being a close adherence to the grammar rules of natural language. Such an unstructured environment is the bane of digital mechanization, but it is to this problem that we have addressed ourselves.

The system described below is more than a passive repository for textual information. As an interactive, time-shared, computer-based system, it can serve a community of users, working privately or in concert. Each user can engage in such activities as browsing, searching, annotating, reorganizing, editing, indexing, and composing. He can digress at any time, perform another activity, and return to his original line of action. He can share his collection with others. He can apply one activity to the results of another: a particularly good example, and one that is possible on few commercial information storage and retrieval systems, is performing a search on the results of a previous search.

Of course, the architecture of the computer and its peripheral gear impose some constraints on the form of the textual information, much the same as typewriter model and page size do in manual filing systems. SHOEBOX has three conventions: a maximum line length of 72 characters, no word division at the right margin, and typography coded as nonalphabetic prefixes to words (for those applications where it is relevant). No attention need be paid to page boundaries, which are, after all, an artifact of paper systems

It must be emphasized here that SHOEBOX* is only a cover term. As relevant research products within the larger context of investigation mentioned at the beginning of this paper come to fruition, they are refined and incorporated into SHOEBOX. Thus the system is a continually evolving one, and this description of it is already obsolete. However, we do intend to maintain upwards compatibility.

Other systems oriented toward handling personal files are being developed by Engelbart,[2] Nelson and van Dam,[3] and Reitman.[4]

DESIGN CONSIDERATIONS

SHOEBOX is designed to be especially accommodating to its human masters. Since the anticipated users of the system may have had no (professional) exposure to computing machinery, we deemed it important to ease the culture/technology shock. At the same time, we wanted to provide a versatile system capable of a wide range of text manipulations. Of equal importance, we wanted a system that could be transported to other computer centers. These overall considerations influenced our decisions in the realms of hardware configuration, software control, and data base structure.

The system is interactive, as mentioned earlier, to provide the user with immediate response to his action and to allow him to react immediately to that response. For terminals, displays were chosen over typewriters or teletypes because of the ability of displays to present large amounts of text almost instan-

* The name derives from the containers in which one of our projected users currently stores his vital information: they bear a striking resemblance to footwear packages.

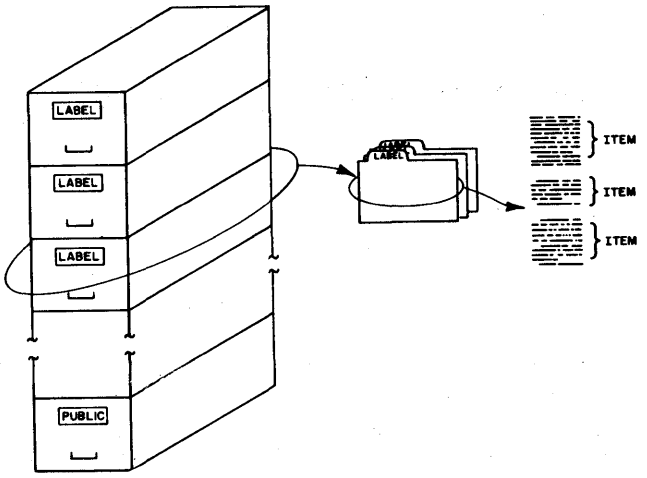


Figure 1—Data base structure of SHOEBOX

taneously. In addition, their comparatively silent operation is more conducive to the human thought processes the system is supposed to enhance. For transportability, the system operates on the rather common IBM 360/50, under OS. The display consoles are IBM 2260's, whose prime attraction is their availability and relatively low price. We would prefer a larger capacity scope with some kind of cursor or position controller like a "mouse," but no acceptable versions were available at the inception of this project, or are now, for that matter. Further details on the hardware and system software are discussed later in the paper.

From the user's point of view, the data base is a collection of files, each labelled with a mnemonic of 24 characters or less. (See Figure 1.) The files may be placed into a number of private file drawers, and each of these also has a mnemonic label. One file drawer, termed *public*, is available to anyone using the system.*

A file is a sequence of lines of text, which may be grouped into items. An item, which is typically several paragraphs in length, may have a one or two character category code associated with each line. The category code can be used to indicate titles, authors, dates, descriptors, personal annotations, or what-have-you. Although the user must identify which lines have which categories, the code scheme itself is completely his choice and need only remain consistent within any one file.

Meta-textual information can be coded as well. For example, if page reference to the hard copy original is needed, a line containing the page number and having the category "p" associated could be used at the change in page.

Note that items, files, and file drawers have no fixed size. They expand and contract to accommodate however many lines of text are inserted or removed.

BASIC OPERATION

To facilitate the detailed discussion which follows later in this paper, it will be helpful if we introduce at least the basic file manipulation operations.

After logging on to the system by identifying himself, the user opens the file drawer he intends to work with and selects a particular file to examine. Until he chooses another file, subsequent commands (with a few obvious exceptions discussed below) are considered to apply to the file he has "placed on his desk." The command EXAMINEFILE(irving), entered on the top line of the CRT, will select the file named "irving" and will display the initial lines of *irving* on the remainder of the scope.*

Beyond the right margin of each text line are displayed the line category, if any, and the line position in the file. Line positions are numbered sequentially upwards from 1; they are automatically maintained by SHOEBOX and need not be a concern of the user except for ontological reference.

To browse through *irving*, the user can repeatedly enter the command NEXT, which will step him through the file one scopeful at a time. The command PREVIOUS reverses the direction of travel. More rapid skipping around is achieved by entering an integer n, which will cause *irving* to be displayed beginning at the nth line. To jump directly to the last several lines of the file, the user enters LAST. Content-directed browsing is also possible. Suppose he is curious as to whether the description "fig newton" occurs later in the file. The command FIND('fig newton') will cause SHOEBOX to search the rest of *irving* for the next line containing the character string "fig newton". The displayed response is either the relevant portion of *irving* or the comment NOTHING FOUND. This is a primitive kind of search; later in the paper we will discuss the more powerful search capabilities available in SHOEBOX.

* There is also a system file drawer, which contains the programs SHOEBOX actually calls on to run itself. This arrangement makes modifying and improving the system quite convenient. Source programs are reorganized and manipulated in much the same way as straight text. This aspect of SHOEBOX will not be discussed here.

* We will adopt the convention henceforth of printing in full caps system commands and system responses.

Personal comments can be interspersed throughout a file. Whatever is entered below the response line on the display can be incorporated into *irving* with the command INSERTSCOPE. The new material will be treated the same as any other part of *irving*, although it would be judicious to mark the comment lines with a distinctive category code.

When the user is finished with *irving*, he can place it back in the file drawer with the command FILE. Alternatively, he can call out another file, say EXAMINEFILE(arnold), which puts *irving* away and places *arnold* on the desk.

With this brief introduction to the "flavor" of SHOEBOX, we would like to turn our attention first, to the very important human accommodation factors, and second, to a detailed exposition of the really powerful things the system can do.

HUMAN FACTORS

In the belief that an interactive computer system should be a complement to human thought rather than an intrusion, SHOEBOX commands are designed to mimic human textual activity. At the same time, actions which could be harmful if employed inadvertently are made somewhat difficult to perform.

One way we have made the system easy to learn is by grading the entire instruction set. The grading is done on the basis of both name and capability. For example, the novice may use the command EXAMINEFILE(irving) to call out the file labelled *irving*. After a brief exposure to the system, he would undoubtedly prefer using the abbreviated equivalent command EF(irving). All long-named, frequently used commands have "obvious" abbreviations: IS for INSERTSCOPE, ML for MOVELINE, SF for SUBSTITUTEFILE, S for SUBSTITUTE, etc. More experienced users can step up to the level of extended commands, e.g., EF(irving 55) will bring *irving* out from the file drawer and display it beginning at line number 55. Thus the user can promote himself to advanced levels to match his own needs, work habits, etc. We are currently engaged in publishing a series of graded instruction manuals and reference guides to aid and encourage this kind of personal development.

Once learned, retention is high. The operations performed by the user, for the most part, parallel human desires, not machine requirements. The choice of command names and their verb or verb-noun construction is a familiar mnemonic device. Consistency also plays a role. The top of the scope is reserved exclusively for user commands; the second line, for system response.

We have endeavored to adhere to the principle that for every user action, there is a machine response. Ordinary conversation between humans becomes trying unless the listener occasionally nods his head or murmurs some sort of acknowledgment. When the conversational partner is a computer, a lack of response or even a delayed response can be deadly. This is especially so for the intermediate-level user: the tyro has no expectations anyway and the sophisticate will sit and curse the system programmers, but the sophomore is the one most likely to panic and perform a disastrous action. After the user enters a command on the command line, SHOEBOX indicates that it has received the command by erasing the response line. If the command is incorrectly formulated, or if the request cannot be carried out for some reason, a comment identifying the difficulty is displayed on the response line. If the command is properly executed, either a confirmatory statement is displayed on the response line or a relevant portion of the file "on the desk" is displayed from the response line down. For commands which are likely to take over six seconds to carry out completely, a running commentary of some reassuring sort appears on the response line.* **

The command set is designed to protect the user against himself. For example, wholesale deletion of material requires just a bit more thought. Unlike most operations on the "desk" file, the command to delete the entire file requires that the file be named specifically. In another situation, the unsophisticated user is not told how to directly replace the contents of one file with the contents of another. While such a command does exist, its hasty application could cause the unwanted destruction of the target file. Instead, the inexperienced user is taught to first *append* the source file to the target file and then delete that portion of the target file he doesn't wish to save. By requiring such deletion actions to be *explicit* on the part of the user, we hope to save him from unnecessary grief.

Perhaps the ultimate in user accommodation is reached with the facility provided in SHOEBOX for coining personal alternates for any of the commands. One of the staff members of the project, for instance, feels happier with EDFIL instead of EXAMINEFILE, STORP instead of SUBSTITUTE, and ADSCOPE instead of INSERTSCOPE. Any such neologism can be defined in the system lexicon with a single command.

---

* At the time of this writing, we have not fully implemented this feature: it appears only on search commands at present, as a rendering of the number of (1) the current line being searched (in multiples of 20, to avoid overdemand on the central processor) or (2) the latest line where a match was found.

** See Miller[5] for an extended discussion of responsiveness in interactive systems.

Furthermore, because the lexicon will handle any number of alternates to a SHOEBOX command name, potentially every user could satisfy his iconoclastic desires.

We all make mistakes. If the user wishes to prevent SHOEBOX from completing the execution of his last command, he hits the "enter" key on the console keyboard again. Control then pops up to the neutral, or waiting state. At this point, the user can return the file, unaffected, to the file drawer with the command GIVEUP. Or, he can change relevant variables (e.g., output format for a PRINTF command) and resume execution with the command RESUME.

When SHOEBOX itself is suspect, the command ? will cause the system to interrogate itself, indicate what its state of activity was, and return to that state. Sample responses to ? are WORKING, WAITING (for a command), and SWAPPING (between users). In case of system failure, the response to ? is a locked keyboard.

SHOEBOX is accommodating up to a point. At twenty errors, it becomes intolerant and logs off.

SPECIAL CAPABILITIES

Editing

Correcting typographical errors, rewording phrases, and inserting and deleting passages are typical editing operations on a text. With SHOEBOX, the author can immediately inspect the results of his decisions. If he remains dissatisfied, he can try again, or he can put the questionable material aside and come back later, or he can make a copy of it for subsequent comparison with alternate phrasings.

There are two basic ways to effect local changes in the stored text. One is to make use of the display hardware and type over the displayed lines of text. When the command SUBSTITUTE is given, the text lines on the scope replace the corresponding lines in the file. As a check, SHOEBOX redisplays the same scopeful, but from the now modified file.

The second way makes use of a search-and-replace command called CHANGE. The command CHANGE('*a*' '*b*' *n*) replaces the string of characters *a* in line *n* of the desk file with the string of characters *b*. If there is more than one occurrence of *a*, only the first will be affected. Taking the previous sentence as an example and with the correct choice of *n*, CHANGE('will be' 'is' n) will transform the consequent phrase to read, "only the first is affected".

With its ability to insert arbitrarily long strings, CHANGE can affect as well the text outside line *n*. On the specified line, deletions are closed up; and insertions are spilled over to a new line (and line numbers greater than *n* are incremented accordingly to allow for the interlineation). The contractions and expansions caused by a CHANGE command are not bubbled through the entire file. That would degrade the response time. Instead, when he has completed his editing modifications, the user can request that the file be PACKed. This operation is analogous to the request one makes of a secretary to "type over" a manuscript; however, with an electronic file, the redoing takes only seconds.

CHANGE is a graded command. If a second numerical argument is specified, as in CHANGE('*a*' '*b*' *m n*), the first occurrence of *a*, if any, in line *m*, line *m*+1, . . . line *n* is replaced by *b*. The command CHANGEALL makes the same replacement for every occurrence of *a*. Multiple changes within a line can be achieved by providing successive pairs of strings in the argument, as in CHANGE('*a*' '*b*' '*c*' '*d*' *m n*). Another CHANGE-type command affects the line category code only.

An extensive error analysis is provided which informs the user of an error in syntax, the non-occurrence of *a* in a line, the non-occurrence of a line number in the file, and, for the wise guys, the requirement that *m* cannot be greater than *n* nor can either of them be less than one or greater than the number of lines in the file.

Copying

Copies of items or of entire files can be made and placed at the end of any file or within any file drawer. For safety's sake, however, no command can make reference to a particular place within a file unless that file is on the desk (so that the affected portion can be visually inspected). Thus, only portions of the desk file can be copied and placed at specified points within the desk file.

The commands which copy from the desk file are exemplified by the following:

APPENDTO(arnold 34)—places a copy of line 34 of the desk file at the end of *arnold* (which can also be the desk file)

INSERTLINE(12 34)—places a copy of line 34 of the desk file before line 12 of the desk file

In the first example above, if *arnold* doesn't exist in the file drawer, a new, empty file with that name will be made; and then the appending operations will be performed. Items as well as individual lines can be

AFTER COMPLETING THIS QUESTIONNAIRE, HIT "ENTER".
.....IS THE FILE YOU WANT SEARCHED. TYPE SEARCH REQUEST BELOW:
.....ARE THE CODES OF THE LINES YOU WANT SEARCHED IN THE FILE.
.....IS YOUR NAME FOR THE FILE WHERE THE ANSWERS WILL BE PUT.
.....ARE THE NUMBER OF LINES BEFORE THE MATCH LINE THAT YOU WANT IN EACH ANSWER
.....ARE THE NUMBER OF LINES AFTER THE MATCH LINE THAT YOU WANT IN EACH ANSWER
.....DO YOU ALSO WANT THE DATE OF THE RELEVANT DOCUMENT ALONG WITH EACH ANSWER?

Figure 2—Questionnaire for SIMPLESEARCH

copied: APPENDTO(arnold 34 52) will copy the item encompassed by lines 34 through 52 to the end of *arnold*. INSERTLINE(12 34 52) would be processed in a similar fashion. Further, extensions of these commands allow several non-contiguous passages to be copied at the same time. In this manner, an extract of an article could be taken, and stored in a file of such extracts or tacked onto the article itself. The command INSERTFILE, with appropriate arguments, will copy an entire file and place it before a specified line of the desk file.

To make a separate copy of a whole file, there is no need to take the file out of the file drawer. The command COPYF(irving arnold) makes a copy of *irving* and labels it *arnold*. If there already is an *arnold* in the same file drawer, its contents are discarded first. Because of the danger of catastrophe with this command, the incautious user is well advised to employ instead the following two-step procedure:

EXAMINEFILE(irving)

APPENDTO(arnold 1 Last)

If there was something in *arnold* beforehand, the appending operation will preserve it. An alternate safe procedure is to enter first

EXAMINEFILE(arnold)

and then, only if the response is ARNOLD NOT A FILE, enter

COPYF(irving arnold)

The command APPENDFILE(irving arnold) makes a copy of *irving*, if it exists, and appends it to the end of *arnold*, if it exists. Thus, a third alternative is

APPENDFILE(irving arnold)

Searching

While rapid access and facile manipulation materially aid the collector and analyst of information, the crux of the problem is discovery and rediscovery of what is pertinent. There is always an uncomfortable feeling about pigeonholing material under headings or even tagging material with a multitude of descriptors—critical snips of information seem to get lost by this precategorization.

The situation is especially acute in public affairs, where personalities, events, and issues which received little notice in the past may suddenly move to center stage. When past history lies scattered among the files, retrieval is difficult. The ability of SHOEBOX to search on the actual words in the texts, as well as on descriptors and categories, greatly enhances the reliability of the search.

We have already introduced our most primitive search command, FIND. Briefly, FIND('fig newton') will search the desk file, from the second text line currently displayed to the end of the file, for the first occurrence of the string "fig newton" which lies completely within a single text line. (At the request of some users, we have implemented a related command, FINDANCHORED, which looks only along the left margin.)

A higher degree of search capability is attained with SIMPLESEARCH. When this command is entered, the questionnaire in Figure 2 appears on the scope in response.*

Note that:

1. the file to be searched need not be on the desk
2. the entire file is searched
3. the search can be restricted to lines with certain categories
4. the search does not cease at the first match, but begins again at the next item
5. the "answers" are not displayed but put into a file for subsequent inspection and manipulation

* Not explicitly called out in the questionnaire are the values for two system variables, which indicate (1) the category assigned to the date line (or any other category line to introduce each answer) and (2) the category of the line which begins or terminates every item. We assume these variables, once set by the user, are uniform for his entire file collection.

TABLE I—Composition Possibilities for SIMPLESEARCH

Concept	Example	Searches for all occurrences of
Word	newton	*newton, newton's, newton?, newton.'', 3newton* (supposing the prefixed numeral denoted typographical coding), *3newton-3raphson, newton-sec/m*; i.e., any matching string of characters delimited by non-alphabetics
Concatenation	fig newton	*fig newton, fig newton's, "fig" newton*, etc.
Disjunction	fig | date	*fig, date, fig's, date?*, etc.
Specific Optionality	newton %interference rings	*newton interference rings, newton rings*, etc.
Complementation	newton ¬interference rings	*newton diamond rings, newton pastry rings, newton circus rings*, etc., but not *newton rings* or *newton interference rings*
Phrasing	date | (fig newton) date | fig newton	*date, fig newton*, etc., but not *date newton;* *date newton, fig newton*, etc., i.e , disjunction takes precedence over concatenation (and over specific optionality and complementation, as well)
Anchored Stem	'newton*'	same as the word *newton*, plus *newtonian, newtonians, newtons*, and (*newtonian*), *newtons'*, etc.
	'*newton'	same as the word *newton*, plus *antinewton*, etc.
Free Stem	'*newton*'	both the above, plus *antinewtonian*, etc.
	'newton'	equivalent notation
	'3newton'	*3newton, 3newton's*, etc., but not *newton, newton's*, etc.
Non-specific Optionality	newton 1 rings	*newton rings, newton interference rings, newton diamond rings, newton pastry rings*, etc.
	newton 7 rings	*newton*, etc., followed by up to 7 intervening words and stems, followed by *rings*, etc.

6. pre- and post-context can be specified (with FIND, the pre- and post-context are fixed at 0 and 9 lines, respectively)

We will describe the form of the search request by illustration rather than by formal definition. In this instance, a formal statement would be pedantic. The reader is first advised to consult Table I.

As an example, let us suppose an analyst keeps a file of newswire reports on South Vietnam. Let us suppose further that many months later he is assigned a task which involves retrieving those news bulletins which give actual figures in mentioning American troop withdrawals. One of the search prescriptions he could try is:

'*000' %us 'trōop' 3 'withdr' | ('pull' out)

Reports containing any of the following phrases would be identified and retrieved:

. . . 30,000 US troops withdrew . . .
. . . 5,000 US troops will withdraw . . .
. . . 46,000 troops will be pulled out . . .
. . . 10,000-man troop withdrawal . . .
. . . 365,000 troops could have been withdrawn . . .
. . . 44,000 troops to pull out . . .

It should be noted here that, under SIMPLESEARCH, unlike FIND, these phrases can be found even if they begin and end on different lines.

After the search has been completed, the results and a copy of the questionnaire itself are placed in the designated answer file for examination. Consequently, the search can be reformulated and applied to the source file or the answer file. If a third file is designated as the repository for the second search, the two results can be compared. Or, the same search can be applied to another file without rekeyboarding anything except the file label. Finally, using the annotation and reorganization commands, our Vietnam analyst can compose a report and incorporate the corroboratory search results.

A third degree of search capability is reached with CSEARCH (for *contingent* or *complex* search). This search request provides greater selectivity over the choice of items to be searched within the file and allows greater control over the format of the answers. Especially potent is the search prescription itself: it allows unordered search terms, contingent searches, and synonymic substitution.

The command CSEARCH causes the questionnaire in Figure 3 to be displayed.

```
**NAME OF FILE->                                                          1
**SEARCH FROM LINE->          **TO LINE->                                 2
**REMEMBER LINE CODES->(            ) **IGNORE->(      ) **TERMINATOR->   3
**SEARCH REQUEST ('LINE CODES' 'SPECIFICATION'; . . . ;)-> (               5
                                                                          6
                                                                          7
                                                                          8
                                                                          9
                                                                        ) 10
```

Figure 3—Questionnaire for CSEARCH

(The line numbers are displayed to allow modifying the answers to the questionnaire with the SHOEBOX editing commands.) Line 1 asks for the name of the file to be searched; line 2 allows the user to limit the extent of the search; line 3 provides for the setting of certain system variables; and lines 5 through 10 accept the search prescription. Table II lists the additional composition possiblities under CSEARCH.

Conjunction is typically used when dealing with key words or descriptors. For example, the search prescription

(k d) wood, preservative

would be expected to find items relating to preservatives for wood, where d and k are the categories of the lines containing the index terms.

$NUM is a two-argument function which can be embedded in a search specification. To illustrate with a simple usage, let us assume that the entire search prescription is of the form:

c $NUM('$m$' '$n$')

where c is a category code and m and n are non-negative integers (with or without internal punctuation; e.g., *52,000, 69.03.14, 3.14159*). Then any item whose c line has a number in the closed interval $[m,n]$ will be found and selected (provided m and n have a consistent format). This numerical capability has many applications; the example shown in Table II for calendar dates is particularly appealing.

One of the most attractive facilities within CSEARCH is the power of pattern substitution. Consider our analyst again. Suppose now he is given the assignment of accumulating all reports which mention cease-fire violations in the Middle East. He might begin by formulating a tentative search specification

arabs 3 'attack' israel

which will find sentences such as "Arabs attacked Israel last night". But since many nations fall under the rubric "arab", he will soon realize that a better specification is

jordan | egypt | lebanon | syria | iraq |

(saudi arabia) 3 'attack' israel

This phrasing will find "Jordan has attacked Israel", etc. Using stems instead of words, as in

'jordan' | 'egypt' | 'leban' | 'syria' |

'iraq' | (saudi 'arabia') 3 'attack' 'israel'

he can locate "Jordanian troops attacked Israeli positions", "Lebanese artillery attacked Israeli fortifications", etc. Rather than have to rekey this representation for "arabs" each time he modifies his search request, the analyst can define them to be mem-

TABLE II—Additional Compositional Possibilities for CSEARCH

Concept	Example	Searches for all occurrences of
Conjunction	date, fig	both *date, date*), etc. and *fig, fig/*, etc. anywhere in same text portion
	date, fig newton	both *date*, etc. and *fig newton*, etc., i.e., concatenation takes precedence over conjunction
Numerical Interval	$NUM('69.01' '70.03')	items dated between January, 1969 and March, 1970 [see text for further explanation]
Substitution	$CLASS snacks	the components dictated by the search pattern labelled "snacks" [see text for further explanation]

bers of a labelled set. Then he would write the previous search specification as

$CLASS arabs 3 'attack' 'israel'

where he has defined $CLASS arabs, after further thought, to be the long disjunction

'arab' | 'guerilla' | (al fatah) | 'jordan' |
'egypt' | 'leban' | 'syria' | 'iraq' |
(saudi 'arabia') . . .

In this manner, he has defined the set of entities he can henceforth refer to as "arabs". This maneuver not only saves him from keyboarding this long pattern again as he constructs related search prescriptions, it also saves him the bother of remembering or redetermining what political groups of people comprise Arabs in the Middle East. Thus he can readily specify the reverse request,

'israel' 3 'attack' $CLASS arabs

"Attack", of course, is not the only verb which connotes aggressive activity. The analyst could again reformulate his search specification as

$CLASS arabs 3 $CLASS aggress 'israel'

where he has defined $CLASS aggress to be the pattern

'attack' | 'bomb' | 'shell' | 'strik' |
struck | 'hit' | 'straf' | 'raid' | 'sink' |
sank | sunk

These class definitions become a permanent part of the user's file collection. At any time, however, he can employ the editing commands of SHOEBOX to examine the CLASSes he has defined and to effect a change in content or label.

$CLASSes can be embedded in one another. For example, journalistic style allows such synecdochic expressions as "Amman claims their artillery shot down. . ." Our analyst might very well wish to set up an equivalence between country and capital, viz.:

$CLASS israel : = {'israel' | (tel aviv)}
$CLASS jordan : = {'jordan' | amman}
$CLASS egypt : = {'egypt' | cairo}
$CLASS lebanon : = {'leban' | beirut}
$CLASS syria : = {'syria' | damascus}
$CLASS iraq : = {'iraq' | baghdad}
$CLASS saudi : = {(saudi 'arabia') | ——*}

* The capital of Saudi Arabia is left as an educational enterprise for the reader.

and in turn:

$CLASS arab : = {arab | guerrilla | (al fatah) |
$CLASS jordan | $CLASS egypt |
$CLASS lebanon | $CLASS syria |
$CLASS iraq | $CLASS saudi}

and perhaps even further:

$CLASS mideast : = {$CLASS arab | $CLASS israel}

Concomitant with the power of embedded $CLASSes is the danger of self-referral. CSEARCH expands all $CLASSes in the search prescription given it before actually proceeding with the search. If there is an infinite loop, or if the expanded search prescription is so huge that it threatens to take up all the allotted list space in the computer, the comment VICIOUS LOOP is displayed on the response line of the scope.

The concept of $CLASS is also useful in dealing with spelling variations in transliteration. The name of the leader of Al Fatah has appeared in print as "Yasser", "Yassir", and "Yasir". Often one entity has several names: trinitrotoluene, TNT; lysergic acid diethylamide, LSD, LSD–25; Songmy, Song My, Son My, My Lai, Mylai, Mylai #4, Mylai Village #4, Pinkville. Establishing a substitution class both records the alternations and ends the reliance on one's memory.

The indolent and the impatient can also use $CLASS to advantage. Why keyboard "Society for the Prevention of Cruelty to Animals" if a substitution class with "spca" can be set up? If there is no standard acronym, make up your own nickname; for example:

$CLASS rocky : = {governor nelson h rockefeller
| nelson rockefeller |
the new york governor}

When first conceived, $CLASS was seen as a means to facilitate synonym sets and thesaurus hierarchies (and hence the name, $CLASS). As programmed, however, it is an instrument for wholesale pattern substitution. Any pattern, even a search prescription itself, can be defined as a $CLASS. Thus, a frequently required search prescription can be incorporated within a $CLASS and called out for subsequent use.

There is one search mode in CSEARCH left to discuss—contingency. If a sequence of search prescriptions is given, separated by semicolons, the subsequent search prescription only applies to the items satisfied by the previous prescription. While effectively only a conjunction of searches, this mode of search will save a considerable amount of search time in large files.

By way of illustration, the following search prescription

```
    d   $NUM('69.06.20' '69.08');
    a   south vietnam;
(h b)   '*000' %$CLASS us 'troop' 3
        'withdr' | ('pull' out);
```

will, first, select from the file all items whose *d* line contains a date between June 20, 1969 and the end of August, 1969; second, select from this subset all items whose *a* line contains the descriptor "south vietnam"; and third, select from this new subset all items whose *h* or *b* lines contain a text passage satisfying the search specification shown.

At the completion of the search, a line number reference to each "hit" and the total number of "hits" are displayed. Based on this information, the user may decide to broaden, narrow, or reformulate his query. Or he may decide to place the items found by CSEARCH in an answer file. To do this, he completes another questionnaire, which allows him to label the answer file, to store with the answers the original search questionnaire with the $CLASS terms in the search prescriptions expanded out, and to control the formatting of the answer file and certain associated data.

Reorganizing

Armed with the results of his searching, the user can make new files, discard unwanted files, merge files together, rearrange items within a file, and move files from one file drawer to another. For the author, reorganizing commands can be used effectively to cut-and-paste portions of text in writing reports. Indeed, in organizational structures where reports are passed from the lowest stratum to the highest level through a series of middle management filters, SHOEBOX offers a convenient way to extract data from an incoming report, add salient observations of your own, and pass the "new" report on to the next higher level.

The items within the desk file can be placed in ascending or descending order with the command ORDER. Among the arguments to this command is a sequence of {category code, sort key} pairs. If, for any item, the first pair cannot apply because the code and key do not co-occur in that item, the second pair is tried, and so on. This feature is useful, for example, if we wished to arrange a file of documents by author, or, if none, by editor. By employing ORDER several times, one can organize the items in a file say, first with respect to date; and within date, by author; and within author, by title.

Items can be discarded from the desk file with the command REMOVELINE(*m n*), where *m* and *n* are the line numbers delimiting the unwanted passage. Alternatively, they can be shifted to the end of another file, say *irving*, with REMOVETO(irving *m n*). Or, they can be shifted to another position within the desk file, using MOVELINE(*k m n*), where *k* denotes the line number before which the passages are to be inserted. Of course, if $m=1$ and $n=$Last, the entire file is affected.

Comments keyboarded on the display may be inserted into the desk file with INSERTSCOPE(*k*); or they may actually replace specified lines in the desk file, using SUBSTITUTESCOPE(*m n*). Scope comments can also be appended to the end of any file, say *arnold*, with the command APPENDSCOPETO(arnold).

Input/Output

A hard copy of *irving* can be printed off-line with the command PRINTF(irving); similarly, PUNCHF(irving) punches *irving* onto cards. Other commands allow the reading and writing of files from and to seven- and nine-track tape or any OS sequential data set. For example, we have successfully passed files from SHOEBOX to IBM's Conversational Programming System (CPS) and vice versa; we also have passed files to a KWIC program.

Housekeeping

All files are automatically date-stamped by SHOEBOX with the day/month/year that the file was last changed in any way. When a file is brought out for display, this information, together with the name and size of the file, is displayed on the response line.

If the user wishes to see all the file labels in his collection, he enters the command LISTF ( ). The argument within the parentheses determines the file drawer; if there is no argument, all file labels for both public and private file drawers are displayed.

For both training and system debugging purposes, a record is made of all commands and other material entered into the system at the display keyboard and all displayed responses by the system. The hour/minute/second when each command was entered is also saved. This record-keeping function can be partially or totally deactivated at log-on time. Otherwise, the record accompanies the off-line printout for each user.

HARDWARE AND SOFTWARE DETAILS

SHOEBOX is implemented on the IBM 360 under OS/360 (PCP, MFT, or MVT), in a 225K high- or low-speed partition, and with a private IBM 2316 disk pack. If SIMPLESEARCH and CSEARCH are not needed, it will operate in 140K. The system as presently implemented will support up to eight IBM 2260 Model 1 (12×80) display stations in a local configuration supported by the Graphics Access Method (GAM). Budget considerations, however, have limited our work experience to four stations. These terminals can be located up to 2000 feet from the display controller. The transmission speed is quite fast: 2500 characters per second, or on the order of half a second to write the entire scope.

SHOEBOX is written in TREET, a list processing language[6] derived from LISP; and the system is run under the TREET/360 operating system. The TREET filing system operates, using OS direct access techniques, by dividing a single OS sequential file into small individual segments called pages, which are dynamically allocated and released. A SHOEBOX file is an ordered set of these pages. (The line numbers of the file are not stored with the file, but they are generated as required.) In this manner, modification of the file does not entail rewriting the entire file, but only the pages affected and the file directory. The directory is hash-coded, so that file access time is nearly independent of the number of files.

The approximate capacity of the system is 100 to 300 file labels per file drawer (depending on label length) with direct access to 2000 double-spaced typewritten pages of text per file drawer. The number of file drawers is dependent on the amount of space available on direct access storage devices.

Initial response time to a request is on the order of a few seconds. Time to process completely the request is a function of whether other jobs are utilizing the central processor or the input/output equipment, whether we are given a partition in high- or low-speed core, and the complexity of the request.

Because of the experimental nature of the project, plus the variety of accounting algorithms that *we* have been experimental guinea pigs to, we have been unable to compile meaningful operating cost figures for SHOEBOX. We hope to be able to do so in the near future.

CURRENT DEVELOPMENT EFFORTS

In the SHOEBOX system described here, searching within a line is carried out sequentially, character by character. An experimental system has been programmed to test the potential savings of an index search. That is, for each file, an alphabetized list of its words was compiled, together with a set of pointers to their locations in the text. There are two advantages to searching the index instead of the text itself: (1) the time of search is more-or-less independent of the size of file, and (2) the search, being sentence-oriented, takes cognizance of natural text boundaries; e.g., a request for the phrase "fig newton" won't deliver the item containing the passage, "The Birge-Vieta method isn't worth a fig. Newton-Raphson isn't worth much either." There are compensating difficulties: (1) what algorithm determines sentence boundaries (recall that a period has more than the terminal function)?, (2) how should right-anchored stems occurring in the search prescription be handled?, and (3) must the index be recompiled after every modification to the file? Nonetheless, our findings with this experimental program showed savings in search time of an order of magnitude. With this encouragement, we are in the midst of incorporating a similar index search in SHOEBOX.

For use in a community of research workers, we would like to implement a "mailbox" feature. At log-on, the individual's name would cause any messages left for him by others to appear on the display. A less intrusive alternative would be for a message like SOMETHING IN YOUR MAILBOX to appear.

Other projects in the early design stages are (1) graded error messages, (2) structured file drawers, and (3) statistical capabilities.

Under consideration is a facility for designating one member of a synonym set as having "print" status. This option would be especially welcome in composing reports: the author could keyboard his *personal* acronym or nickname for a wordy phrase or hard-to-spell proper noun, but the display and the file would contain the "*official*" forms.

It is our intention to incorporate into SHOEBOX increasingly sophisticated linguistic techniques of the kind represented in SAFARI,[7] a MITRE text processing system in which the information content of English sentences is stored and retrieved directly.

ACKNOWLEDGMENT

In the age of big science, progress is a group effort. Henry Bayard, Carter Browne, Stanley Cohen, Jeanne Fleming, Louis Gross, Ted Haines, Lewis Norton, and Donald Walker have all contributed to the design, implementation, and testing of the system. We also are indebted to Thyllis Williams of Inforonics, Inc., for guidance in both design and testing.

REFERENCES

1 D E WALKER
Computational linguistic techniques in an on-line system for textual analysis
MTP-105 The MITRE Corporation July 1969

2 D C ENGELBART W K ENGLISH
A research center for augmenting human intellect
AFIPS Conference Proceedings Fall Joint Computer Conference Vol 33 Part 1 pp 395-410 1968

3 S CARMODY W GROSS T H NELSON D RICE A VAN DAM
A hypertext editing system for the /360
In M Faiman and J Nievergelt (ed) Pertinent Concepts in Computer Graphics
University of Illinois Press pp 291-330 1969

4 W REITMAN R B ROBERTS R W SAUVAIN D D WHEELER W LINN
AUTONOTE: A personal information storage and retrieval system
Proceedings 24th National Conference of the ACM pp 67-75 1969

5 R B MILLER
Response time in man-computer conversational transactions
AFIPS Conference Proceedings Fall Joint Computer Conference Vol 33 Part 1 pp 267-277 1968

6 E C HAINES
TREET, a list processing language and system
MTP-104 The MITRE Corporation March 1969

7 L M NORTON
The SAFARI text processing system: IBM 360 programs
MTP-103 The MITRE Corporation September 1968

HELP—A question answering system*

by ROGER ROBERTS

University of California
Berkeley, California

INTRODUCTION

HELP—A Question Answering System—enables a user, sitting at a console of a time-shared computer, to obtain information based on questions typed in. This information might concern the operating system itself, the format of commands to the user interface executive program, the use of a selected subsystem, or an area totally separate from the computer. The content of the data base in HELP is completely arbitrary, and determined by the creator of each individual HELP system. Questions are presented to HELP in standard English and elicit one or more separate responses, depending upon the nature of the question. If HELP cannot generate an appropriate response, a standard "I don't know" message is output. A second system, called QAS, was developed to enable a user to conveniently generate a HELP program. This paper will discuss the structure of both programs. All of the work discussed in this paper was performed on a modified SDS 930 computer, developed at Project Genie at the University of California, Berkeley.

BASIC PHILOSOPHY

One of the major considerations in the design of HELP was to produce a system with a fast response time for the majority of the questions it encountered. In other words, a system was desired which would require no more than one second between the time it was called to the time it was ready to accept a question, and, for 75 percent of all questions, would require no more than one second between the time a question was terminated and the time the printing of the answer began. It was felt that this constraint was necessary since HELP was to be an aid to users sitting at terminals, in the process of using the system. Users would ask questions of HELP, in their course of work at a terminal, in the same maner as they would consult a reference manual. Therefore, if HELP was to be useful, it had to be easier and quicker to use than a manual.

A further design consideration was that many different HELP systems would be constructed, each with its own data base and each by a different person. This implied that a "shell" would be designed, which contained all of the logic necessary for HELP, but without a data base. A working HELP system would then consist of this "shell" and an appropriate data base. Since many different people would be constructing HELP systems, the procedures for building a data base should be uncomplicated.

The above considerations led to the following conclusions. The analysis of the questions was to be kept as simple as possible. Complex syntactic analysis was ruled out since activation and response times had to be low and core space had to be limited. In addition, the relationship between the questions and the responses had to be straightforward, so as to facilitate the construction of a data base.

That a system could be designed with these constraints was supported by work done prior to this paper. In this preliminary version of HELP,[1] it was observed that the meaning of most questions, of the type we would encounter, is independent of word order. This observation allowed for a design which only reacted to particular words in a question, called KEY WORDS, and ignored both the word order and the remaining words. Using this mode of analysis produced a question answering system which both conformed to the restrictions stated above and correctly answered the questions put to it.

STRUCTURE

The primitive objects used by HELP are called KEY WORDS. These are words which have been

* This work was partially supported by Contract No. SD-185, Office of the Secretary of Defense, Advanced Research Projects Agency, Washington, D.C. 20325.

defined previously, and only a member of this set of KEY WORDS will be considered in the formation of the answer. Certain sets of these KEY WORDS are singled out as defined KEY WORD LISTS, and these KEY WORD LISTS are used by HELP in determining what answer to give. The basic idea is to extract the KEY WORDS from the question, and from this set to determine what KEY WORD LISTS are present. For example, assume that the words "file", "open", and "input" have been defined as KEY WORDS and, in addition, no other words have this property. Then the question "What is a file?" would present to HELP the set of KEY WORDS "file", the question "How can I open a file?" the set "open, file", and the question "What is used to open a file for input?" the set "open, file, input". These sets of KEY WORDS are the only pieces of information which are extracted from the questions and, in fact, are unordered sets. "Which instruction is used to open an input file?" would generate the same set as above, namely "open, file, input". Notice also that all words not KEY WORDS are ignored, so the question "Input file open?" would have been just as meaningful to HELP.

Now that we have these KEY WORDS, what do we do with them? As mentioned above, some sets of KEY WORDS are defined to be KEY WORD LISTS, and these lists are used to determine what information should be given in response to a question. When creating a data base for HELP (to be described below), a KEY WORD LIST is defined by specifying the KEY WORDS which comprise the list *and* the response to be generated when this list is recognized in a question. This link between a KEY WORD LIST and a response is the major mechanism which HELP uses to answer a question. To return to the above example, assume we have now defined the KEY WORD LIST "file" to have the response "A file is a collection of data. . . ." Also, assume we have linked the KEY WORD LIST "open file" to the response "The instructions to be used to open a file . . ." and the KEY WORD LIST "open file input" to the response "To open a file for input use. . . . " Now, with these definitions, we would want the question "What is a file?" to elicit the first answer, the question "How do I open a file?" to elicit the second answer, the question "How do I open a file for input?" to elicit the third. For HELP to do this, another mechanism is required, one which can decide which KEY WORD LIST to extract from the set of KEY WORDS in the question. Without this additional mechanism, we would encounter a problem. For even though the word "file" is present in all three of the above questions and this word is a KEY WORD LIST itself, we obviously do not want the description of a file to be generated in response to the second two questions. These two questions are specific enough to preclude that answer.

HELP decides which KEY WORD LIST to use by the following mechanism. The set of KEY WORDS in the question is searched to find the *longest* KEY WORD LIST, and the message associated with this KEY WORD LIST is used as the response. This operation allows HELP to give the answers described above. Assuming that the KEY WORD LIST of zero length is always defined, and is linked to an "I don't know" message, the only time we cannot find a longest KEY WORD LIST in the set of KEY WORDS is when we have two or more KEY WORD LISTS of maximal length. In this case, we generate the responses associated with *all* of the lists with this property. To continue our example, assume the KEY WORD LIST "close file" is also defined, and linked to the response "To close a file use. . . . " Now let us see what happens with the question "How does one open and close a file?" The set of KEY WORDS taken from the question is "open, close, file". We first see if there is a defined KEY WORD LIST of length 3 (the order of the original set). In this case, there is not. We will then find a KEY WORD LIST of length 2, say, "open file", and its response will be generated. We then find the other KEY WORD LIST of length 2: "close file", and its response is also generated. Now, since no other KEY WORD LISTS of length 2 exist, and we have found at least one of this length, we stop searching and consider the answering phase complete. Notice that if the question was "How do I open or close?" HELP would have output "I don't know", since out of the set "open, close" the only defined KEY WORD LIST is the default one of length zero.

Even though the above mechanisms are uncomplicated and make no use of word order, they allow HELP to answer questions with great accuracy, and little redundancy. The assumption that a longer list of KEY WORDS in a question (i.e., more modifiers), implies that a more specific answer is required seems to be quite adequate in determining which answer is desired. For an example of how a user of HELP can go from the general question to the more specific, see Figure 1.

Also shown in this figure is a facility in HELP which will be described in greater detail below. It is the idea of a "text subroutine", and it both aids the writer of a data base and reduces the size of the HELP program. With this facility, answers to less specific questions can be built up out of answers to more specific ones. This is accomplished by having a response "call" another body of text, in much the same way as standard computer languages do. This means that body of text can exist just once, but can be used by many different answers.

The details of the mechanisms by which HELP attempts to answer a question are described below.

GENERATING ANSWERS

We first read in the question and partition it into words. A word, in this case, is defined to be a sequence of non-blank characters. We then look up each word in a hash table. If the word is found in this table (i.e., it it is a KEY WORD), we place its index in the table into a temporary buffer. If the word is not found, we perform some simple spelling transformations (e.g., saves→ save, going→go, file.→file, etc.), and check the hash table again. If the word is still not found, it is completely disregarded.

After the entire question has been reduced, the resulting set of numbers is sorted by value. If any of these numbers are duplicated in the list, all of the repetitions are removed, so that we get a strictly increasing ordered set of numbers. We now present this set to a data structure called the ANSWER

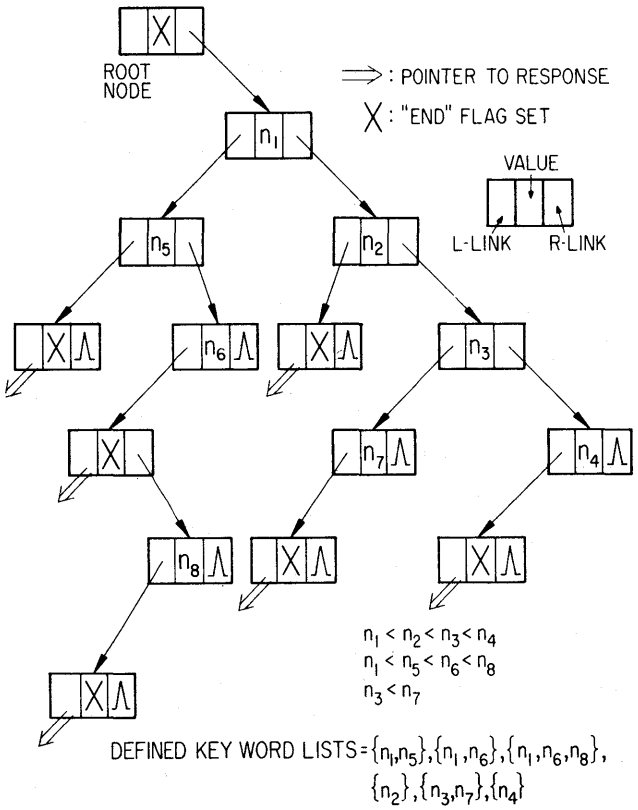


Figure 2—Answer lists

LISTS, which contains all of the defined KEY WORD LISTS and pointers to their associated responses.

The ANSWER LISTS structure is a binary tree, with each node consisting of three fields; a LEFT LINK, a RIGHT LINK, and a VALUE field. The VALUE field can contain either an index into the KEY WORD hash table, or a flag indicating that this node is an "end of list" node. The RIGHT LINK of a node is either null, or points to another node whose value field is greater than itself. The LEFT LINK of a node points to either a node whose value field is greater than itself, a node whose VALUE field has the "end" flag set, or a response (in the text storage table) (see Figure 2).

A KEY WORD LIST and the pointer to its associated response exists in this structure as follows. If there are n members of the KEY WORD LIST, there are $n+1$ nodes in the tree which describe it; one for each of the KEY WORDS, and one for the "end of list" node. Each of these nodes exists on a different *level* of the tree, where level has the following recursive definition. Level 1 is defined to be the set of those nodes which can be reached from the root node by following RIGHT links. For $m>0$, level $m+1$ is the

Definitions

KEY WORD LIST	RESPONSE
FILE	M1: A FILE IS A COLLECTION OF DATA.
OPEN INPUT FILE	M2: USE BRS 15 TO OPEN A FILE FOR INPUT.
OPEN OUTPUT FILE	M3: USE BRS 16 TO OPEN A FILE FOR OUTPUT.
OPEN FILE	M4: [M2] [M3] (WHERE [M] MEANS A "CALL" ON MESSAGE M).
CLOSE FILE	M5: USE BRS 17 TO CLOSE A FILE.
BRS 15	M6: BRS 15 OPEN FILE FOR INPUT. [M8]
BRS 16	M7: BRS 16 OPEN FILE FOR OUTPUT. [M8]
	M8: A=CONTROL WORD, X=DUAL FILE NUMBER.
DUAL FILE NUMBER	M9: A DUAL FILE NUMBER HAS THE COMMAND INPUT FILE IN THE LOWER 12 BITS AND THE COMMAND OUTPUT FILE IN THE TOP 12 BITS.

Question Phase

```
?WHAT IS A FILE?
A FILE IS A COLLECTION OF DATA.

?HOW CAN I OPEN A FILE?
USE BRS 15 TO OPEN A FILE FOR INPUT.  USE BRS 16 TO OPEN A FILE FOR
OUTPUT.

?BRS 15?
BRS 15 OPEN FILE FOR INPUT.  A=CONTROL WORD, X=DUAL FILE NUMBER.

?TELL ME ABOUT A DUAL FILE NUMBER?
A DUAL FILE NUMBER HAS THE COMMAND INPUT FILE IN THE LOWER 12 BITS AND
THE COMMAND OUTPUT FILE IN THE TOP 12 BITS.
```

Figure 1—Definitions and output

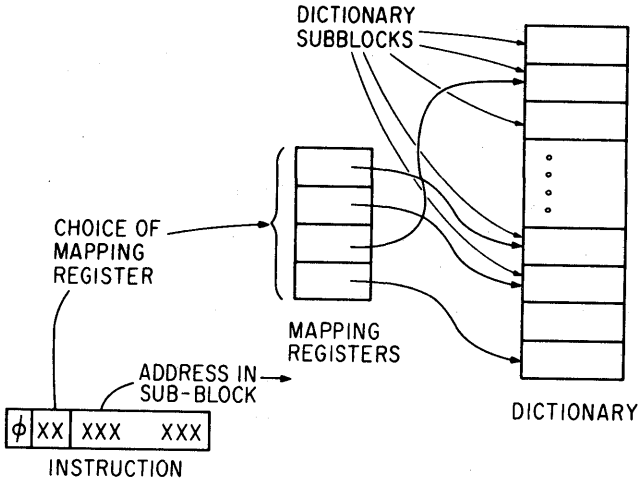


Figure 3—Dictionary addressing

set of nodes which (a) are pointed to by LEFT links of level m nodes and, (b) which can be reached by following RIGHT links from the nodes in (a). In other words, the first node in every KEY WORD LIST is in level 1, the second node is in level 2, etc. Now, for the KEY WORD LIST of n members, the first member is described by some node in level 1. The LEFT link of this node points to a node in level 2, and the second member of the KEY WORD LIST is in this set of level 2 nodes (i.e., start from this node and follow RIGHT links until the desired node is found). The LEFT link of this node just found will point to a node in level 3, etc. After descending n levels in this manner, the last node of the KEY WORD LIST will be encountered. The LEFT link of this node will point to a node with the "end" flag set, and the LEFT link of this node will point to the response associated with the KEY WORD LIST.

Since the nodes throughout the tree are ordered by value, the algorithm for deciding if a list of KEY WORDS is a defined KEY WORD LIST is quite simple, and requires very little time to compute. A failure exit is caused if and only if either a null pointer is encountered when traversing RIGHT links, or an "end" node is not encountered when the entry list is exhausted.

If the KEY WORD LIST constructed from the question is, in fact, a pre-defined list, we will find it in the above structure and will reach an "end of list" node. This node contains an identification of the appropriate response to be generated. In this case, the question has been answered successfully, so we go back to listen for another question. However, let us assume that the KEY WORD LIST we generated from the question does not exist in the ANSWER LISTS structure. We then search for pre-defined lists among the subsets of the original list, beginning with the maximal proper ones. If at any point during this search we find a pre-defined list, we ouptut the response associated with it and with all such lists of the same order, and terminate the answering phase as above. In the event that no subset of the given list exists in the ANSWER LISTS, a standard "I don't know" response is generated, and we return again to listen for the next question.

TEXT STORAGE

Since the size of the HELP program was a consideration, we elected not to store the text of the responses literally. Instead, we utilize a dictionary and have the response which is stored in HELP be a sequence of calls on that dictionary. Our dictionary can contain a maximun of 2048 (text) words, while the address field of the dictionary reference instruction only allows for specifying one of 256 words. Therefore to allow any word in the dictionary to be referenced, we separate the instruction operand address into two fields. One of these fields specifies one of four "mapping registers," while the other field designates one of 64 words in a dictionary sub-block (see Figure 3). The four mapping registers are given initial values at the start of the text output phase, and instructions exist to change their contents when necessary. Experience with several HELP programs has indicated that the map change instructions account for only 3 percent of the total number of instructions in the text, so this device significantly reduces the size of the transformed text.

In addition, we introduced the subroutine facility to allow a body of text to be associated with two or more responses, with virtually no increase in storage space. Consequently, the internal structure of the text which in output in response to a question is not a string of characters, but a sequence of instructions in a simple language (see Figure 4). As can be seen from this figure, these instructions can be divided into several classes (i.e., use word n from the dictionary, output a single character, end the message, call another body of text as a subroutine, etc.). We have found that transforming the text in this manner also considerably reduces the space needed to store the responses. The SDS 930, for which this program was written, has a user address space of 16K 24 bit words. The entire HELP program resides in this space and can contain the equivalent of from 30 to 40 printed pages of text, enough to entirely describe the user interface of our time sharing system.

ADDITIONAL STRUCTURE

Since natural languages contain synonyms, a method of defining equivalences between words was built into HELP. This facility allows us to equate not only one word with another, but also sets of words with each other. When we say equate, we mean that HELP will respond with the same answer no matter which of the words of a synonym equivalence class is used. For example, we might cause "run" and "execute" to be synonyms, so that the two questions "how do I run a program?" and "how do I execute my program?" will elicit the same response. As another example, suppose we want the key word "help" to cause an explanation message to be generated. In addition, we want any one of the words from the set: "exit, finished, goodbye, stop" to cause HELP to return to the TSS executive. As above, the KEY WORD LIST consisting of "help" would point to the desired message, and the words "finished", "goodbye", and "stop" would be equated to "exit". "exit" would, in turn, indicate to HELP to stop its execution (see below).

TAG	FIELD	MEANING
0ZZ	XXX XXX	Dictionary entry preceeded by a blank. ZZ = map to be used. Map contains high order 5 bits of dictionary address, XXXXXX are low order 6 bits.
100	AXX XXX	Alpha. character or multiple blanks. A=1 ==> preceed by blank. X>5: X+33b=character. X<6: X=# of blanks.
101	AXX XXX	ASCII character or control. A=1 ==> preceed by blank. 0<=X<=15: X=character. 16<=X<=22: X+10=character. 23<=X<=27: X+36=character. X=28: Text subroutine return. X=29: End partial message. X=30: suppress blank before next entry. X=31: CR, LF.
110	AXX XXX	Digit or common 2&3 letter words. A=1 ==> preceed by blank. 0<=X<=15: MOPTBL+X=WORD address. 16<=X<=25: X=digit. 26<=X<=31: MOPTBL+X-10=WORD address
111	PQR XXX	Control X=1: Text transfer. New address=(next entry (9 bits)).PQR X=2: Text subroutine call X=4: Change map QR to field of next entry X=6: Subroutine call on undefined text

Figure 4—Format of 9-bit instructions for text storage

A problem has now been created. If a user asks the very natural question "How do I stop HELP?" he will receive the answer about how to use HELP in addition to terminating the program. Presumably this is not what he wanted. The solution to this problem is to define the KEY WORD LIST "exit help" to be a pseudo-synonym of the KEY WORD LIST "exit." Now, when the same question is asked, "stop" will be reduced to "exit," "exit help" will be reduced to "exit," "exit" will terminate HELP, and no message will be generated. In the sme way, many multi-word KEY WORD LISTS can be equated to one another, with the resulting desired reduction.

The synonym facility is implemented by using the same ANSWER LISTS structure described above. In the case of a synonym, the terminal node of a KEY WORD LIST path in the tree indicates that the list in question is equated to a certain node in the tree, and does not point to a message.

There exists another piece of machinery in HELP which has been quite useful. All of the responses which can be pointed to by KEY WORD LISTS do not have text associated with them. Some number of them (10 in particular) indicate to HELP to perform some "special action." One special action is to terminate the program. So, as above, saying "goodbye" to HELP will cause the user to exit from HELP. Another special action is to commence execution of another HELP program, and have the ensuing questions directed to it, rather than to the original program. This facility is helpful for two reasons. First, the total amount of information about the entire system is too large for one HELP data base. Even if the capacity of the data base were expanded, problems of ambiguity would arise owing to the context dependency of the questions. Second, since different areas of the system are maintained by different people, it seemed advisable to have the HELP programs also maintained separately, so that one area could be modified without global repercussions. With this mechanism, any HELP program can call any other HELP program. A user therefore has immediate access to all information about our system, with very few contextual problems arising.

As mentioned above, a question can elicit more than one response if it contains more than one KEY WORD LIST of the same length. If a user asks a question of this type, pressing a break key (rubout, in our case) during the output of any of the several messages will stop just that message. Output will then continue with the next message in the sequence. This feature has proved to be quite convenient, especially in the case of long, multi-part answers where the user only wants to see one part of each.

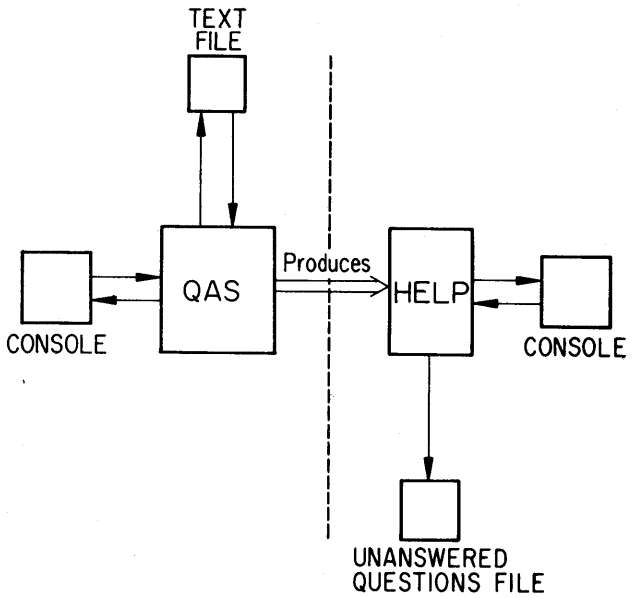


Figure 5—QAS and HELP

COMPACTING DATA

As we have discussed earlier, an important criterion in the design of HELP was to keep its size small. Some of the methods used have already been presented (encoding the text into a sequence of interpreted instructions and mapping the dictionary references). There are also two more which will now be shown. The convention we made concerning the dictionary was that it would contain words of only two or more alphabetic characters. Now, in the SDS 940, all of the alphabetics have an internal representation of 32 or greater. Therefore, by subtracting 32 from the last letter of each word, we compacted the words as densely as possible and were still able to know the locations of the word boundaries. However, this scheme by itself would be quite inefficient, since half of the dictionary, on the average, would have to be counted through to find a word. Accordingly, we have a DICTIONARY ADDRESS TABLE of 64 entries, each of which points to the start of a 32 (text) word block. To locate a word in the dictionary, the high order 6 bits of the dictionary address are used to select one of the entries in the DICTIONARY ADDRESS TABLE. Starting from this location in the dictionary, the nth word we encounter is the word we want, where n is the lower order 5 bits of the dictionary address. In this manner, the dictionary is as compact as possible, and the time to find a word is not astronomical.

The second method of reducing storage in HELP concerns the KEY WORD hash table. The algorithm which we use to construct a hash code from a word in the question guarantees that all words of less than 6 letters will transform uniquely (we transform the word into base 26 representation). Also, since the hash code is 24 bits long, while the hash table has room for only $2 \uparrow 10$ entries, the probability that two arbitrary words will have the same hash code is quite small. We therefore do not check to make sure that a word whose hash code we find in the table is, in fact, the word we want. We make the assumption that if we find a word's hash code, we have found the word. This obviously reduces storage, since we do not have to retain the words themselves.

This last mechanism might seem strange, but we felt that since this was only a question answering facility and not, say, an assembler, we could exist with a small amount of inaccuracy. Our experience has shown that errors due to recognizing the wrong word almost never occur, and that when they do, only cause extra answers to be generated (due to recognizing an undefined word as a KEY WORD).

CREATING A HELP PROGRAM

We now describe the machinery by which a user may create a question answering (HELP) program. Each such program contains the code for execution and the data base, unique to it, which indicates KEY WORDS, text to be output, and the relationships between KEY WORD LISTS and that text. The means of defining these objects (i.e., creating a data base) is another program, denoted QAS, for Question Answering System. In QAS, a user can define KEY WORDS and KEY WORD LISTS, can define responses to be typed out, can associate these responses with KEY WORD LISTS, and can create his particular HELP program. He can also discover what objects have already been defined, what synonym relationships exist, the size of the various internal tables, etc. In other words, QAS is designed to allow a person to interactively construct a HELP data base, defining and redefining objects as he sees fit (see Figure 5). A brief description of some of the operations which can be performed in QAS is given below.

1. Define a KEY WORD LIST and the named response which is associated with it.
2. Define a named body of text.
3. Define a KEY WORD LIST equivalence class.
4. Define a KEY WORD LIST which will take one of the "special actions" described above.
5. Edit a named response.

6. Ask for the HELP program associated with QAS, to ask questions about QAS.
7. Investigate the size of various tables.
8. Redefine a KEY WORD LIST.
9. Given a word, determine which KEY WORD LISTS it is a member of.
10. Given a word, determine what synonym relationships exist between it and any other words.
11. Given a list of words, determine which of its subsets are KEY WORD LISTS.
12. Given the name of a message, determine which KEY WORD LISTS point to it.
13. Take the input from a file, instead of from the console.
14. Write the dictionary on a file.
15. Create the HELP program.

As indicated above, each body of text given to QAS must have a name attached. The pupose of this name is to allow for the subroutine facility described previously. The inclusion in a body of text of the name of another body of text will cause the second body of text to be inserted during message output. This second body of text can, in turn, "call" another, etc. An example of the input given to QAS can be seen in Figure 6.

From Figure 6 we see that the structure of the input presented to QAS is in the form of "commands", followed by the arguments for the command. For example, the command, ANSWERS, has as its arguments a KEY WORD LIST, a text name, and a body of text. This command will, after receiving the arguments, define the KEY WORD LIST, compile the body of text into its internal form, associate the name given with the body of text, and cause the newly-defined KEY WORD LIST to point to this text. Another command, KILL KW LIST, will erase the pointer from the given list to a response or a synonym. This allows us to redefine a list if the original definition was faulty.

Figure 6 also shows the power of the subroutine facility. We can define the answers to specific questions by giving the text to be output. We can then build up the responses to the more general questions by utilizing calls on the more specific text.

QAS has proved to be a very powerful tool for the creation of a HELP program. Using it, a person can construct a preliminary HELP for, say, our test editor in a few hours. Then, after other users have tried it and asked questions which the writer did not anticipate, QAS can be used again to modify the data base. To facilitate this procedure, the writer can tell QAS, during the creation of HELP, to have HELP write on a file all the questions it cannot answer. The writer can then

```
SYNONYMS.

INPUT
READ

NUMBER
NO.

NUMBER
NO

OUTPUT
WRITE

ANSWERS.

FILE
[FILE]
A FILE IS A COLLECTION OF DATA.

OPEN INPUT FILE
[OIF]
BRS 15 IS USED TO OPEN A FILE FOR INPUT.

OPEN OUTPUT FILE
[OOF]
BRS 16 IS USED TO OPEN A FILE FOR OUTPUT.

OPEN FILE
[OF]
[OIF] [OOF]

INPUT NUMBER
[INNUM]
BRS 36 IS SUED TO INPUT A NUMBER.
```

Figure 6—Format of input to QAS

```
*HELP.
TERMINATE QUESTIONS WITH A '?'.  THIS VERSION OF HELP WILL ANSWER
QUESTIONS ABOUT MATERIAL IN R-21.  TO LEARN MORE ABOUT THE USE OF
HELP, TYPE "HELP?".  (11/27/68)

?HOW CAN I READ THE LENGTH OF A FILE?
USE BRS 143 WITH ATTRIBUTE NUMBER 2 TO READ THE LENGTH OF A FILE.

?TELL ME ABOUT BRS 143?
BRS 143:  READ STATUS.  THE CALLING SEQUENCE IS:
A:  TABLE ADDRESS OR DATA (DEPENDS ON BØ).  IF A TABLE ADDRESS, A IS
    INCREMENTED TO POINT TO ONE PAST THE LAST WORD TRANSFERRED.
X:  THINGY NUMBER
B:  DECODES AS FOLLOWS
    BIT Ø:  Ø IF A HAS DATA, 1 IF A POINTS TO A TABLE
    BITS 1-11:  'TYPE' OF THINGY
    BITS 12-23:  'ATTRIBUTE'
'TYPE' IS 1 FOR A DRUM FILE, 2 FOR A JOB, FOR DETAILS SEE PAGE 9-2
IN R-21.

?WHAT IS A THINGY?
A "THINGY", AS USED IN BRS 143 AND 144, IS EITHER A FILE OR A JOB.

?HOW CAN I OUTPUT A STRING?
USE BRS 35 TO OUTPUT A STRING.

?MORE ON BRS 35?
BRS 35:  OUTPUT STRING.  X=FILE NUMBER, AB=STRING POINTER.

?BRS 2 3?
BRS 2:  EXEC ONLY  CLOSE SPECIFIED FILE
        A=FILE NUMBER.  NON-EXEC IS BRS 2Ø.

BRS 3 DOES NOT EXIST

?
```

Figure 7—A session with HELP

investigate this file occasionally to discover what he has overlooked.

It is also possible to have HELP write on a file all of the questions which it receives. In this way, we can determine how much use a particular HELP system is getting, and how well it is doing. Our experience at Project Genie is that our efforts in designing a simple question answering facility have been successful. HELP receives a great deal of use, generates useful answers, and, using QAS, can easily be updated to allow for modifications and oversights. Figure 7 gives an example of a short session with HELP.

CONCLUDING COMMENTS

The algorithm which we use to find KEY WORD LISTS from the set of KEY WORDS in a question has one drawback. If the number of KEY WORDS in the question is large, while the length of the longest KEY WORD LIST in the set is small, the searching time is very long. This is due to the fact that, starting from the original set, we check every subset. We have never experienced a problem of this sort, since most of the questions presented to our HELP systems have three or less KEY WORDS. However, it is a possibility. To solve it, a test would have to be made before starting to check the subsets of a given order, to insure that the number of these subsets is less than some pre-determined maximum. If not, all of those subsets would not be tested. The calculations involved are trivial, and could be incorporated if necessary.

ACKNOWLEDGMENTS

The author would like to acknowledge the help of both Bret Huggins and Butler Lampson. Mr. Lampson designed the initial structures of the ANSWER LISTS and the dictionary, and Mr. Huggins implemented them. In addition, both contributed many hours of their time to discussions during all phases of the design of HELP and QAS.

REFERENCE

1 C S CARR
HELP—An on line computer information system
Project Genie Document No P-4 January 19 1966

CypherText: An extensible composing and typesetting language

by C. G. MOORE and R. P. MANN

The Cyphernetics Corporation
Ann Arbor, Michigan

INTRODUCTION

CypherText is a programming language designed for text formatting and typesetting in a time-sharing environment. Text to be formatted or typeset is input on a terminal and may be output at the terminal or on various typesetting machines.

Although a number of computer typesetting languages have been written for particular applications (such as newspaper work), few of these languages are adaptable for any other application.[1,2,3] This inflexibility has remained one of the most serious limitations of existing computer typesetting languages.

CypherText, an extensible language, overcomes this problem of inflexibility to a great extent. Because CypherText is truly extensible, it is possible to tailor specific formatting capabilities to meet the needs of particular typesetting applications by predefining formats for each application. Both large scale projects such as catalogs and parts lists, as well as smaller operations, such as job-shop typesetting, may now be accommodated within the scope of one language.

By predefining formats, a set of format definitions for a specific application may be "packaged" so that the definitions come "ready to use," i.e., the user does not have to know anything about how to make up formatting definitions for himself. This "packaging" of formats has already been accomplished for architectural specifications, technical report writing, and job-shop typesetting applications. In the first two cases, the format definitions are so comprehensive that the user almost never requires any of the unextended features of the language. In fact, most users are unaware of the "unpackaged" features because the packaged definitions meet all their formatting requirements.

In addition to providing wide formatting flexibility, CypherText also provides flexibility in choosing typesetting devices on which the text is to be output. Other typesetting languages have typically been geared to one or a few specific typesetting machines. CypherText, on the other hand, is "device independent": a "post-processing" feature allows users to set their text on many commercially available typesetting devices, including photocomposition devices, "hot lead" devices, and even typewriter-like terminals, with no change required in the input text.

The extensibility of CypherText and the flexibility it offers derive from the structure of the language and the method of its use.

THE STRUCTURE OF CYPHERTEXT

The major structural features of CypherText are its command syntax, command definition capability, and string storage capability.

Syntax

One prerequisite of an extensible typesetting language is an unambiguous syntax. Every effort has been made to keep the CypherText syntax simple and consistent.

CypherText input consists of the text to be typeset and the formatting instructions for the text. The formatting instructions ("commands") are distinguished from the text by "command break characters." Though the command break character may be any character the user chooses, throughout this paper the slash (/) will be used. The following fragment of input shows some text and one command:

as shown on the following page/NEXTPAGE/

In the above example, the text "as shown on the following page" would be set on a particular page, after which the command "NEXTPAGE" would cause any subsequent text to be set on the next page.

More than one command may be placed within the

break characters, provided that the individual commands are separated by semicolons, as in the following example:

as shown on the following page./NEXTPAGE;

CENTER/Chapter VI

In this example, the two commands "NEXTPAGE" and "CENTER" are placed within the same set of slashes; "NEXTPAGE" causes a skip to the next page, after which "CENTER" causes the text "Chapter VI" to be centered at the top of the new page.

Some commands require one or more modifiers (parameters) to fulfill their formatting functions. In these commands, the parameters are separated from the name of the command by a space, and multiple parameters are separated from each other by commas. For example, the command "SPACE" requires as a modifier the amount of vertical space to be left on a page, expressed in points. Thus, the command

/SPACE 24/

causes a vertical spacing of 24 points.

Among the commands requiring multiple parameters is "NEWTYPE", which has as modifiers the name, style, and point size of the type face to be set. Thus,

/NEWTYPE TIMES ROMAN,8/

would cause a switch to 8 point Times Roman as the current type face.

A list of the most commonly used CypherText commands and their functions is provided in Table I.

Command definition

The capability of defining new commands is integral to the extensibility of CypherText and contributes greatly to its ease of use.

New commands are created by combining a number of basic commands and assigning a name to the combination. The name is assigned by means of the "DEFINE" command. In the following example, a new command called "PT", requiring one parameter, "LINES", has been defined. The definition of the command appears between the quotation marks, and consists of three basic commands: SKIPIF, NEXTPAGE, and ENDIF:

/DEFINE PT(LINES),

"SKIPIF<,72, LINES *12;

NEXTPAGE;ENDIF"/

Having defined the new command "PT(LINES)", it would be used by supplying a value for the parameter

TABLE I—Commands

DEFINE
Used to define a new CypherText command. It gives a name to the command and indicates how the parameters are to be used.

ENDIF
(See SKIPIF)

EVALUATE
Evaluates an arithmetic expression and stores the value in a specified string name.

INCLUDE
Requests that the contents of some string (or combination of strings) be set as text at this point.

LEADER
Requests that a 'leader' of some particular character be used to fill out the current line of text. Used mostly in tables.

MAP
Gives a character whose occurrence is to be 'mapped' into some string. Every subsequent appearance of the mapped character will be treated as though the string of characters it is mapped into had occurred instead.

NEXTPAGE
NEXTPARAGRAPH
NEXTFIELD
Cause a new page, paragraph, or field (respectively) to be started at this point.

OUTPUT
Specifies the output device to be used for setting the text (for example, LINOFILM, PHOTON 713, terminal, etc.)

PUSH
POP
Together, allow the current contents of some string to be saved, and then later recovered.

SET
Assigns a new value to some string. Corresponds to the use of the equal sign (=) or replacement operator in most programming languages.

SKIPIF
Allows commands and text to be skipped (or ignored) in the setting process, if a specified condition is met. No text will be set or commands processed until an ENDIF command is encountered.

SPACE
Leaves a vertical space of the specified amount, on the page currently being composed, at the point the command occurs.

USE
Gives the name(s) of one or more files whose contents are to be included as input at this point. These files may include commands or text or both.

"LINES". For example, the command

/PT 15/

would cause any text following the command to be set on the current page if more than 15 lines remain, or to be set on the next page if less than 15 lines remain.

String storage

The capability of assigning names to strings of characters and of storing the strings for future use is also crucial to the extensibility of CypherText. Strings are assigned names and stored for retrieval by means of the "SET" command. For example, the command

/SET X, "NEXTPAGE"/

would store the 8-character string "NEXTPAGE" under the name "X".

Such stored strings may be used as commands, parameters to commands, or even as text to be set. For example, after the above command, the command "X" is equivalent to the command "NEXTPAGE."

Using stored strings as commands or as parameters to commands merely involves substituting the string name for the string. For example, the sequence

/SET LINE, "12"/

.
.
.

/SPACE LINE/

stores the value "12" under the name "LINE", so that when "LINE" is used as a parameter to the "SPACE" command, a vertical spacing of 12 points is left on the page.

Using stored strings as text to be set involves the use of the "abbreviation character." Though this character may be any that the user chooses, in the following example the "at sign" (@) has been used.

Commonly used words, phrases, or paragraphs may be assigned string names and stored; whenever the words, phrases, or paragraphs are to be used as text, only the string name need be used, preceded by the abbreviation character. For example, the command

SET CT, "CypherText,

an extensible language,"

would store the quoted text under the name "CT". Whenever the user wants to include the text "CypherText, an extensible language," he has only to type in "@CT". In this case, 35 characters have been reduced to the 2-character abbreviation "CT".

Stored strings may also be used in a way analogous to the use of variables in algebraic programming languages. Thus, stored strings may be used in an arithmetic expression, as a parameter to a command. For example, the sequence

/SET LINES, "12"/

.
.
.

/SPACE 5*LINES/

causes a vertical space of 5 times the number of points specified in the string named "LINES" to be left on the current page.

Reserved String Names

Many of the formatting functions of CypherText are controlled by the use of "reserved string names". These are string names whose contents are constantly monitored by CypherText. Whenever the value of one of these reserved strings is changed, CypherText takes some special action. For example, the reserved string variable "LINELEADING" indicates the amount of space to be left between each line of the final text. Changing the value of this string will change the amount of space left between lines. Thus, the command

/SET LINELEADING,"12"/

indicates that from this point on, 12 points of space are to be left between each line in the output text. For typewriter-like terminals, this command is effectively a double-space command. As another example, the formatting of the top and bottom of each page is controlled by two reserved string names, "HEADER" and "TRAILER". Any combination of commands and text may be stored in these strings. Whenever CypherText begins a new page of text, it examines the contents of these strings to determine what to place at the top and bottom of each page. For example, the command

/SET HEADER,"/SPACE 36;CENTER;INCLUDE

TTEXT; SPACE 36/"/

stores in the reserved string "HEADER" a set of commands which will center at the top of each page the current contents of the string named "TTEXT", with 36 points of space between this line and the top of the page, and 36 points of space between this line and the first line of text. Of course, the contents of the string "TTEXT" may be changed at any time, via the "SET" command. Thereafter, each page will have the new contents of "TTEXT" as a centered title.

TABLE II—Reserved Variables

FIELD
Controls the number, width, and placement of columns on the page. Also controls the placement of text within the field: centered, justified, flush right, flush left.

HEADER
Controls the formatting of the top of each page. Title, if any, spacing, and so forth.

HYPHENATION
Controls automatic hyphenation, which is done only if the contents of this string name is "ON".

INDENT
Specifies the amount of the indentation at the beginning of each paragraph.

JUSTIFICATION
Controls the amount of 'filling' with spaces allowed to justify a line of text.

LINELEADING
Controls the amount of space to be left between lines.

PAGEHEIGHT
Controls the height of each page.

PAGEWIDTH
Controls the width of each page.

PARAGRAPHLEADING
Controls the space to be left between each paragraph.

TRAILER
Controls the formatting of the bottom of each page, as with HEADER, at the top.

TYPEFACE
Controls the current type face (TIMES, BODONI, etc.).

TYPESIZE
Controls the current type size.

TYPESTYLE
Controls the current type style (ITALIC, BOLD, etc.).

A list of the most commonly used reserved string names and their functions is given in Table II.

Defined String Names

"Defined string names" is another class of string names which has special meaning to CypherText. These are strings which the user may always assume to contain some particular piece of information. Whenever the user references one of the defined string names, CypherText determines the current value of that piece of information and supplies that value as the value of the string. For example, the defined string name "TOTALPAGES" always contains the number of pages set so far during a particular run. The value stored in "TOTALPAGES" may be conveniently used to set a page number on each page.

Many of the defined string names are used primarily for testing certain conditions. The defined string name PAGELEFT contains the number of points (vertically) left on the current page, before it will be necessary to start a new page. Before beginning the setting of a table in his input text, a user may embed a conditional skip command ("SKIPIF") in his input which will test PAGELEFT to determine if there is enough room on the current page for the entire table. If there is not enough room, CypherText will start a new page; if there is enough room, the table will be set on the current page.

A list of the most commonly used defined string names and their functions is given in Table III.

Many other features of CypherText, such as automatic justification and hyphenation, are not discussed here because they are available in other languages as well.[4,5,6] The primary emphasis here has been to illustrate the extensible features of CypherText, particularly those features which differentiate it from other typesetting languages.

USING CYPHERTEXT

Using CypherText to transform original copy into finished text is a five step process:

1. Embedding
2. Inputting
3. Proofing
4. Postprocessing
5. Typesetting

Embedding is the insertion of CypherText commands into the original text. The commands may be written in by an editor for later inputting by a typist, or, in the case of experienced users, the commands may be embedded extemporaneously as the text is being input. The following example shows an original manuscript with the commands embedded by an editor:

```
/center/
/newtype HELVETICA, BOLD, 12/
CYPHERTEXT: A DEMONSTRATION
/space 12; newtype TIMES, ROMAN, 10; text/
#CypherText enables you to transform unformatted rough copy
into finished text by embedding CypherText commands in the
rough copy.
#The CypherText commands provide for all the formatting
requirements of the printed page, including justification,
hyphenation, tabulation, leadering, and runarounds.
```

In this example, the commands specify that the heading is to be centered and set in 12 point Helvetica Bold, while the two paragraphs are to be set in 10 point Times Roman. (Note that the command for starting a new paragraph has been abbreviated to "#".)

After the commands have been inserted, the "embedded copy" is input into a general purpose time-sharing system on virtually any terminal input device.

Several advantages derive from the fact that the copy is entered into a time-sharing environment: first, the copy may be stored on any of a number of direct-access devices, depending on factors of economy and convenience; second, output from other programs in the time-sharing system may serve as input to CypherText; and third, the copy is always immediately accessible for updating.

The following example shows how the embedded copy would appear on a terminal during inputting:

```
/center/
/NEWTYPE HELVETICA,BOLD ;12/CYPHERTEXT%: A DEMONSTRATION
/space 12;NEWTYPE TIMES,ROMAN,10;TEXT/
#CypherText enables you to transform unformatted
 rough copy into finished text by embedding
 CypherText commands in the rough copy.
#The CypherText commands provide for all the
 formatting requirements of the printed page,
 including justification, hyphenation, tabulation,
 leadering, and runarounds.
READY
```

After the copy has been input, immediate proofs may be obtained by having the system compose and print out the text at the terminal. Of course, the proof copy takes on the limitations of the terminal on which it is output. However, for many applications, the proof copy obtained at the terminal is satisfactory enough to serve as final output for reproduction by printing or other means. For these applications, where limited type variety and non-proportional spacing are of no concern, the proof copy is the end product of the CypherText process and the last two steps, postprocessing and typesetting, are omitted.

TABLE III—Defined Variables

DATE
Current data, in the form DAY-MONTH-YEAR.
LINECHARACTERS
Current number of characters set so far on this line.
PAGEDOWN
How much text has been set on this page, i.e., how far 'down' the page text has been set.
PAGELEFT
How much space is left on the page, vertically, before it will be full.
PAGELINES
How many lines have been set on the page currently being composed.
TIME
Current time of day, in the form HH:MM (24-hour time).
TOTALPAGES
Total number of pages set so far in this run.

Whether or not the proof copy is the final output, proof copy is useful for checking the formatting and for catching typographical errors. If errors are found, or if the formatting is to be changed, it is a simple matter to edit the input copy by using any of the text-editing facilities of the time-sharing system. After the copy is edited, further proofs may be output until the user is satisfied that the text is composed as desired.

The following example shows proof copy obtained at a model 37 teletype terminal:

```
CYPHERTEXT: A DEMONSTRATION
    CypherText enables you to transform unformatted
rough copy into finished text by embedding CypherText
commands in the rough copy.
    The CypherText commands provide for all the
formatting requirements of the printed page, including justi-
fication, hyphenation, tabulation, leadering, and runarounds.
```

For those applications where a variety of type faces and proportional spacing are important, the next step is to postprocess the input copy for setting on a particular typesetting device. Postprocessing is handled automatically by the system, producing a tape (paper or magnetic) to drive any of the most commonly used typesetting machines.

To achieve complete typesetting flexibility, the CypherText language has been made as "device independent" as possible. This independence has been achieved by defining the input language independently of the characteristics of any specific typesetting device; the output is targeted to an idealized typesetting device (which does not actually exist). Producing output for an actual typesetting device is the function of the post-processing program, which translates the device independent output to the particularities of the desired typesetting machine. Translating the copy for typesetting on different machines requires only a change in the "OUTPUT" command, which takes one parameter, the name of the desired typesetting machine. The "OUTPUT" command is also used for obtaining drafts at the terminal, typewriter-like devices being considered a special kind of typesetting machine. The device independent translators generally run in parallel with the CypherText language itself, as co-routines, effectively making the entire process a one-pass operation.

The final step, typesetting, consists of running the postprocessed tape on a particular typesetting machine to obtain finished, typeset copy. The number of type

faces and sizes, as well as the spacing characteristics, depend, of course, on the typesetting machine itself. The following example shows the sample text set on a Linofilm Quick:

CYPHERTEXT: A DEMONSTRATION

CypherText enables you to transform unformatted rough copy into finished text by embedding CypherText commands in the rough copy.

The CypherText commands provide for all the formatting requirements of the printed page, including justification, hyphenation, tabulation, leadering, and runarounds.

Although CypherText can be used in any composing and typesetting application, it is especially suited for text requiring frequent revision, complicated or repetitive formatting, and high speed and accuracy. Despite the sophisticated capabilities of the language, experience has shown that both novices and trained editors alike can be taught to use CypherText easily and effectively in a broad range of composing and typesetting applications.

APPENDIX A

SYNTAX OF THE LANGUAGE

Normally CyperText operates in "text mode", a mode in which the characters in the input stream are simply set according to whatever current formatting parameters are in effect. Commands which alter the formatting parameters may appear anywhere in the input text stream. These commands are bracketed by the current "command break character", which is normally a slash(/). One or more commands placed between command break characters in this manner is called a "command group", and must follow certain syntactical rules.

The syntax of a command group is given below. Rigor in the formal sense has been sacrificed for readability. Such sacrifices are indicated by enclosing parentheses. In the definitions we use the convention that lower case character strings stand for a generic type. Upper case strings and punctuation characters not mentioned in these conventions must appear as shown. Square brackets surround optional material. Three dots following a syntactic unit indicate that it may be repeated an arbitrary number of times. The sequence ':=' is used to mean 'is defined as'. A vertical bar is used to indicate that one of the options in curly brackets should be chosen. Curly brackets are also used to group syntactic units for some purpose. The special generic name 'nullstring' and 'blankstring' stand for a string of no characters and a string of one or more blank characters, respectively. The generic name 'alphanumericstring' stands for an arbitrary string of upper and lower case letters and numbers. The generic name 'numericstring' stands for a string of digits, possibly with a leading plus or minus sign, and an optional embedded decimal point.

commandgroup := commandbreakcharacter
commandstring
commandbreakcharacter

commandstring := [commandelement;]···
commandelement

commandelement := {primitivecommand |
macrocommand |
stringname | nullstring}

primitivecommand := (one of the commands from Table I) blankstring parameterlist

macrocommand := macroname blankstring
parameterlist

stringname := alphanumericstring (of length less than 64 characters, beginning with a letter, and which has previously appeared as the first parameter of a SET or EVALUATE command).

macroname := alphanumericstring (of length less than 64 characters, beginning with a letter, and which has previously appeared as the first parameter of a DEFINE command).

parameterlist := [{(parameterlist) |
simpleparameter},]···
{(parameterlist) | simpleparameter
| nullstring}

simpleparameter := {alphanumericstring |
stringexpression |
numericexpression}

stringexpression := {stringname | quotedstring}
[& stringexpression]

quotedstring := "alphanumericstring" |
'alphanumericstring'}

numericexpression := {numericstring | stringname}
[arithmeticoperator
numericexpression]

arithmeticoperator := {+|−|*|/}

APPENDIX B

Implementation details

CypherText has been implemented for a PDP–10 time-sharing system. It is written entirely in assembly

language. This choice was dictated by the fact that the only other option available at the time was FORTRAN. FORTRAN was felt to be too awkward and inefficient to use as an implementation language for what is essentially a string-handling program. It should be noted that higher-level languages available on other computers (such as PL/1) would be unquestionably preferable for implementing this type of program.

The programs, both first and second passes, are reentrant. In fact, the PDP–10 system allows the first pass to be shared simultaneously by a number of time-sharing system users. The first pass program occupies about 6500 (36-bit) words of memory for the code. A minimum of 6000 additional words are needed for working storage (page buffers, string storage, etc.).

The size of the second pass programs, which are usually loaded with the first pass for a particular run, varies considerably with the type-setting device selected. All the current second pass programs are less than 2000 words long, including both code and working storage.

The device independence of the PDP–10 input/output support allows input text to be accepted from a variety of media. The same comment applies to system output. No scratch files are written by the system, but CypherText does access several support files in the course of a run, which must be stored on a random access device.

The running time of the program varies with the number and complexity of the commands embedded in the text. For "straight matter", such as non-technical books, running time for first and second passes combined is about .3 seconds per 1000 characters. For very complicated work, such as some parts catalogs, run time may approach 2 seconds per 1000 characters. Unless final copy is being printed at the terminal, additional time will be needed on the type-setting device chosen to set the text.

REFERENCES

1 G M BERNS
Description of FORMAT, a text processing language
Comm of the ACM Vol 12 3 March 1969 pp 141-146

2 *TEXT360: Introduction and reference manual*
Form C35-0002 IBM Technical Publications Dept White Plains N Y March 1969

3 *Harris Composition System: Language manual*
Harris-Intertype Corp Cleveland Ohio March 1970

4 *Textran-2: User's manual*
Form T2-102-3 Alphanumeric Inc Lake Success N Y 1969

5 J W SEYBOLD
The market for computerized typesetting
Printing Industries of America Washington D C 1969

6 *HYPHENATION360: Application description*
Form E20-2130 IBM Technical Publications Dept White Plains N Y 1969

Integration of rapid access disk memories into real-time processors

by R. G. SPENCER

Bell Telephone Laboratories, Incorporated
Naperville, Illinois

INTRODUCTION

In large real-time systems such as telephone, traffic control, and process control, the required amount of high-speed random access memory becomes cost prohibitive. In these types of systems, much of the data stored in memory is accessed infrequently. For such low priority data, a rapid access disk or drum memory controller can be used to advantage; such a controller is described in this paper.

In a real-time processor, the multiprogram environment generally consists of two types of programs. The first type is event-associated and is executed only when a specific event occurs. The second type is a routinely run program. The event-associated programs, typically the high runner programs, use the largest percentage of the processor real time. If the required response time for an event is greater than the average latency of a rotating memory, data pertinent to each event can be stored on a disk file. Upon demand, an autonomous disk controller delivers data to the system.

A program requests a block of data from the controller, and instead of waiting for the disk controller to deliver the data, the program exits to an executive program. The executive program calls another program to be executed. When a disk task dispenser program finds a request for data completed, it returns control to the program requesting the data. With this mode of program flow, many disk requests for data can be operated concurrently. With a large queue of random requests, the disk controller can execute a job at most positions on the disk as it rotates. Thus, the throughput of the disk controller can approach the data rate of the disk.

A disk controller system can serve as a memory for many other types of data. It can store the only copy of low priority or infrequently run programs. It can serve as a backup for main programs normally held in high-speed random access stores. This disk complex is also well-suited as a low-speed data buffering memory for data links or analysis information.

The system

In a real-time system, the processing capacity is usually limited by the amount of real time required to handle a given capacity. To maximize the capabilities of a system using a disk file controller, the operation of the disk as a memory should not require excessive real time. To satisfy this requirement, the controller runs autonomously from a large hardware queue in the controller. It further communicates directly with the random access memory on an interleaved bus cycle with the central processing unit (CPU).

The basic cycle times of the CPU, random access memory, and disk file controller are the same. However, the buses can be used twice every CPU cycle, since the disk file controller's cycle time is staggered one-half of a CPU cycle. During the first half of its cycle, the CPU uses the high-speed memory bus; the disk file controller is capable of using the same bus during the CPU's second half-cycle. The CPU provides a path from the disk file controller to the high-speed memories as shown in Figure 1.

Since the high-speed memory cycle time is twice the bus cycle time, the CPU and disk file controller cannot access the same high-speed memory during the same cycle. To make best use of the capabilities of the disk file controllers, several independent high-speed memories are used on the same bus. Although the controller can communicate with any memory, the desirable program system is designed so that the disk controller communicates most frequently with high-speed memories having low CPU occupancy. Because many independent sources compete for use of the

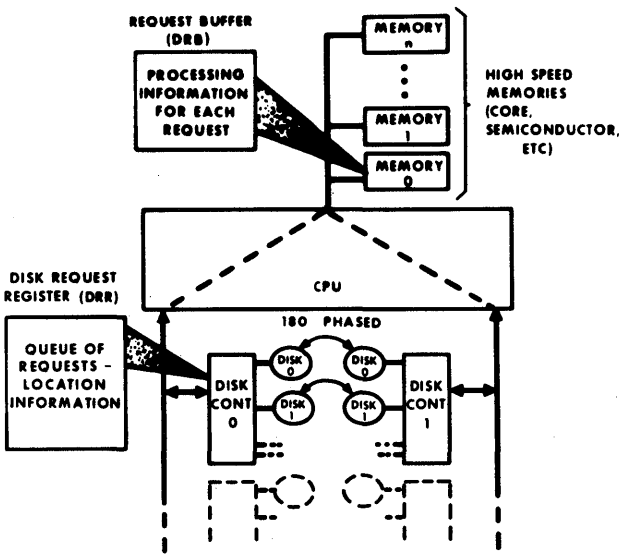


Figure 1—Processor complex

high-speed memories, a priority-blockage circuit is incorporated in the CPU. Once each cycle, this circuit selects the highest priority controller or the CPU. The circuit grants the controller use of the bus, only if the controller accesses a memory different from that accessed by the CPU during its first half-cycle. If the controller requests the same memory, it is blocked until the next cycle, when it tries again. A disk controller can be blocked for a limited number of contiguous cycles. The disk data rate and the CPU cycle time determine this limit. Before the next word overflows from the disk, the controller will be granted the use of the bus.

In the simulation (see *Simulation results*), the system studied could be blocked for three contiguous cycles before the CPU need be interrupted to allow a disk controller access to the memory. In this simulation one word was read from disk every seven CPU cycles. Thus, up to two tracks could be read into the same high-speed memory used by the CPU, without causing the disk to overflow. In an actual operating system, the CPU never occupies one particular memory every cycle for long periods of time (excluding a program memory).

Since the least reliable hardware in a processor is a disk file, it should be duplicated to decrease the downtime of the processor. By phase-locking the disks (rotationally 180 degrees) with duplicate information, the average latency for the pair is half that of a single disk. If the rotation time of a disk is 40 milliseconds, a request submitted to that disk is returned in 20 milliseconds, on the average. If the same request is entered in both duplicate controllers, the first to reach the proper disk location completes the job, in an average of 10 milliseconds. With the duplicated and phase-locked disks each working independently, the average latency is thus reduced. The throughput is doubled, and the system reliability is greatly increased.

As depicted in Figure 1, the system is expandable to meet any requirements. The number of disks per controller is variable as is the number of controllers and high-speed memories per system.

The disk file controller

Within a controller complex, each disk is electronically divided into a large number of pie-shaped sectors. Each disk face is further split into a number of radial tracks. Figure 2 shows a division consisting of 100 sectors and 100 tracks: 10,000 sector-track data records in all. Each data block has a distinct record address. The opposite disk face is similarly segmented into another 10,000 addresses. Addressing to a word within the block is accomplished by specifying a "starting" word within a sector-track. When multiple words are desired, a "number of words" is specified in the request. With this type of address scheme, request sizes range from one word to an entire disk face.

Information concerning a request is placed in two queues as indicated in Figure 1. One entry is made in the disk request register (DRR), located in the controller hardware, specifying location information only

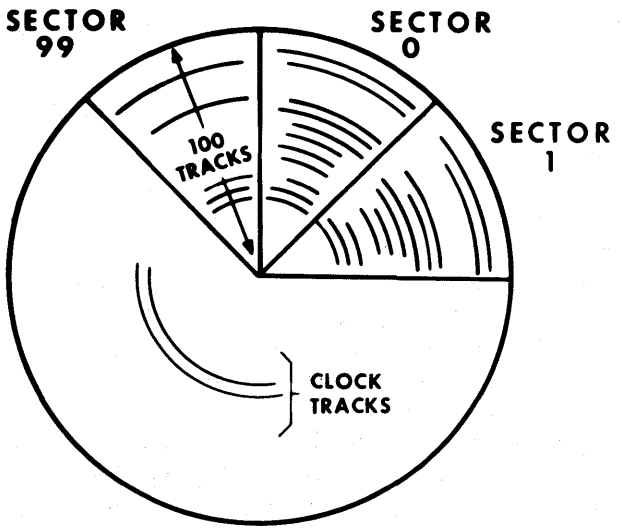


Figure 2—Disk segmentation

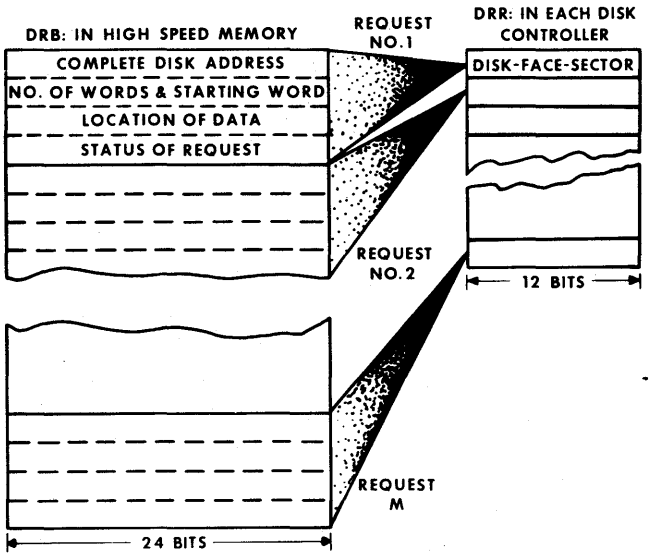


Figure 3—Disk request buffer (DRB) and disk request register (DRR) organization

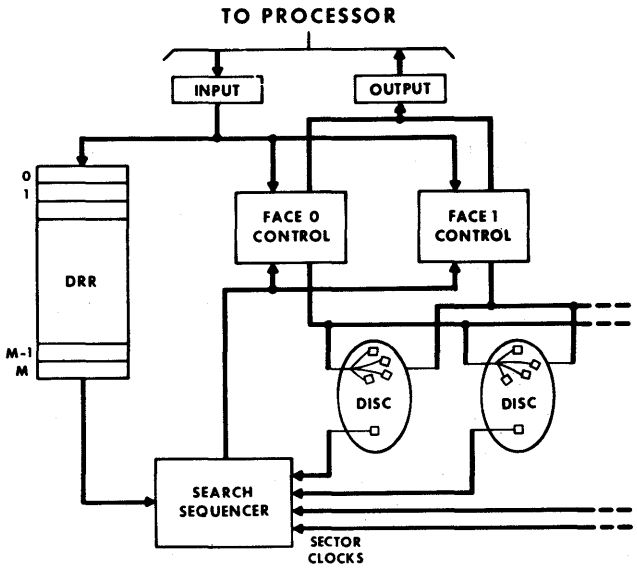


Figure 4—Disk controller

(i.e., sector, disk, and face). Each slot in this queue is typically less than 12 bits in length since this hardware is moderately expensive. In this queue, the number of slots is expandable and depends on the throughput requirements of the disk controller. Each register position in the DRR has a corresponding four words in high-speed memory (as shown in Figure 3), which specify all the processing information for a job. The disk request buffer (DRB) has the complete disk record address, disk-face-sector-track, and starting word in the specified record. It also contains the number of words following the starting word to be delivered to the system and the location in high-speed memory in which the data is to be placed (read mode) or to be taken from (write mode). The last word in the DRB is reserved for the status of the job and a "done" indication which is set when a controller completes a job.

The done indication in the fourth word of the DRB signals the two duplicate controllers to update the status of their respective DRRs. Requests shorter than ½-disk revolution are placed in both controllers, which search their respective DRRs for jobs in the next sector on each disk. For example, if disk controller 0, in the jth position of its DRR, finds a job which matches the next sector on one of its disks, controller 0 will access the four words of the corresponding DRB to obtain the processing information. If the done indication is not set, controller 0 will complete the job as specified, zero the jth slot in its DRR, and write back a done indication to the fourth word of the DRB (in jth position). One-half revolution later, when disk controller 1 finds the same job and obtains the fourth word of the DRB, it detects the done indication, zeros the jth position of its DRR, and continues searching for another job.

A complete search of the DRR takes place each sector for jobs in the upcomming sectors on all disk faces within a controller. As shown in Figure 4, each disk controller has two independent circuits which allow two simultaneous operations in each controller. The sector information is derived from each disk's timing tracks and is registered in the search sequencer circuit. Using this information, the circuit searches all M positions of the DRR. Having found a job (address sector match), the search sequencer informs the proper face control circuit which executes the job. The search sequencer does not search for that face again until that face informs it of a job termination. In this mode of operation, if the request size is smaller than the number of words in a sector, the controller has the capability of processing 50 of the 100 sectors per revolution on two faces: a total of 100 jobs per revolution. Thus, a controller community (a controller pair with duplicated and phase-locked disks) has maximum capabilities of 200 jobs/revolution with average latency of one quarter of a revolution time.

Simulator

To understand and engineer the system more fully, a FORTRAN real-time simulator was written for use

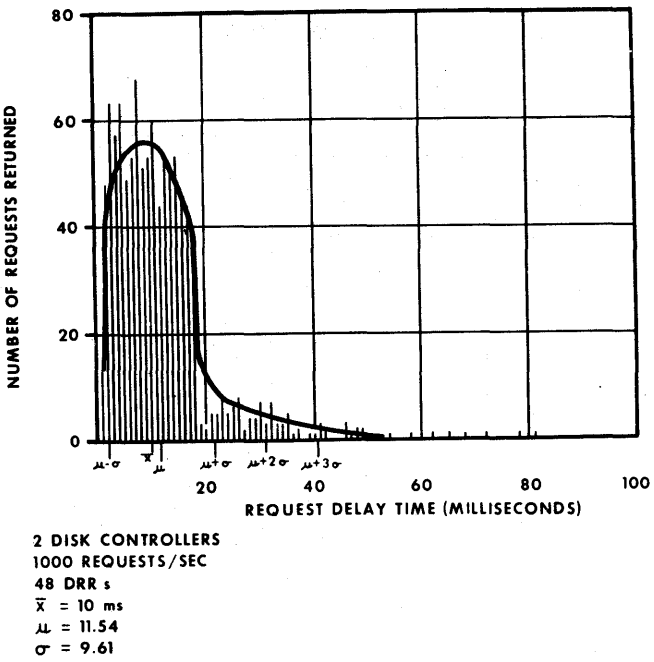


Figure 5—Distribution of returned requests

on an IBM 360 model 67 computer. At this time, the simulated system includes "n" controller pairs with one CPU and one high-speed memory. The CPU occupies the high-speed memory, varying from 0 to 100 percent of the time. The simulator incorporates all blockage and priority characteristics in the program. The throughput, request submission rate, and distribution of request sizes are all parameters of the programs. The output of the simulation programs is a request delay distribution, the percent of CPU real-time blocked, DRR occupancy, and the number of jobs aborted because of disk data overflow (a low priority controller repeatedly blocked by CPU and other controllers).

Data pertinent to the simulation is listed below to clarify the forthcoming results:

a. 24 bits/disk word
b. 32 disk words/sector
c. 100 sectors/disk revolution
d. 35 milliseconds/disk revolution
e. request size characteristics:
   1. maximum request size = 32 words
   2. minimum request size = 2 words
   3. average request size = 8 words
f. simulation run time = 1 second of real time.

The disk address for each request was randomly distributed. The number of requests submitted at each submission interval was uniformly distributed. The size of each request was drawn from a distribution with characteristics as shown in (e) above. Since write requests generally store data for later use, and since system response times are not critical factors, all requests asked for data reads from the disk.

Simulation results

In the delay time distribution shown in Figure 5, the average request delay was near the ¼-revolution time predicted. The largest percentage of requests is answered within the first half revolution time; requests for the same sector are dispersed over several revolutions.

The next important characteristic is the amount of real time wasted by the CPU due to disk controller blockage. Note that in Figure 6, with the CPU using the high-speed memory 60 percent of the time, the disk system (delivering 100 requests/second) uses only 1.5 percent of the CPU's real time.

In Figure 7, the maximum and average occupancy curves of the DRR allow optimum design of the DRR size for a particular application. For a desired throughput of 1000 request/second, a 12-slot DRR would be desirable. With some degradation in delay characteristics, any size between 6 (average occupancy) and 12 (maximum occupancy) would suffice.

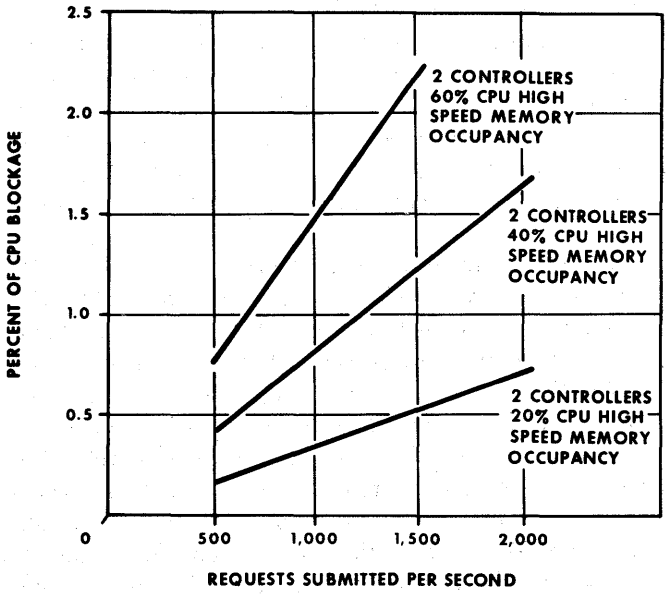


Figure 6—Central processing unit (CPU) blockage

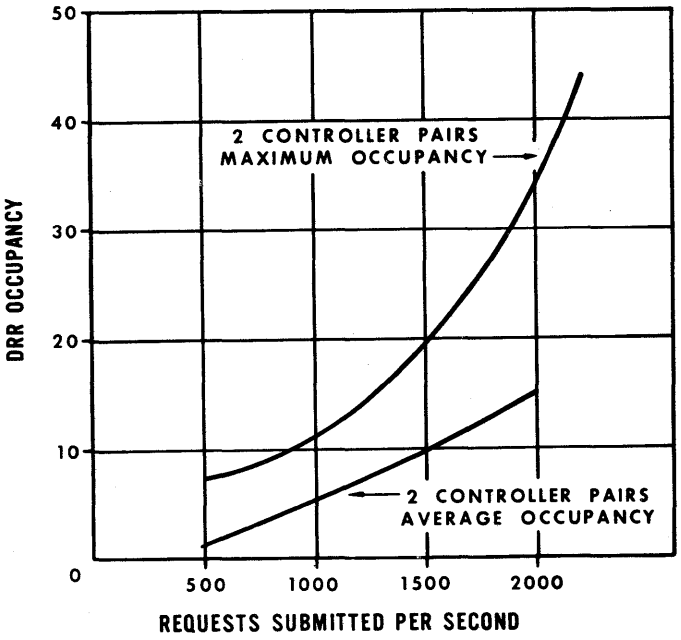


Figure 7—Disk request register (DRR) occupancy

SUMMARY

This paper presents the design of a high throughput autonomous disk memory system. Through extensive use of hardware and software queues and direct main memory access, the disk controller delivers high volumes of data, requiring a small amount of the CPU's real-time capabilities. By the 180-degree phase locking of duplicate disk files, the latency of the memory system is reduced to half that of a single disk, while providing high system reliability. The computer simulation results show that the design criteria are indeed satisfied.

ACKNOWLEDGMENTS

The author wishes to acknowledge his indebtedness to the many members of the Indian Hill Switching Division of Bell Laboratories who have participated in the design of the disk controller. Deserving of particular mention are D. M. Collins and R. D. Royer for system and hardware design. Also, many thanks to S. G. Wasilew for his design of the simulator.

Management problems unique to on-line real-time systems

by T. C. MALIA

IBM Corporation
Chicago, Illinois
and
G. W. DICKSON

University of Minnesota
Minneapolis, Minnesota

INTRODUCTION

In the latter 1950's, the SAGE air defense system began operating and thus became the first of the large real-time computing systems. Initially such systems were feasible for only military use or for a few very large commercial applications. Today this is no longer the case. Modern managers need better and more timely information to keep pace with the rate of change, the complexity and the competition within the business environment. Therefore, an increasing number of organizations will, of necessity, be designing and implementing on-line, real-time systems. To better prepare for this evolution, management must understand the unique problems such systems will cause; both in terms of the initial design and implementation phase, and the potential effects such systems will have on the organization. The purpose of this paper is to outline the particular problems that managers will have to deal with in an on-line, real-time (OLRT) environment.

For the sake of clarity, several terms utilized throughout this paper require some discussion. These are: (1) on-line, (2) real-time, and (3) time-sharing.

On-line refers to a system in which input data enter the computer directly from their point of origin and/or in which output data are transmitted directly to where they are used.

An exact definition of real-time is difficult because this is a relational concept. What is real-time in one instance is not in another. Probably the most widely used definition is one that says such a system is one that receives data from remote terminals, processes them and returns the results sufficiently quickly to affect the environment in which it is operating. A more operational definition is as follows: "t is the amount of time that the computer could suspend computation and then resume without changing any of the inputs or outputs in the system."[1] The shorter t is, the more possible it becomes to class the system as real-time. For example, a one second interrupt in a missile launching is intollerable. The inputs to the system would have to be altered to account for the movement of the missile. On the other hand, a three hour delay in a payroll program would not usually require any such changes. Systems are usually not referred to as real-time unless t is at most a few seconds and more often a few milliseconds. One should note that it is possible to have an on-line system that is not real-time, but the reverse is not true.

We include time-sharing systems as a subset of OLRT systems. These systems, which "simultaneously" permit a number of users connected to the system by remote terminals to utilize the power of a central computer, not only share many of the problems associated with OLRT systems, but introduce additional complicating factors.

As noted earlier, the scope of this paper is management problems associated with OLRT systems. It will pertain to not only the managers directly responsible for the system design and implementation effort, but the top managers of the firm as well. Special emphasis will be placed upon OLRT systems which heavily emphasize human interaction such as information retrieval type systems (e.g., airline reservation systems) and on-line management information systems. Time-sharing systems will be covered implicitly where time-sharing problems are similar to OLRT system

considerations and explicitly where time-sharing systems pose unique problems. Many problems associated with OLRT systems are similar to those encountered in the design and implementation of any large computer system. In these instances, the paper will discuss only those additional complications which operating in an OLRT environment introduce. Finally, the paper will not discuss the myriad of technical factors involved in designing real-time systems, but it will refer to these factors because of the need for management to control and guide the effort involved in solving these problems.

OVERVIEW

Regardless of the type of OLRT system involved or the specific application, there are some general reasons why the design and implementation problems of a real-time system are more formidable than those encountered in batch processing systems. These reasons include:

1. OLRT systems rank among the very largest computer systems ever conceived. This is usually due to the fact that such systems are assuming functions that in earlier systems were performed by human operators. In addition, these systems may, by virtue of their complexity and related expense, be applied to only the largest and most difficult problems.
2. OLRT systems are more complex due to their size and type of application. It is very difficult to engineer, install and maintain a system with many simultaneous users. Also, the program required to poll and respond to these users will have much more involved logic than one written to control only one input device.
3. OLRT systems are new. The science of OLRT applications was first developed for the SAGE application in the middle 1950's. The first commercial application was the SABRE system which was developed in the early 1960's. These systems are new in essentially two respects—the application and the hardware. Frequently the application of real-time systems is to an already existing problem. However, usually it is a problem that has not lent itself to computer solution before OLRT. Thus there are no previous attempts at computerization from which to learn. In the case of the system to monitor manned space flights, the designers were confronted with not only a pioneering attempt at computerizing the problem, but also the fact that the problem itself was entirely new. In addition, many of the pieces of hardware necessary to facilitate an OLRT system are new. Most OLRT installations may be somewhere between 5 and 100 percent new in an equipment sense.
4. OLRT systems are more vital and thus disruptive in the event of a malfunction. By definition, an extended interruption in an OLRT system will require changing the inputs or outputs of the system. Unfortunately, if the delay is extended, severe damage to the environment may occur before the corrections are made. The earlier reference to the real-time space flight monitoring system is a good example of an extremely vital application and one that could be seriously impaired by a failure of the system. In a batch processing environment, a down system simply means that records will not be updated or reports generated for a few hours. The OLRT system's inability to function can have a very direct and costly impact on operations. These systems are adapted more to the immediate operations of an organization, rather than historical record keeping. In the case of the airlines reservation system, the failure of the system will seriously threaten a waste in the firm's vital seat inventory not to mention the impact upon customer satisfaction.[2]

PROBLEMS IN THE APPLICATION OF OLRT SYSTEMS

Management involvement

The McKinsey Company's survey of computer installations showed that the one thing that the majority of successful installations had in common was the fact that the executive managers of the company devoted time to the computer systems.[3] This time was spent in reviewing the plans and programs for the computer systems effort and in following up on the results achieved. The involvement becomes even more crucial in the installation of an OLRT system. Management must insure that the tremendous resources involved in the design and implementation of such a system are allocated in an optimum fashion. Thus management must not only plan for the system, but must also get involved in the planning of the system. In this planning process, managers should be very aware that such systems might cause extensive changes within the firm and should attempt to predict

the effects of these changes. Possible changes may include: (1) the eventual reduction in the power of middle managers, (2) a trend toward centralized decision making, or (3) the necessity of middle managers to work in a very restrictive and highly controlled atmosphere because of the information their superiors have. By being knowledgeable of these effects, top management can design their managerial philosophy, and the system itself, in such a way as to minimize the undesirable effects and to optimize the areas where good effects can be achieved. The persons within the company most affected by the new system must be integrated into the system design effort to gain their support and cooperation, which is essential if the system is to succeed. Throughout the company, top managers must prepare personnel for the effects of the change, thereby reducing the reluctance of people to comply with the system requirements. Executive management effort is also needed to coordinate the many parts of the system and to insure that the designers are given the cooperation the system's eventual success requires.

The system design is far too crucial to leave in the hands of technicians. The system will change the work methods of humans and will be dependent upon humans for proper functioning. This human element is extremely important and certainly should not be ignored. Another reason for the involvement of top management is that extremely expensive trade-off decisions must be made regarding acceptable levels of system performance, reliability, and capability. For instance, top management should not allow systems people to make the decision to spend $1 million annually to improve the response of the system by five percent. These are decisions for top management to make, based on recommendations from technical and operating personnel.

Personnel

The first problem confronting the project management will be that of finding qualified personnel to work on the system. Because OLRT systems are so new, there are not many people available with related expertise or experience. Yet because of the size and nature of such projects, it will be necessary to acquire a large number of programmers and analysts. It will be necessary to train these people in the new terms, new techniques and new hardware, all of which must be understood and integrated into the design effort. In an application where present operating procedures are being computerized, it would be well to have people with a knowledge of present operating procedures as members of the project team. The importance of this team can be shown by Martin's statement that "no single effort is going to have more effect on the success of the system than the recruiting of the best possible programming team[4]."

System design

General consideration

The next problem facing project management is that of system design. This is unquestionably the most difficult, yet most crucial phase of the system's life. What is done in this stage will have a tremendous effect on later stages and on the success of the system as a whole. Essentially the objective of this design effort is to design a system that will best meet the needs of all the users. This must be accomplished within the constraints of reliability, design time and cost.

The reason the design phase is so critical in an OLRT environment is that all parts of the system must be integrated, and the shortcomings of any one part will degrade the total system. For instance, regardless of the quality of the balance of the system, if the users cannot operate the terminals, the entire system is essentially worthless. Or if one channel of the central computer becomes overloaded, the performance and thus the value of the entire system will be affected.

The first step in the design effort is to determine precisely what the system is to do—what applications are going to be converted. It is necessary to fully understand the information handling requirements of these applications. Having determined the particular applications for the system, there are three general considerations which must be evaluated to narrow in on the specific system configuration. These considerations simply provide a framework within which the more specific details of the system can be analyzed and determined. The considerations include:

System Availability—Is the extent to which the system must make itself available to its users. How many hours per day will the system be required? What are the requirements in the event of system failure? Can the users somehow continue to function off-line or can they put off their work for some period of time? Perhaps it is crucial that the system always be available; but, what cost is acceptable to achieve total availability? These considerations all entail the system's overall capacity for performing the application processing in a prompt and satisfactory way throughout the period in which processing is required.

System Variability—Involves determining the long-

term volatility of the applications both in terms of volume and changed requirements. It is difficult to predict this volatility. The existing system is so different from the future system, that it does not even provide a good base from which to project an estimate. Most applications will require modifications and expansion that, where at all possible, should be anticipated and planned for when the system is designed.

Communication Characteristics—Is related to the above two considerations and involves such questions as how geographically dispersed will the terminals be, or what speed of response will be required. Analysts must also determine if there are peaks or cycles, both in terms of the volume of data transmitted and in terms of response time requirements. If such peaks exist, they present a dilemma to system designers, who must decide on the appropriate capability level for the system. If the system is designed with enough capacity to provide for the peaks, this capacity will not be utilized much of the time and will thus represent an unnecessary expense. On the other hand, if the system's capacity is far below that that the peaks demand, the system may become so overloaded during periods of high activity that deteriorated performance renders it useless. Therefore, at some level, a compromise must be made in terms of minimizing expenses while at the same time providing adequate system performance.

The system design will probably be an iterative process in which it is recommended that during the first phase the complexity of the design be held within manageable bounds. In later phases, the more sophisticated features can be added. The largest constraint will probably not be hardware considerations, but rather, the complex programming that will be necessary to achieve the desired results. It thus becomes important that the system design team be aware of the factors that contribute to the cost and complexity of the system in order that, where possible, these factors may be taken into consideration.

There are several specific design factors which are pertinent in the design of an OLRT system. These factors are by no means autonomous and somehow must be integrated and balanced to optimize the system in terms of overall performance and cost. In addition, because this paper is primarily devoted to OLRT systems requiring human interaction, the aspects of man-machine interface are very important. This topic will be covered in depth in the last section of the paper. However, man-machine considerations will also be discussed where they specifically affect other facets of the system design. These other facets include such things as the design of the data base, the terminals and system scheduling all of which will be discussed independently. It is important to keep in mind, however, that they are all interrelated and that each affects the overall system design.

Data base

This design aspect entails the physical and logical organization of the data within the system. In an OLRT environment, data base design is particularly important due to the response time requirements and the typically large amount of data involved. The general objective of the data base designers is to optimize the following interrelated features: (1) minimize the access time to get information, (2) maximize the ability of the system to respond to questions both planned and unplanned, and (3) achieve the above two features with the least overall cost. Man-machine considerations will undoubtedly impact the acceptable limits for the first two objectives. In addition, for applications where the machines must interact with untrained users or must interact on a broad spectrum of topics, the data base designers might want to enable the user to converse with the system without having to translate his requirements into codes or very specific formats. However, while such free form formats will improve the user's ability to interact with the computer, it will significantly affect the system because of the need for special software to interpret the input. The design team must be acutely aware of the particular needs of those using the data base. It must plan for expected needs of these and future users and the expected increase in volume of additions/deletions and inquiries to the data base. There are several packaged data base management systems available (like TDMS, IDS, IMS) which designers would do well to investigate concerning the applicability of these systems to their needs. If such systems are appropriate, their implementation will obviously eliminate a tremendous amount of programming. However, systems performance should not be sacrificed to achieve such initial economy. Designers must devote special emphasis to the data base portion of the design process because of the tremendous effect this design can have on the performance of the total system and on the ability of humans to readily interact with the system.

Scheduling

This design problem is especially crucial in a time-sharing system but, depending on the applications involved, can be very important also for any OLRT system. The problem is that the scheduler must be

established beforehand to determine how the computer is to service the terminals and how it is to carry out the necessary computations. Essentially the objectives of the scheduler are to:

1. Minimize the average response time and the number of users waiting.
2. Recognize the users importance and the urgency of the request.
3. Serve the users in a fair order and limit the length of the wait.

Some of the possible scheduling methods include: (1) first come-first served, (2) round robin (the many users get small slices of time until their job is completed), and (3) a priority system, where users get quantums of time depending on the size of their job, their priority level and system activity. The disadvantage of the first come-first served method is that users with very short jobs have to wait as long as a person with a large job. The round robin method would resolve this problem by completing the short job in the first time slice and thus reducing waiting time. As a general rule, the round robin method is more beneficial than the first come-first served when the amount of computation required is uncertain.[6] Here again, the particular scheduler used will be affected by the requirements of the human users and the desired performance of the entire system.

Data communications

Due to the nature of OLRT systems, it is necessary to transmit data over communication lines. In the great majority of applications, this necessitates using common carrier's lines, which immediately poses a problem for managers—that of coordinating the design and compatability of the system with another vendor. New hardware is also introduced, whose capability and compatability must be understood and implemented. The problem is further complicated by the fact that these communication vendors have a whole new set of "buzz-words" like bauds and duplex lines which must be comprehended. Also, in order to design an economic and efficient communication network, it is crucial to understand the various rates and facilities offered by the common carriers.

Terminals

The introduction of terminals poses problems similar to those in data communications. Here a third vendor may be introduced, further complicating the coordination and compatability problem. Essentially the selection of terminals involves, to one extent or another, three interrelated factors. These include: (1) the man-machine interface, (2) training of users, and (3) the physical design of the terminals. The first two of these factors will receive treatment later in the paper.

Essentially the goal of the terminal designer is to design a terminal which will best enable the user to interact with it, will require a minimum amount of training, and will not have a significant detrimental effect on the cost or response performance of the system. That is obviously a large order. The trade-offs essentially involve the initial cost of the terminal and the ease of use of the terminal in a specific application. The initial cost may be reduced by utilizing an "off the shelf" all purpose terminal, which is tailored to meet the user's needs. It is usually the case, however, that such terminals will involve extensive training for the users and will not achieve the ease of use level that is best if humans are to readily utilize the hardware. The other alternative is to utilize terminals that are specifically designed for the particular application. This will make them easier to use and thus reduce training requirements, but it will increase the cost of the terminals. Some basic recommendations based on past experience are:

1. Keyboards should be optimally designed to provide the user with a choice of relatively few options at any step in the solution process.
2. Where feasible, the user should be provided with tags or plates which contain information familiar to the user and the coded representation of that same information. These codes must be recognizable by the computer and easily scanned at the point of origin.
3. In instances where the input data must be entered by typewriter type keyboards, it is essential to provide a means by which the user can visually verify and correct the data before it is entered into the processor.

Also important in the physical design of terminals are security factors. Such things as whether audit purposes will require a hard copy of all input and output data, or whether the keyboard might have to be physically locked to prevent unauthorized users must be considered. The designers will also have to determine if they want the terminals to be able to perform some functions off-line in the event of system failure.

Failure

A very important consideration throughout the design process is that of minimizing both the chance

of the system malfunctioning and the overall effects of such a malfunction. It should be obvious that an OLRT system on which many people are relying for information has some very stringent reliability requirements. In addition, because much, or in some cases all, of the information pertinent to a firm's operations, is contained in this system, its security requirements —in terms of information being available—are very high. The hardware reliability problem is usually solved by duplexing or duplicating the equipment. Depending on the need for reliability, the entire system may be duplexed or more simply, various modules may be replicated. In an airline reservation system, for example, the entire CPU is duplexed to permit back-up. An example of a duplexed module is IBM's 2314 disc drives on which one spare drive is available in the event of failure. These precautions obviously greatly add to the cost of the system and thus reliability design involves some important trade-off decisions. Usually in cases where the CPU is duplexed, the spare unit is kept busy doing batch processing, but is ready to be switched over in the event of a failure to the real-time system. The ability to process while serving as a back-up helps somewhat to alleviate the reliability expense.

The reliability or security of the data base is also assured by duplication. This is usually accomplished by systematically storing a duplicate of every record in a file in a slow and inexpensive storage device, usually magnetic tape. To complete these security precautions, it is necessary to maintain a record of transactions occurring between the transcriptions. In this way, if a disc is accidentally erased, the previous status of the file can be taken off a tape and the subsequent transactions affecting this file can be re-created from the transaction record mentioned above, thus bringing the disc file up to its former status.

Assurance of reliability is not as simple as duplexing operations. There are extremely complex software routines which must be written to detect pathological conditions. In such an event, the system automatically switches to another unit or informs the operator that such an operation is necessary. However, before the switch is enacted, the status of transactions within the failing system must be determined and transferred to a spare system or module. Often times OLRT systems are designed to "fail softly". This occurs in a situation where only one component of the system fails. Rather than interrupting service entirely, the system modifies its mode of operation and will continue to carry out the critical jobs, but will give a degraded form of service. This is called "graceful degradation", rather than total system failure.

In the event of total failure, the users should be notified immediately so that they can convert to bypass procedures. It is important that system designers have such a set of procedures established so that users can continue to conduct their business. Terminal operators might function by utilizing operating information which is periodically given to them or they might have a central office to phone to get critical information. Just as it is important to plan for failure, it is also important to plan methods to enable the computer system to update its records with the transactions that occurred while it was down.

The duplication of data files is complicated in an environment in which they must constantly be available for update and inquiry. Here software must be written to momentarily lock-out—not permit access to—small portions of the data base and during this time, the data is copied and subsequent transactions monitored to facilitate re-creation if necessary. These procedures will minimize the effect of errors due to (1) electrical or mechanical failure of hardware units, and (2) accidental erasure of memory through hardware or software error. Another cause of failure—physical damage by disaster—must be planned for. In many installations, the tape copies of the data base are physically stored at another location to minimize the chances of destruction in event of disaster. Basic safety precautions i.e., fire door and alarms and moisture alarms, are of course, important to minimize the chances and effects of major disasters. Insurance is a must, but the amount must be evaluated based on cost and the probability of various types of disasters.

Security

The security of the system mentioned above pertains to the security of the data once it is in the system. Also important, is the assurance of correctness of the data base. This assurance is especially critical in a real-time system because users are relying on the information to make immediate decisions. If the data are in error, many potentially bad decisions may be made before the error is discovered. On the other hand, the data in the system are more difficult to check because they come in directly from many remote locations. Thus these inputs must be edited by programs and any incorrect or unrecognizable data rejected and the sender notified.

It is also essential that the data base and the transactions be auditable. CPA's, internal auditors and other audit agencies will want to review not only the files and the transactions, but also the programs that are being used. The possibility of auditing around

the computer in such configurations is essentially nil. Therefore, the system must provide the necessary trails and documents to satisfy the auditor's requirements.

A final area of required security is that of insuring that the data base cannot be altered or addressed by unauthorized persons. This is especially important in time-sharing systems, where controls must be present to preserve the security of proprietary information. The problem is typically handled in two ways. First, terminals may require a special key or badge or code to activate them. Secondly, the system software may allow only input from certain terminals. Another feature like memory protect may be used to allow only certain portions of memory to be accessed from some terminals.

It should now be obvious that there are a vast number of interrelated factors which will affect the design of an OLRT system. Essentially the problem confronting system designers is to tie all these factors together and devise a system which is an optimum compromise between financial practicality and operating perfection. As of yet, there is no general mathematical procedure which can be followed to achieve this compromise. Such tools as simulation, however, have proved very helpful. The problem with simulation is that much of the data needed for simulation are not available until the system to be simulated has been designed and put into operation. Therefore, the initial design may best be done by trial and error and simulation used to refine the system. In this way the affects of any design changes can be investigated. On the other hand, if simulation is used early in the design phase, many of the facts will not yet be known and thus the designers must revert to judgment and intuition. Extreme caution must be utilized in this situation, to insure that the inputs are reliable or the GIGO (Garbage in-Garbage out) theorem will come into effect. Thus if properly utilized, the benefits of simulation will offset the time and expense necessary to design and utilize it. This technique will also require designers to take an organized approach to system study and will help to insure that no important factors have been overlooked. It is also a useful tool to assist designers to plan for future hardware needs.

Hardware selection

The selection of hardware to use in the real-time environment is, for the most part, a technical matter. There are some factors, however, for non-technical managers to be concerned about. The first of these is the ability of the hardware to expand. OLRT systems have a marked tendency to grow and thus it is important that the selected system can expand in terms of number of communication channels, size of internal memory and capacity of file storage. The most important criterion in this selection process should be that of reliability. In this respect, it is important to analyze not only the amount of downtime on projected systems but the type, duration, and distribution of periods of downtime. In an OLRT system, an extended period of downtime is much more critical than several short periods, although several brief failures would certainly lead designers to question the reliability of the system.

Modularity is a factor which is pertinent to both the expandability and reliability of the system. If the entire system is composed of many connected modules, capacity can be added in small units so as to reduce costs for excessive capacity. In addition, because most failures in a real-time environment are in one component of the system, modularity can assure reliability at a much lower cost. For instance, 20-30 percent redundancy in a modular system buys the same reliability that 100 percent redundancy buys in a non-modular system.[7] However, connecting the operations of these modules does put an added burden on the system's executive routines.

The hardware selection should also be based on the knowledge and experience of the vendor in real-time applications. A very important factor, in this respect, is the amount and quality of real-time software the vendor has available. The data communications software and a real-time operating system entail a very large and complicated programming effort. Thus it greatly reduces the work load on the system programmers, if the vendor has such routines available. The routines should be of proven quality and should provide an adequate performance level to meet the application's specifications.

Programming

Programming is by no means a consideration separate from system design. As mentioned, the designers should clearly assess the programming complexity of their design considerations. Aside from the technical programming problems in an OLRT environment, such programming also requires additional managerial capabilities of those in charge of the department. These needs are brought about by the main difference between programming an OLRT system and a batch processing system of similar complexity. The difference is that the former must be a tightly integrated and controlled piece of teamwork. Programmers are by

nature an independent, creative group of workers, most of whom feel they need freedom to work. The programming team manager must closely supervise their activities, must control their creativity and yet must motivate them to solve difficult problems. The programming job is often very frustrating because even very minute changes can necessitate extensive modifications of the work already completed.

Aside from these behavioral problems, programmers in an OLRT environment are exposed to a wealth of technical problems which they must work with and resolve. These problems are essentially:

1. The use of terminal devices requires the programmers to learn their operating characteristics. Programmers must send and receive special command characters to utilize the remote terminals and to control such things as keyboard shifts, carriage return and message marks.
2. The problem of errors is especially critical in a real-time environment. Data errors must be dealt with in such a way that erroneous data are not allowed to enter the system. Yet the program response time must stay within the time constraints.
3. The programmer must adapt himself to the somewhat different characteristics of the computers used in OLRT configurations.
4. The data that programmers work with will be unusual. It will have control characters imbedded, and will often be represented by a code structure different than the one used internally by the computer.
5. Programmers must be especially cognizant of testing, storage and timing considerations unique to terminal oriented systems.

Programming managers can greatly alleviate their problems by dividing the entire system into semi-autonomous sub-systems. If the division is properly handled, the number of interactions between programs and programmers can be reduced. Once this is achieved, however, input/output formats among programs must be standardized and rigidly adhered to. A standard procedure should be established which programmers should follow if they believe a change in the system is necessary or desirable. No changes, however minute, should be allowed without following this procedure. A special control group may be established to coordinate and evaluate suggested changes in formats and to decide on changes and communicate these to the parties affected.

Training of users

Training of the user in a real-time environment is crucial for essentially two reasons. First, most of the raw data in the system are provided by personnel with other operating responsibilities. For instance, the airline clerk is concerned with selling tickets, the production worker is involved with his assembly work, and the bank teller has a line of people waiting to be served. Thus it is crucial that these people are trained to easily, yet accurately, input the data into the system. As noted earlier, the physical design of the terminals, combined with proper training, will affect the user's ability to achieve the desired ease of use.

Secondly, one of the basic purposes of providing an OLRT system is to enable user personnel to extract meaningful, timely information. Obviously if they are not properly trained to utilize all the features, the system is not going to provide the service or have the effects which its designers intended. Along the same lines, it has been found that many times the reason why people do not utilize new equipment is that they are unsure of their ability to operate it. To solve these problems, many OLRT users have connected terminals to small test computers, to simulate the system, and to enable users to get experience in the man-machine interaction environment. Another area where training is important is that of by-pass or off-line procedures when the system is down. Hopefully, the users will not have too much actual experience in this mode, so they will have to be periodically retrained on the proper procedures.

System testing

Many of the problems of OLRT system testing were alluded to in the section on programming. Essentially the complicating factor is the fact there are a large number of programs, software, and hardware features all interacting, plus an infinite combination of input/output states. Thus it is difficult to systematically test all these interactions and combinations. In addition, once an error condition exists, it is extremely difficult to go back and re-create the situation which caused the error. Despite these difficulties, the vitalness of the system to the firm's operations, requires that the system's reliability be fully tested—more so than other configurations.

By far the most crucial element in OLRT system testing is to plan the system from the beginning so as to include all of the testing facilities which will be necessary. The hardware configuration should be planned in such a way as to isolate the causes of error. The preparation of testing facilities must be carefully

monitored to assure that they are finished and available when the operational portions of the system are completed. Part of these testing facilities are programs to simulate the various portions of the system, such as the remote terminals. The more sophisticated these programs, the greater assurance designers will have of the reliability of the system. As an example of the amount of work involved in preparing these testing programs, Desmonde tells of an installation in which more than twice as much labor was expended in preparing utility programs and programs for testing and simulating the system, than was used in writing the operational programs.[8]

Robert Head outlines a recommended five step approach to the complete testing of OLRT systems.[9] For the first step, the individual programs or packages of related programs are debugged and tested with simulated programs which duplicate more or less the functions of both the hardware and control programs with which the programs will be interacting. The second step is to test these programs or packages in conjunction with the control program. This satisfies the dual purpose of testing both the application programs and the control program, if it was written or modified in-house. The third step is to supply simulated inputs from the multiplexor to determine the possibility of this device as a source of error. Also in this phase, the program packages are further combined into subsystems and the volume and variety of the input data is vastly increased to test the performance and overload characteristics of the subsystems. The fourth step is that of system testing. Here all the pieces are put together for a final integrated test of the entire system. Prior to this phase the terminals have been debugged from an equipment standpoint. They are now connected to the system in order to provide entries in a mix, a sequence, and a format and content that duplicates actual operating conditions. Also at this stage, the automatic switching and other "graceful degradation" software of the system should be checked. The final step has to do with evaluating not whether the system is capable of performing satisfactorily, but how well it is performing when measured against the functional or performance requirements. This acceptance testing should be an on-going process for the life of the system to ensure an efficient system organization and to provide sufficient lead time for system modifications necessitated by oversights in the initial design or by load growth.

Conversion

The problems encountered in the conversion and implementation phase are not really too different from those in any large system conversion. One of the factors that makes this conversion crucial is that, unlike batch processing systems, once the cutover of a real-time system is accomplished, the user is fully committed, i.e., he has virtually no satisfactory way of turning back to his former system. For this reason and because of the huge sums of money involved and the typical disruptions such conversions cause, the implementation must be carefully planned and scheduled and closely monitored. The SABRE system was implemented on a location by location basis and the designers recommend this procedure because it: (1) shortens the learning period at each location, (2) enables the firm to operate with only a small portion of its real-time functions undergoing a major change at any one time, and (3) the remaining bugs in the system are eliminated before the entire company is relying on the system for its operation.[10]

MAN-MACHINE INTERFACE

The man-machine interface problem is certainly pertinent in discussing the design of a computer system with which humans must interact. For this reason, the man-machine interaction issue has been discussed as a part of the design phase of an OLRT system. However, because of the complexity of the problem, the vastness of the questions involved and the extent of the still unexplored areas, this topic will be discussed in more detail in this section. The discussion will center upon what are the causes of the problem and some general rules that have been established to deal with the man-machine interface problem.

Sackman gives an indication of the complexity of the problem by his statement that "the most difficult and vexing problems in an OLRT system are not in the maze of hardware or the intricacies of the software, they are in the enigmatic nature of the human users."[11] This issue of man-machine interface is by no means a new one. For a long period of time, our society has had machines which are highly analogous to man's muscle and yet controlled by man's brains. In recent times, we have developed information processing machines that are in many ways functionally equivalent to man's brains.

In an OLRT environment, users are interacting with the system in two different contexts. In the first, the information processing machine is interacting with man's muscle or essentially the machine is telling man what to do. Examples of such systems include: (1) the space program in which information is processed by machines and men are told which

functions to carry out, and (2) airline ticket agents that are informed by the system whether or not to write a ticket for a specific flight. The second context is where the information processing machine is interacting with man's brain. In this instance, the man-machine interface is extending and augmenting human thought. Here the ability to completely and readily communicate is much more crucial, yet much more difficult to achieve.

The objective of those concerned with the man-machine problem is essentially to design a system that will most effectively take into account the limitations and talents of both man and machine. It is important to realize that the human factors are in certain respects diametrically opposed to those of the computer. The respective talents and limitations we must account for are the reliability, speed, and accuracy of the computer and the intelligence of man. Intelligence is the ability to learn or understand from experience and as a result the ability to respond favorably in a changing environment. Therefore, the system must not only utilize the talents of the components involved, but must also enable the human user to readily and comfortably interface with the machine to offset his limitations with the talents of the computer. This interface gives rise to many questions such as: How fast a response from the computer does the user need for different types of tasks? How much variability in computer response time can the user tolerate? How concise or redundant should man-computer communications be? What are the optimal languages for man-computer communication? What sort of feedback should users receive from the computer for various classes of human and system errors? How can the system help the user when he gets into trouble?

To answer these questions and to effect an interface, designers have essentially two variables with which to work—the system hardware and software. However, in dealing with these variables, designers must have some concept of what to expect of man. A knowledge of the underlying psychology and physiology of man is helpful in dealing with this problem.

Some would argue that since man has great flexibility and can readily learn skills, he can ease the interface problem by adapting. The truth in this statement is not at all clear. Clearly there are some tasks man can never learn. He cannot for example reduce his reaction time below some limiting value on the order of 100 milliseconds. Some of the other limiting factors in a man-machine interface include:

1. The concept of information necessarily involves a choice of one from a set of alternatives. This necessity for discrimination however is far from the limiting factor. The ability of man to perceive changes is ever so much more acute than the ability to identify items in isolation. Professor George Miller has developed "the tale of seven plus or minus two".[12] This rule is that for most single perceptual dimensions, it is true that the average subject can perceive or can identify at most 7 ± 2 stimuli presented on that dimension. Multidimensional displays can be used to improve this identity rate.
2. The human reaction time limit mentioned earlier was for the most elementary tasks. For more difficult tasks, involving identification, reaction time increases linearly with the number of choices, unless the identified objects are drawn from a very familiar set such as letters or numerals.
3. To maximize the information transmission speed across a man-machine interface, it is best to choose a large familiar alphabet. If this is done properly, rates up to 40 bits per second can be achieved.
4. The rate of learning or the ability to identify from a set increases with the familiarity of the set. It is not always obvious from the physical dimensions of the stimulus what is familiar. However, objects or displays which are in habitual use often turn out to be good choices. Thus despite the fact that man is flexible, he is not infinitely so. Designers should take his limitations into account when designing systems.

The essence of the man-machine interface is the dialogue which occurs between man and machine. This dialogue is very dependent on the language and hardware used, but also on the speed of response. Ideally the response should be rapid enough to not cause discomfort on the part of the user and to enable him to forget that he is sharing the system with anyone else. Thus the supervisor or executive is a critical factor for effective man-machine interaction. The executive or scheduler must strive to return an answer to short problems as quickly as the user can react. For longer problems, where considerable computation is involved, the user is psychologically set to endure delays. Therefore, the system should provide immediate turnaround for short jobs and push the congestion delays and thus variation in response time toward the "long-problem" end of the spectrum. Psychologically this makes the device appear more private and self-contained and lessens irritation due to delays.

The man-machine dialogue also presents a language

barrier. Man's languages are imprecise and context orientated; computer languages are unambiguous with minimal context modification. Most users will generally be untrained in artificial languages precise enough to be interpreted unambiguously by computers. In addition, the computer processes information much faster than man can respond. Thus the interface requires translating and buffering so that the requirements of both man and machine can be met. The translation can be facilitated by the development of problem-oriented or people-oriented languages which do not require specialized programming skills.

Consoles must be designed to enhance the interaction between man and machine. In deciding on the type of console, designers must determine the advantages of the particular console with respect to ease of use, flexibility in format and content, and the achievement of man-machine symbiosis. Ease of use refers to the amount of special preparation required and will be affected by not only the console but also by the interface language used. Flexibility is simply the ability of the user to get what he wants in the form he wants. Finally, symbiosis relates to the ability of the user to hold a discourse with the machine. He must be able to talk back to it in his language in developing problem solutions.

The interactive cathode ray tube consoles seem to best fulfill these requirements. Such terminals, often called graphic input/output devices, should allow other than alphanumeric symbology. Some of the ones now in use provide capability for drawing, following lines and curves, displaying shapes, and responding by touch only. What is most important is that they facilitate information exchange between man and computer directly via graphics without the need for reducing all such exchange to words.

CONCLUSION

The essence of designing an OLRT system that meets the user's goals and that incorporates an effective man-machine interface is to achieve the proper balance between all the pertinent factors in the system. There are a multitude of considerations that must be interconnected and blended in the proper proportion to realize the design goals. Control, coordination, and communication throughout the entire system design and implementation phase is thus extremely critical. But before this phase is initiated, designers must be fully educated on the various aspects of the system and must understand how these factors will interact to affect the eventual performance of the on-line, real-time system. The designers must not only face up to the technical problems involved but, even more importantly, must meet the behavioral problems stemming from the fact that OLRT systems involve human interaction.

REFERENCES

1 D KLAHR H J LEAVITT
Tasks, organization structures, and computer programs
The Impact of Computers on Management (C E Myers ed) The MIT Press Cambridge Massachusetts 1967

2 R V HEAD
Real-time business systems
Holt, Rinehart, and Winston, Inc New York pp1-4 1964

3 J GARRITY
Top management and computer profits
Harvard Business Review Volume 41 No 4 pp 6-13 July-August 1963

4 J MARTIN
Programming real-time computer systems
Prentice-Hall Inc Englewood Cliffs New Jersey p 368 1965

5 R V HEAD
pp 63-65

6 W J KARPLUS (ed)
On-line computing
McGraw-Hill Book Company New York p 89 1967

7 Ibid
p 94

8 W H DESMONDE
Real-time data processing systems: introductory concepts
Prentice-Hall Inc Englewood Cliffs New Jersey p 152 1964

9 R V HEAD
Testing real-time systems
Datamation pp 42-48 July 1964

10 R W PARKER
The SABRE system
Datamation p 52 September 1965

11 H SACKMAN
Computers, systems science, and evolving society
John Wiley and Sons Inc New York p 79 1967

12 G MILLER
The magical number seven plus or minus two: some limits on our capacity for information processing
Psychological Review Volume 63 pp 81-97 1956

ECAM—Extended Communications Access Method for OS/360*

by GERALD J. CLANCY, JR.

Programming Sciences Corporation
Natick, Massachusetts

INTRODUCTION

Installations utilizing OS/360 which wish to extend the operating system's use into a teleprocessing environment all face a similar problem: How to prevent the significant waste of resources, particularly that of main storage, that inevitably accompanies a move from batch to on-line processing? QTAM organization normally utilizes one region (or partition) for its Message Control Program and one region (or partition) for each process, or application, program. Thus, the TP configuration becomes inordinately expensive due to resident core storage requirements, most particularly if the applications are low-volume oriented. An alternate approach, via the use of the BTAM facilities, requires much more extensive knowledge on the part of both system designers and programmers and may well generate more severe and complex problems.

The Extended Communications Access Method (ECAM) was developed by Programming Sciences Corporation to meet this common problem and, additionally, to minimize the programmers' required knowledge of teleprocessing and to provide the installation with dynamic operational control over its TP environment.

MOTIVATION

QTAM overview

The Queued Telecommunications Access Method (QTAM)[4,5] is part of the software communications support supplied by IBM under the MFT (Multiprogramming With a Fixed Number of Tasks) and MVT (Multiprogramming With a Variable Number of Tasks) options of Operating System/360. It is similar to the other "queued" access methods, such as QSAM and QISAM, in that the application programmer is removed from the details of device dependencies. Under control of the QTAM Message Control Program (MCP), incoming messages are routed to specific input queues on direct access storage, as directed by message header information. To the application programmer these input queues are very similar in appearance to a QSAM sequential data set, with which the programmer is most likely already familiar. Even the accessing macros—OPEN, GET, PUT, CLOSE—are similar, differing only in minor detail. The only basic distinction which the programmer must take into account is the fact that the "end of data set" condition may only be a temporary one, occurring whenever the program is active and a lull in incoming messages for the application exists.

When the programmer wishes to send a message to a terminal, he simply constructs the message in main storage, including a message prefix, and "puts" the message to his output "data set," which is referred to in QTAM as a "destination queue." The MCP retrieves outgoing messages in order from the destination queue and forwards them to the terminal.

Systems considerations

While QTAM eases the application programmers' transition from conventional batch data processing to on-line communications processing, the organization of QTAM raises numerous systems considerations for the installation, primary of which are the main storage requirements and the lack of a high-level language interface to QTAM.

* This work was developed in part under contract to the United Aircraft Corporation.

Main storage requirement

A QTAM system is organized with the MCP residing in one partition (MFT) or region (MVT) and the application, or processing, programs in one or more other partitions or regions. There are at least three advantages to assigning each application its own partition or region. First, the applications are protected from one another; second, the addition of a new application program or the deletion of an existing one in no way effects the operation of another application; and third, in the event that an application is abnormally terminated, the balance of the applications remain intact.

The major disadvantages of assigning partitions or regions to applications on a one-to-one basis is that the main storage requirement can become prohibitive, perhaps requiring additional systems, and that abnormal termination recovery is left to the operator rather than the system. The main storage requirement is particularly alarming in view of the fact that, in our experience, low-volume applications predominate in most QTAM installations, thus resulting in very inefficient use of main storage. Consider for a moment the problem faced by an installation with five 40K applications, each of which handles 300 messages, on average, in an eight-hour, first-shift period. The total daily volume of 1500 messages can be handled with ease on a 360/50 or 360/65, with the bulk of CP time still available for background work. Yet, if each application is assigned its own partition or region, a total of 200K must be dedicated to the communications applications. Double the number of applications or the size of the applications and a 512K system must be virtually dedicated after taking into account the residence requirements of the MCP and the Operating System/360.

Even though CP time is available, it is conceivable that the residence requirements of the communications subsystem could severely inhibit the amount of background work which could be executed.

One alternative—assigning multiple applications to partitions or regions—therefore becomes attractive on the basis of more throughput per dollar expended for main storage. However, the questions of added application complexity, storage fragmentation of the parition or region, the effect of abnormal terminations on other applications and subsequent maintenance must be carefully weighed. It is precisely these considerations which were the prime motivation for the development of ECAM.

Lack of HLL interface

One additional installation concern is the lack of a high-level language interface to QTAM, which is supported only at the assembler level. This is of particular concern to many commercial installations who have been accustomed to writing most of their batch applications in COBOL or a similar high-level language. The move to on-line processing may involve the hiring and/or training of additional assembler-level programmers. In addition, the advantages of programming in the high-level language—decreased application development and debug time and cost—are lost to the installation. Such an interface is, therefore, quite desirable.

DESIGN OBJECTIVES

The major design objectives of ECAM, then, were to provide:

1. the capability for application programs to share a common main storage space in a manner transparent to the applications themselves;
2. a high-level language interface to QTAM facilities;
3. automatic restart facilities in the event of abnormal terminations due to specified conditions;
4. the flexibility to reconfigure the application mix both statically and dynamically; and
5. the capability to execute ECAM in multiple regions.

In anticipation of the difficulties of debugging both ECAM and, subsequently, the applications in the on-line mode, it was also decided that ECAM would be able to simulate the telecommunications environment in a manner that would be transparent to the applications and, for most purposes, to ECAM itself. This allowed most of the program checkout to be conducted in a batch mode and significantly reduced the test time which would have been required had testing been done via terminals.

Constraints

Two constraints were placed upon ECAM; no modifications were to be made to OS/360 and all ECAM code was to be totally re-entrant. Though several modifications were contemplated, the limited gains did not in our opinion warrant the risks inherent in performing such modifications. The requirement for invariant code was necessitated by the possibility that ECAM could be executed in several regions at once; this avoided the need to load more than one copy of the code.

The choice of MVT, rather than MFT, for the ECAM operating environment was dictated by our

desire to (1) dynamically manipulate application task priorities, (2) recover from abnormal terminations and (3) allow the asynchronous execution of the operator interface modules with the balance of ECAM.

It was also our contention that the potentially significant main storage savings resulting from the use of ECAM could justify an installation's move from MFT to MVT.

STRUCTURAL CONSIDERATIONS

The assumption that the main storage requirements of applications will frequently exceed region size was implicit to the design of ECAM. At first, we considered anticipating storage overrun by examining the main storage queues (PQEs, DQEs, SPQEs, etc.) of the OS/360 main storage supervisor, but this was rejected both because it would have made ECAM too sensitive to internal OS changes, and because it would have been a duplication of the OS effort. Instead, we decided to maintain a summary counter of the main storage in use by the ECAM complex at any given instant. This information, coupled with knowledge of the region size and the storage requirements of every application, parameters which are contained in ECAM control tables, enabled ECAM to avoid making requests (e.g., attach an application) which would obviously result in exceeding region size. However, this approach did not account for storage loss in the region due to fragmentation, thus making some abnormal terminations due to out-of-core conditions inevitable. It did, nevertheless, reduce their frequency to a more acceptable level.

This inevitability of abnormal terminations and the necessity to recover from them dictated our design of the ECAM intertask relationships and led to a distinction between *task* and *queue* priorities. Under MVT, the abnormal termination of any task also results in the abnormal termination of all of that task's subtasks as well. Thus, were ECAM to abnormally terminate when attempting to attach an application subtask (an implicit request for main storage), the entire ECAM complex would be terminated, including all currently executing application tasks, since the ECAM task is the highest-level task in the ECAM task hierarchy. In order to prevent this, the ECAM task always attaches an intermediate task (IT) whose sole functions are to attach and detach, when appropriate, the application subtask and to provide a two-way communications path between ECAM and the application subsubtasks. The code required for IT amounts to only 200 bytes and, since the procedure itself is re-entrant, only one copy of the procedure is ever required, regardless of

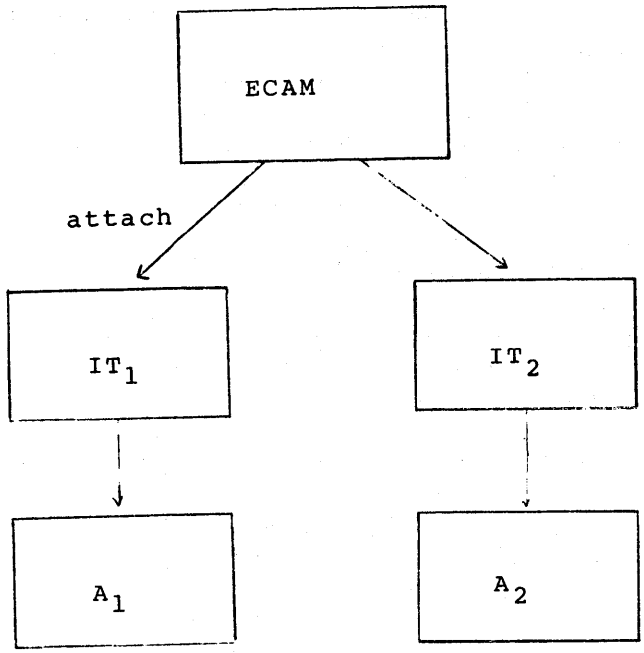


Figure 1—ECAM task hierarchy

the number of subtasks generated. The ECAM task hierarchy can then be represented by the tree structure shown in Figure 1.

The structure illustrated has resulted from the arrival of two messages, one for processing by application A_1 and the other for processing by application A_2. When a message is read from an input queue, it is examined to determine the name of the application which is to process it. ECAM then creates an intermediate subtask and passes to that subtask both the name of the processing program which is to process the message and the location of the message. The IT task, in turn, creates the task for the application program and passes the message on to it for processing. ECAM then continues reading input queues until either region storage is saturated or the input queues are exhausted.

Empty input queues

In the instance when there are temporarily no more messages to be processed, ECAM sets a timer interval (the value of which can be altered by the installation) and enters a wait state for its duration. Though we would have preferred to wait on a list of events (the events being the arrival of additional messages to input queues), with any one event satisfying the wait condition, the implementation of QTAM prevented this.

QTAM provides the user with two options for the

"no message" condition when requesting another message. The first option is to simply wait for a message to appear on the specific queue for which the request was directed. This alternative was unacceptable to us, since ECAM controls the reading of all queues and since the queue to which the next incoming message would be directed was not predictable.

The second option provides for the transfer of control to a user exit routine on the "no message" condition, and it was this option which we chose to implement. The exit routine requests a message from the next input queue, again specifying the same exit routine, with the sequence repeated until all queues have been read without success. It is at this point that the timer interval is set. On its expiration, the entire process is repeated, including, if necessary, the setting of another interval, until a message arrives at one or more of the input queues. The primary advantage of the algorithm is that it allows for background processing when the communications subsystem is idle.

One additional problem was encountered here, although this time the situation was attributable to the OS implementation. OS/360 provides no means of passing input parameters to the timer exit routine in a *re-entrant* fashion. Thus, the timer exit routine itself had to establish addressability to the mainline ECAM control queues. This was accomplished by using the same searching technique as used by the OS supervisors, namely, by starting with the top of the OS task hierarchy (located via the "Old/New TCB" address vector in the Communications Vector Table) and searching for a Task Control Block (TCB) which points to a Program Control Block (PRB) containing the name "ECAMn". The n is used to distinguish between multiple instances of ECAM when the control program is being concurrently executed from several regions. When found, registers are loaded with the appropriate parameters from the register save area in the PTC. That the save area contains the correct parameters is guaranteed because ECAM never alters the general purpose registers containing them after initialization. The structure searched is illustrated in Figure 2. Normally, the first TCB encountered will be the correct one since the "Old" pointer represents the currently executing TCB, namely that for ECAM, which is executing the search.

This search is the only OS-dependent routine in all of ECAM. However, its use was fully justified because the structure is basic to the design of the entire Task Supervisor of OS/MVT. Should the structure ever be altered so also will much, if not most, of the supervisor. As it happens, TCAM[6], a new IBM access method to be released shortly, will eliminate the need for the timer exit routine.

Task priority

One of the reasons that MVT was chosen as the operating environment was so that ECAM could control the assignment of task priorities within the communications subsystem. Exclusive of the QTAM Message Control Program, which runs in a separate region with the highest user priority in the system, there are three levels of priority within ECAM. The highest level is reserved for the exclusive use of the Operator Interface Task (OIT), the next lower level for the ECAM mainline task and all lower levels for application tasks.

The OIT is the primary vehicle for effecting installation control over the communications environment. All operator-ECAM communication is performed via the primary operator's console, utilizing the Write-to-Operator and Write-to-Operator-with-Reply (WTO/WTOR) facilities of OS/360.[3] The Operator Interface Task was assigned the highest priority in ECAM complex to assure rapid response to installation requirements. The task consists of one overlay module with one 1158-byte root segment, which remains permanently resident, and seven transient segments which average 1100-bytes each. Only one transient is ever in main storage at any time and only when required. Their primary purpose is to interpret operator requests and schedule those requests for execution by ECAM.

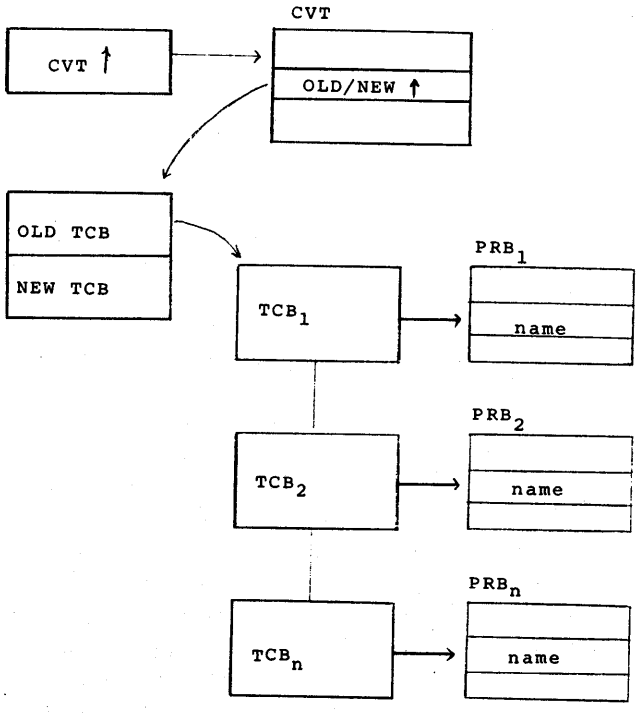


Figure 2—Addressability search

Application task priorities are assigned from an ascending relative scale of from 1 to 13 and can be assigned either statically, prior to ECAM initialization, or dynamically by the operator. The ECAM task is assigned priority level 14 and the Operator Interface Task is assigned priority level 15. All intermediate tasks are assigned the same priority as their related application task.

Queue priority

We felt it desirable to distinguish between task priority and queue priority. The former is normally assigned as a function of the I/O-boundness of a particular task in relation to the other application tasks; i.e., for optimal throughput it is desirable to assign the highest priorities to those tasks which are I/O-bound and lower priorities to those which are more process-bound. However, the priority used for this purpose has no necessary relationship to the *order* in which input queues are serviced. Therefore, each input queue has associated with it a *queue priority*. Initially, it was intended to always return to the highest priority queue after servicing any lower priority queue, but this was rejected (prior to implementation) in favor of cyclic polling of the queues in order to eliminate the possibility of not servicing one or more lower priority queues during periods of heavy message loading.

Input queues are polled in their order of priority. Each queue is queried for the existence of a message and, if none exists or if the queue is "busy," i.e., a previous message from it is still in process, the queue is bypassed and the next lower priority queue is polled. If all the queues are either busy or empty, the timer interval is set and, on its expiration, polling again commences with the highest priority queue.

Assuming queue activity, as many application tasks are activated as possible to the saturation point of the region's main storage allottment. If and when this occurs, queue polling stops and special "quiesce" flags are set in control tables to indicate that a message has been read which cannot currently be processed. When one or more of the in-process tasks complete, the "waiting" message is then processed and the next lower priority queue read.

ENVIRONMENTAL CONTROL

One of the main objectives of ECAM was to provide as much flexibility as possible in controlling the communications environment. Five facilities have been provided which can be used both statically, prior to ECAM initialization, and dynamically, during execution. These facilities are:

—the ability to make application programs resident or transient
—the ability to activate or deactivate specific application processing
—the ability to set and alter application task priority
—the ability to set and alter input queue priority
—the ability to multi-task a particular message type

All five attributes—residence, state, task priority, queue priority and multi-tasking—can be predefined during ECAM generation, utilizing special macros provided with ECAM,[1] or dynamically, via the Operator Interface Task. The time-of-day at which the commands are to take effect can also be specified by either means. A full description of the macros and the command language, which conforms to the OS/360 operator command format, can be found in References 1 and 2.

Residence attribute

During ECAM initialization, the control tables which have been defined by the installation via the macro facility are loaded into main storage and the initial program attributes are examined. All programs which have been marked resident are loaded and permanent intermediate and application tasks are created for them, regardless of current input queue activity. Tasks for transient programs are created only when needed and detached (destroyed) when the storage they occupy is required by another transient program. Transients may subsequently be made resident by the operator and resident programs can be made transient.

It should be observed that the special case wherein all applications are resident is similar to the normal QTAM organization without ECAM. The great difference with ECAM, however, is the ability to dynamically adapt the operating environment to the changing communication situation. For example, it is quite common to find several applications which have one or more abnormally high peak activity periods during the day. During such periods, which very often can be anticipated, the processing programs for those types of messages can be made resident, or given a higher priority, or both. This cannot be done if the applications run in separate regions in a normal QTAM environment.

State attribute

The state attribute provides the capability to shut down or turn on the processing of certain message types. There are several situations where this may be desirable:

1. when there is no activity during periods of the day for these programs; and
2. when the applications being shut down are of the data collection variety (i.e., requiring no response) and the main storage is needed to flatten out peak activity in other application areas.

Task priority attribute

In situations where the mix of application tasks which are active changes frequently in some definable pattern—such as a situation where applications A, B and C are active all day, but D and E are active only from 1:00 to 5:00 P.M.—it may be desirable to alter the priorities of some of the tasks to reflect the new I/O-process ratios. The change of priority is effected by ECAM via the CHAP (Change Priority) supervisor service call of OS/MVT.[3]

Queue priority attribute

As mentioned previously, this attribute determines the order in which the input queue is queried relative to all the other input queues. When a request is made to alter the priority of an input queue, the ECAM control table linkage is resorted to reflect the new polling order.

Multi-tasking attribute

This feature provides the ability to concurrently process messages of the same type. Multiple tasks are created by ECAM for all applications with this attribute. Although not necessary, it is desirable to code the application in a re-entrant manner so that only one copy of the module is required. Serialization control is provided by ECAM on input queues containing segmented messages, but greater efficiency is possible if messages are not segmented.

Other facilities

Two other facilities were incorporated into the design ECAM. The first, the *Test* attribute, will be discussed under environment simulation later. The other feature is the capability of having ECAM control the opening and closing of the application's non-communications files as well as the communications input and output queues. This capability is particularly important for the transient applications, since the OPEN and CLOSE functions for auxiliary disk and tape files can contribute as much as one or two *seconds* to the total response time in inquiry/response applications. If these functions can be separated from the application itself, these files need only be opened when the application is activated and closed when deactivated—not every time the application is loaded into or leaves main storage. A secondary benefit may be a reduction in both disk arm contention.

ABNORMAL TERMINATION RECOVERY

There are two main classes of abnormal terminations which may occur: those which result from ECAM requests for more *contiguous* storage than is currently available in the region, and those caused by the application programs themselves.

Storage requests

As was indicated earlier, terminations due to excessive storage requests are expected by ECAM. The amount of available space left in the region is always known by ECAM and, therefore, a request which exceeds the total available at any instant is never made. However, we chose not to keep track of the fragmentation of the region, which is caused by applications of varying sizes, since this is a normal OS/360 function.

Such terminations occur when the Intermediate Task attempts to attach the application subtask, and ECAM detects it from the condition code which is returned by the operating system to the "mother" task of the terminating subtask. In this case, ECAM is the "mother" task and the Intermediate Task is the abnormally terminated task. It should be noted that at this point a message has already been read into main storage. ECAM saves the location of the message and sets both a termination and quiesce flag, the former to indicate that the application for the message in question is the next to be executed when storage becomes available, and the latter to indicate that transient applications are to be removed from storage, one by one, as they complete processing of their current messages, until such time as enough storage becomes available. As each transient completes processing, it is detached and another attempt is made

to attach the application which started it all. If another abnormal termination results, the entire process is repeated. Eventually, it is guaranteed that enough storage will become available, although it may be necessary to quiesce every transient.

The installation can minimize the effect of fragmentation by designing and coding applications of uniform size. If necessary, modules can be padded with blanks if they are under-sized, or link edited in overlay with equal-size segments. If this is not possible in all cases, application sizes should be in exact multiples of each other, for example, 40K, 80K and 120K.

Application terminations

Some application terminations are inevitable and although most cannot be anticipated, some can. For those that can, ECAM allows the installation to specify for each application up to four abnormal condition codes, upon the occurrence of which the application will be restarted. Abnormal termination due to an "out-of-core" condition (CC=80A) is automatically considered a restart condition. In all cases of abnormal termination, the operator is notified of the condition. Those applications which terminate due to non-restartable conditions are flagged and made inactive. A subsequent request by the operator that day to reactivate the application will result in an "Not Restartable" error message.

ENVIRONMENT SIMULATION

One of the great disadvantages of debugging in an on-line environment is that it requires input from a terminal, or set of terminals, which may be quite remote from the computer center. This cannot only be costly, but a very slow and painful process as well. In order to facilitate the development of application programs, ECAM provides the installation with the ability to debug programs using sequential input from tape or disk, while checking out the ECAM interface as well. This state is communicated to ECAM by indicating, via the macros provided, that the program is in a *Test* rather than *Production* state. When the program is activated, ECAM opens a QSAM file rather than a QTAM input queue and a SYSOUT printer file for output messages. All of this is completely transparent to the application programs. Later, when the programmer feels that the application is bug-free, all that is necessary to do to make it operational is to reset the *Test/Production* attribute flag.

HIGH-LEVEL LANGUAGE INTERFACE

ECAM was developed primarily for COBOL installations. The move from a batch to an on-line operation utilizing QTAM requires either that processing programs be written in assembly language or that COBOL interface modules be written in assembler language to provide the necessary QTAM I/O functions. ECAM provides these interface facilities. The standard OS/360 calling sequence is used so that the application can be written in any OS-supported language, including FORTRAN and PL/I. In addition, in order to facilitate the processing of messages by programs written in COBOL, the record/segment length field in message headers is converted from a binary to a decimal value on input and the reverse way on output. The specific interface is detailed in Reference 1.

OPERATIONAL CHARACTERISTICS

ECAM has been operational since early 1969 in a large aircraft industry installation, running in a 114K region on a 360 Model 50, using 2314 disk storage, which is shared by two CPUs. The number of processing programs has increased from the original three to ten, all of which are inquiry/response applications, with nine programs averaging 40K in size and the tenth 80K. Approximately 4,000 messages are handled daily with a considerable amount of background work, though background throughput is degraded. Response times vary from 3 seconds or less to a worst case of several minutes. The average response time, however, is within 3–7 seconds for most applications.

It should be noted that in the absence of ECAM or some similar control program, the ten processing programs, currently running in 114K, would have normally required 440K of main storage were they to operate concurrently. The main storage overhead required for ECAM itself is approximately 12K, with no applications specified, and approximately 15K, with six applications being driven by six input queues.

Future modifications are currently under consideration, but a careful analysis of ECAM behavior under varying conditions is necessary before they are implemented. Among those modifications under consideration are the implementation of a priority input queue polling sequence, the addition of a data base language facility and inter-region communications facilities. It is also anticipated that ECAM will be upgraded to operate in a TCAM environment. One primary advantage of doing so would be the elimination of the timer exit routine, as TCAM provides a multiple-wait facility.

In conclusion, I must add that perhaps the greatest personal reward we had in designing and implementing ECAM is that it did all it set out to do. There is little that is revolutionary about ECAM; it was designed to plug a known gap in communications processing and this it appears to have accomplished. It is to be hoped that "fourth generation" software will automatically provide most of these facilities.

ACKNOWLEDGMENT

Without the efforts and dedication of two colleagues, ECAM could not have been implemented. The author is indebted to Mr. Richard Dempsey, of Programming Sciences Corporation (PSC), for the implementation of the operator interface modules, some seven in all, and for the generation of the user macros. I must also thank Mr. Richard Loveland, also of PSC, for his evaluation studies and subsequent improvements to ECAM, as well as for his most welcome help in the generation of test data. To both, I must extend my gratitude for their aid in both the intermimable debugging process and in producing the documentation for the system, much of which was used as input to this paper.

REFERENCES

1 *ECAM programmer's guide*
Programming Sciences Corporation
2 *ECAM operator's guide*
Programming Sciences Corporation
3 *Operating System/360 supervisor and data management services*
IBM Corporation Form no GC28-6646
4 *OS/360 queued telecommunications access method message control program*
IBM Corporation Form no GC30-2005
5 *OS/360 QTAM message processing program services*
IBM Corporation Form no GC30-2003
6 *OS/360 planning for the telecommunications access method (TCAM)*
IBM Corporation Form no GC30-2020

Programming in the medical real-time environment*

by N. A. PALLEY, D. H. ERBECK and J. A. TROTTER, JR.

University of Southern California School of Medicine
Los Angeles, California

INTRODUCTION

The Shock Research Unit (SRU) is a specialized clinical research facility developed for the triple purposes of rendering intensive care to seriously ill patients, studying underlying mechanisms of the disease process of circulatory shock, and developing new technologies for evaluating the status and treatment of seriously ill patients.

A computer system for monitoring patients was developed at the SRU in 1963.[1] Based upon the experience with the initial system over a five year period, new specifications for a system were developed.[2] This new system has been implemented and is now in routine use in the two bed Shock Research Unit at Hollywood Presbyterian Hospital in Los Angeles. Figure 1 shows one of the instrumented beds. The major goal of this research project is to automate the critical care environment and, since the facility is supported by a federal research and development grant, patients are not charged for services.

The computer must simultaneously acquire data, control many processes in the ward and manage the retrieval and display of information on both the current and prior status of the patient. The adoption of the general purpose digital computer as an accepted tool in clinical-medical practice has lagged far behind the predictions of several years ago. Hospital information systems, including off-line analysis of patient data, still constitute the major applications of computers in clinical facilities.[3] Most patient monitoring in the recently developed specialized Coronary Care and Intensive Care Units, is performed by special purpose analog devices.

As in many other computer applications areas, there is an ample selection of available digital hardware adequate to the patient monitoring task. Interface hardware including transducers, pre-processors, automated chemical analyzers and display devices of the required capability and reliability are now becoming available. The software interface remains the major barrier to clinical acceptance of the digital computer. It is not the programmer who will utilize patient monitoring system, but the physician, nurse, and laboratory technician. They may not appreciate the computer system's complexity nor forgive its idiosyncracies.[4]

A major effort at the Shock Research Unit has been to make the computer system transparent to the clinical staff and to the medically-trained occasional programmer. In all interactions between the clinical staff and the computer system elaborate and unnatural coding schemes are avoided. All instructions, lists, and alternatives are displayed by the computer at the bedside with computer jargon replaced by medical jargon. Often computer system efficiency must be traded for this improvement in the physician's or nurse's understanding and acceptance of a procedure.

General system description

The primary monitoring functions are accomplished by analog transducers attached to the patient which directly sense physiological activity.[5] The resultant electrical signals such as ECG, arterial and venous pressure wave-forms, are amplified and displayed at the bedside on a multi-channel oscilloscope. Analog signal conditioners perform some pre-processing including the derivation of the heart rate and detection of the R-wave from the ECG signal, and of respiration rate from the

* The research programs of the Shock Research Unit are supported by grants from the John A. Hartford Foundations, Inc., New York, and by the United States Public Health Service research grants HE 05570 and GM 16462 from the National Heart Institute and grant HS 00238 from the National Center for Health Services Research and Development.

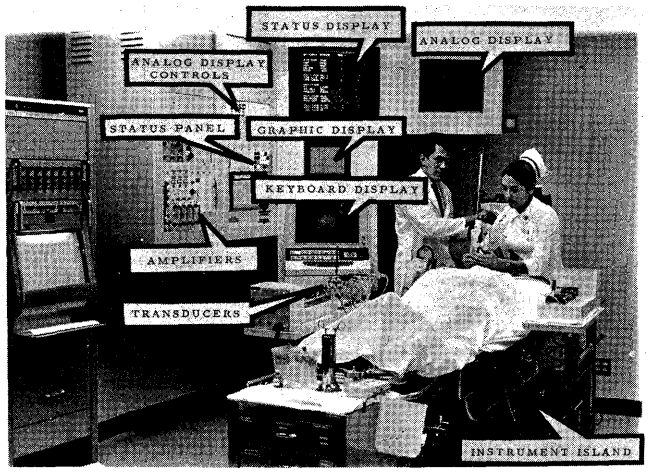


Figure 1—An instrumented bed in the Shock Ward showing the placement of displays and controls

venous pressure signal. The outputs of these conditioners and amplifiers are passed to a multiplexer and A/D converter. In some cases, as in the reading of temperatures, a single point analog read suffices and the digital processing consists in multiplying the converted voltage by the proper factor. The derivation of other parameters such as systolic and diastolic pressures, and left ventricular dp/dt involve more complex digital procedures. The output of laboratory test devices, now in various stages of automation, for the monitoring of blood chemistry values, such as PO_2, PCO_2 and pH, are also input as analog signals. Another category of monitoring functions includes cardiac output[6] and blood volume determinations, the latter being calculated from the dilution of radio-active tracers in successive patient blood samples.

Automatic monitoring programs are run at a frequency which is a property of the particular program and varies from once a minute for heart rate to once every 15 minutes for temperatures. Other programs, such as those involved in the determination of cardiac output, are called up by use of the keyboard as needed.

Monitoring, analysis, and display of current patient parameters is in itself an important task. However, if this were the only function the system were called on to perform, a sufficient number of special purpose analog devices would serve as well. The ability to store the monitored data in highly structured form, to retrieve, manipulate and display the stored data in a variety of modes and post facto rearrangements provides one of the justifications for the use of a digital system.

An elaborate alarm system is under development which makes extensive use of the patient data file.[7] The system employs multivariate statistical techniques to examine the simultaneous values of seven physiological variables and compares them to equivalent sets monitored 5, 15, 30 and 60 minutes previously. Estimates are calculated on the probability of occurrence of these sets of values and changes, and the system reports unusual changes to the ward staff. In addition, critical processes, such as the infusion of fluids and medication, may be automatically halted.

The patient monitoring system, as implemented, uses an XDS Sigma 5 computer with a core memory size of 24K, 32 bit words. This system utilizes standard XDS peripherals including digital I/O, A/D converter, D/A converter, a 3 million byte fixed head disk drive (RAD), 2 seven track tape drives, line printer, card reader and 5 keyboard displays. Other devices include an incremental plotter, storage oscilloscopes, and an alphanumeric TV display system. Those devices which allow information to flow between the ward and the computer are shown schematically in Figure 2.

Data are obtained from the analog and digital inputs, and from the keyboard displays. Once collected, processed and stored in a patient's file on the disk, they are retrieved and stored or displayed on a variety of devices serving distinct purposes. Depending upon the device, this is done either automatically (scheduled) or upon request. Most communication between the medical staff and the computer is conducted through the keyboard displays. From these devices the ward staff can start monitoring procedures, store textual data, call for computation and analysis of the patient data, and

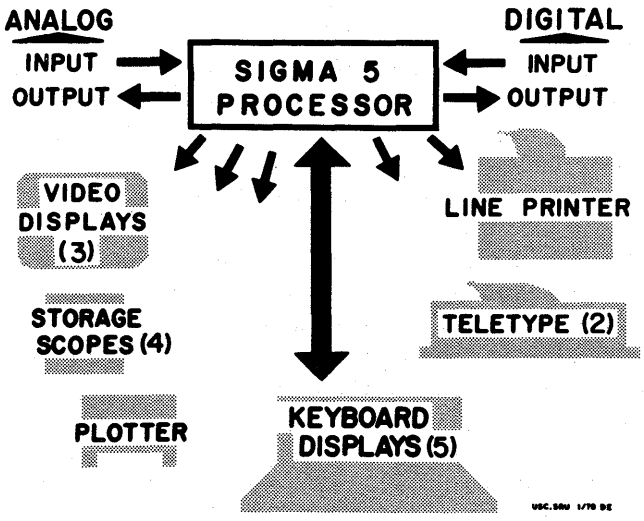


Figure 2—Schematic diagram of the devices which provide communication between the ward and the monitoring system

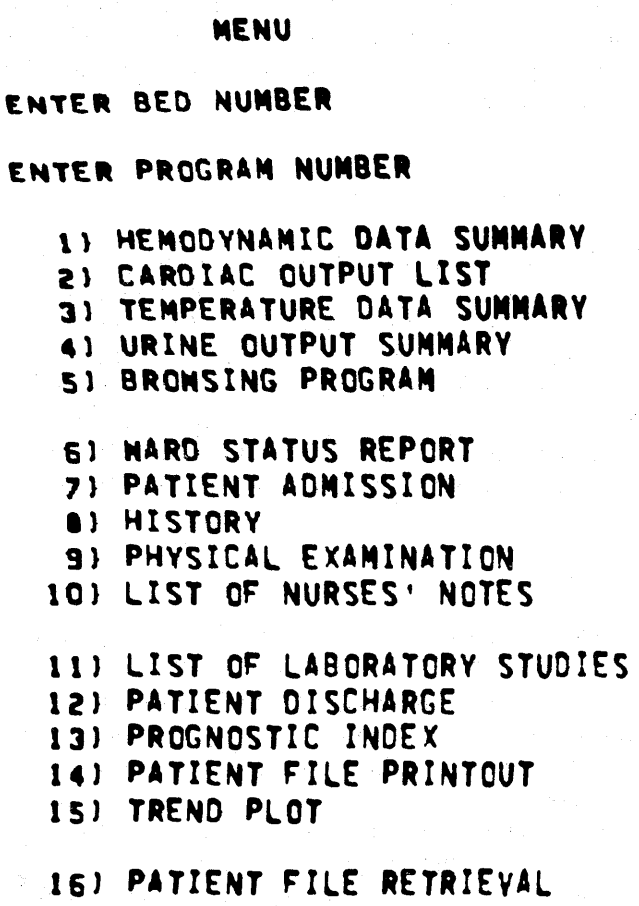


Figure 3—Keyboard display output showing the highest level list of available summaries and procedures

retrieve information from old patient files. A special function key loads a program to display a list of the most commonly used programs, as shown in Figure 3. The user then responds with his choice. The program selected may display data from a file, or present the user with another list representing a lower level of choices. Selections from these lists are indicated by entering the item number. The physiological status of each patient is displayed above his bed on a large screen TV monitor driven by a character generator. This display includes the current values of the most important monitored variables.

SRU subsystem facilities

The SRU Patient Monitor is a library of applications programs, written in FORTRAN, functioning within a comprehensive executive. The executive comprises the manufacturers real-time monitor, RBM* (Real-time Batch Monitor), and a subsystem which has been developed to provide dynamic scheduling and loading of program modules, flexible control of analog input, data management and a variety of interfaces to I/O devices. The primary requirements of a medical real-time system as they have been met by the SRU subsystem are discussed briefly below.

Program scheduling

The subsystem includes a program scheduler which is capable of initializing tasks at timed intervals asynchronously with other tasks and without operator intervention. Although this removes the responsibility for sequencing and program initiation from the applications programs, one applications program may start another as discussed below. The number of foreground processes which may be active simultaneously is limited only by the total core requirements of those programs, since the system program control tables may be set arbitrarily large at system generation. The program scheduler utilizes one of the Sigma hardware clocks and the interrupt structure which allows external interrupts to be triggered internally.

Dynamic relocation of programs

Since the SRU Patient Monitor must be able to load program modules quickly and provide an efficient use of core memory, object code for each task is stored on the disk with all references satisfied, yet with the capability of being dynamically loaded using a relocating loader. There are no fixed partitions; rather, core is resegmented as each relocatable load module is found on the disk, and loaded into the first available space that will contain it. Applications programs, which may be as large as 4K words each, are not re-entrant; fresh copies are loaded as needed, and remain in core until execution is completed, or until the program is released by the user in the case of interactive keyboard display programs. The relocating loader is effectively limited by I/O speed, since modules are stored in core image form, and need only to be relocated. There is a public library of FORTRAN systems routines to lessen the core requirements of individual modules and to speed their loading.

* A second version of RBM has recently been released with a RAD editor which facilitates creation and maintenance of disk files. It also permits foreground tasks to be loaded and run in response to external interrupts. Conversion to the new monitor is in progress.

Dynamic user control of execution priority

A variety of tasks must be concurrently served by the same supervisor. These vary widely in complexity and rate of execution, in dependence on I/O, and in relative importance to the welfare of the patient. The system allows the priority of execution of these tasks to be dynamically altered as a function of the type of processing, and as a function of the patient's condition as determined by analysis programs, or by clinical staff. Since dynamic priority reassignment is allowed without linking tasks to any particular interrupts, it is necessary to provide for the capability of queuing multiple programs at each priority level. The four priority levels available to the applications programs are utilized to maximize I/O and computation overlap, and minimize keyboard display response time, while permitting timing to 1/500th second accuracy within programs for process control applications.

Scheduling and interleaving of real-time I/O

One of the major problems of real-time medical applications is the handling of long duration, low frequency analog input requests. The frequencies and duration of these inputs vary from signal to signal. A scheduling structure is provided for the concurrent input of analog data for multiple analysis programs and the processing of data. The analog scheduler samples only those channels requested, rather than continuously sampling all 96 analog input lines. Data is stored directly into the half-word buffer arrays defined by the program initiating the analog request. Programs are inactive during analog input, keyboard display I/O, and other special systems operations. The applications programmer may segment large tasks in order to minimize core requirements during analog sampling of long duration, using the RAD for temporary storage of arrays. The analog scheduler uses dedicated clocks and interrupts for its operation.

Data management

Patient management in the critical care environment requires on-line sequential and random access to large patient files containing textual and numeric information. The components and the organization of these files vary with changes in the monitoring requirements. Consequently a method has been devised to associate a unique description (or outline) with each file which identifies the set of elements contained in that file. The outline assists in locating information whenever it is to be retrieved from the file. The patient file management system serves to link the separate processes of data acquisition, data analysis, and data display, as well as providing for permanent storage of all information collected.

Systems interface routines

The use of a familiar higher level language simplifies the writing and modification of applications programs. Communication between the FORTRAN applications programs and system functions is provided by a set of system interface routines, the capabilities of which are detailed in Table I. They are written in assembly language, but are directly accessed through FORTRAN subroutine calling sequences in applications tasks.

System parameters

The XDS monitor occupies 5K words of core. Our schedulers, handlers, file management system, buffers (currently set for two beds, and five keyboard/crt displays), special systems interface routines, and the FORTRAN public library require an additional 8K of the 24K words of core. Sufficient core (9K) is reserved for compiling and running background FORTRAN programs, so that program development and off-line statistical analysis can proceed during patient monitoring. However, background processing may be checkpointed during periods of heavy foreground use. Applications programs have access to the background area plus an additional 2K. Applications programs are added to the library only after thorough testing, using recorded physiological signals when necessary, so that program integrity is assured prior to actual use. Initial debugging of new programs and addition of the programs to the library is carried out in the background mode, but monitoring must be discontinued while the updated library is actually being loaded onto the disk.

Between ten and twenty percent of CPU time is consumed by the analog and execution schedulers, which run 500 times/second. The overhead applicable to a particular program depends on the priority level of that program, and upon the mix of analog input and computation time in simultaneously executing programs.

Applications programs

The following description of the hemodynamic monitoring programs, FUZ1 and HEMO, will be used to illustrate some of the unique capabilities afforded the FORTRAN application program by the SRU system.

TABLE I—System Interface Routines

SUBROUTINE NAME Function	Parameters
ANA Analog Input	1) Interval between samples (in 1/500ths second) 2) Number of samples 3) Present sample index 4) Wait/return code & buffer size 5) Number of channels 6) Array of channel numbers 7) Input data buffer 8) Return priority
ANAOUT Analog Output	1) Output control 2) Channel number 3) Voltage
AQUIT Analog Input halt	
CCIWR Status display write	1) Format statement number 2) , . . . , *n*) List of output variables
DELAY Precision delays	1) Time delay in 1/500ths of a second
DELETE Delete scheduled program	1) Bed number 2) Name of program to be deleted 3) Error return
DIGIN Digital Input	1) File or bed number 2) Array of words corresponding to digital input lines
DIGOUT Digital Output	1) File or bed number 2) Line number 3) On/off code
GETDAT Patient file data retrieval	1) File or bed number 2) Summary name 3) Time desired or position code 4) Number of values requested

SUBROUTINE NAME Function	Parameters
	5) Names of the values 6) Array for Data 7) Indicator for textual information 8) Error return 9) Intercall location pointer
IWAIT Wait for external interrupt	1) Number of interrupt to be armed 2) Return priority
KDRD Keyboard/display read	1) Format statement number 2) Parameter being read
KDWR Keyboard/display write	1) Format statement number 2) , . . . , *n*) List of output variables
PINIT Start program	1) File or bed number 2) Name of program to be started 3) Time between executions (+), or interval before starting once (−, 0) 4) Multipurpose variable passed to program
PRIOR Set program priority	1) Desired priority
PUTDAT Store patient data	1) File or bed number 2) Summary name 3) Time to be stored with data 4) Number of values being stored 5) Names of the values 6) Values to be stored 7) Indicator for textual summaries 8) Error return
TIME Get or set time	1) Get/set indicator 2) Time of Day
TTWR Write on ward teletype	1) File or bed number 2) Format statement number 3) , . . . , *n*) List of output variables

HEMO reads arterial, venous, and pulmonary arterial pressure waveform data, as well as the outputs of the electrocardiogram (ECG) preprocessor.

From these primary signals, 15 measures are derived and stored in the patient file. Other programs retrieve and display this information automatically and on demand. This monitoring program normally runs once each five minutes (Normal mode) but optionally may be run once each minute (Acute mode) or be suppressed entirely (Wait mode) under bedside control.

The hemodynamic program is quite large, and any combination of primary signals may not be available at the scheduled initiation of the program. Thus, in order to avoid loading the program unnecessarily, it is not started directly by the program scheduler (PSKED). Instead, a small trigger program, FUZ1, the listing of which is shown in Figure 4, is scheduled to run once each minute.

Applications programs as started by PSCHED are not assigned to an execution priority queue. They are assigned a priority by a call to the resident subroutine PRIOR which is included as one of the first executable statements in the program. The nominal availability of the primary signals and the *Normal/Acute/Wait* information is obtained from digital inputs controlled by an array of switches at the bedside (the "status panel"). FUZ1 first reads the state of the switches into an array, S, through a call to

```
1          SUBROUTINE FUZ1 (NBED, IDUM)
2          IMPLICIT INTEGER (A-Z)
3          DIMENSION S(13), BUFF(5), IG1(6), IG2(3)
4          EQUIVALENCE (ACUTE, S(2)), (WAIT, S(3)), (AP, S(4)), (VP, S(5))
5          EQUIVALENCE (IG2, FG2), (MZ, ZM), (HR, S(6))
6          REAL FG2(3)
7          REAL ZM
8          LOGICAL ACUTE, WAIT, AP, VP, HR
9          DATA MZ /-1073741824/
10         DATA IG1/'HR  ', 'STAT1', 'STAT2'/
11         IBED=NBED
12         CALL PRIOR (2)
13         NP=10
14         LOCCT=0
15         CALL DIGIN(IBED, S)
16         IF (WAIT) GO TO 7
17         CALL DISAST(IBED, 1)
18         IF (HR) GO TO 20
19         IF (AP .OR. VP) GO TO 30
20         FG2(1)=ZM
21         FG2(2)=99999.0
22         FG2(3)=99999.0
23         GO TO 46
24    7    CONTINUE
25         CALL DISAST (IBED, 2)
26         GO TO 90
27    20   CALL TIME (1, MOM)
28         IT=KHANF (IBED, 11)
29         CALL ANA(10, 10, II, 1, 1, IT, BUFF, 2)
30         CTACH=0
31         DO 1000 J=1, NP
32         CTACH=CTACH+I2F(BUFF, J)
33    1000 CONTINUE
34         CTACH=CTACH*24/16384
35         IF(CTACH .GT. 150 .OR. CTACH .LT. 35) GO TO 65
36    30   IF(ACUTE) GO TO 35
37         MOML=9
38         CALL GETDAT(IBED, 'HEMO', MOML, 1, IG1,IG2, IC, &35, LOCCT)
39         IHR=FG2(1)
40         IF(.NOT. HR .OR. FG2(1) .EQ. ZM) GO TO 60
41         IF(IABS(IHR-CTACH) .GT. IHR/10) GO TO 65
42    60   IF(MOM-MOML .GE. 5) GO TO 70
43         GO TO 90
44    35   IF (AP .OR. VP) GO TO 70
45    45   FG2(1)=CTACH
46         FG2(2)=99996.0
47         FG2(3)=99966.0
48    46   CALL TIME (1, MOM)
49         CALL PUTDAT(IBED, 'HEMO', MOM, 3, IG1, IG2, IC, &90)
50         GO TO 90
51    65   CONTINUE
52         CALL DIGOUT(IBED, 5, 1)
53         CALL DELAY (1)
54         CALL DIGOUT (IBED, 5, 0)
55    70   CALL PINIT (IBED, 'TIMO', 0, MOM)
56         CALL PINIT (IBED, 'HEMO', 0, CTACH)
57    90   RETURN
```

Figure 4—Listing of the applications program FUZ1

DIGIN, with the bed number and the array name as parameters, (Table I lists the calling parameters of all of the systems subroutines for reference throughout this section). If it is determined that the *Wait* button is on, the program exits after calling a subroutine DISAST, which, through a call to the system subroutine CCIWR, writes asterisks after the values of the variables on the status display indicating that they are not current. If *Wait* is not on, any asterisks previously written are erased.

When ECG is available, the heart rate is determined directly by reading the cardio-tachometer output of the ECG preprocessor (Figure 4, line 29). System subroutine ANA is used to obtain analog input. It stores the necessary information into a table used by the analog scheduler and optionally returns control to the application program as a function of the *return* parameter. If this parameter is a +1, ANA will fill the buffer specified with the number of points requested while allowing other programs to execute. If the parameter is negative, ANA will start to fill a buffer equal to the size of the parameter and return control to the application program immediately to allow processing of the data. Where large volumes of digitized data are to be retained, it is possible for the program to essentially double buffer this data and write it out on the disk. The analog scheduler is triggered by a clock interrupt, (presently set at 500 times per second), and the interval parameter specifies the number of 500ths of a second between analog reads.

If the preprocessor indicates a heart rate of less than 35 or greater than 150 beats per minute, an alarm is sounded in the ward, using a sequence of calls to DIGOUT (digital output) and DELAY. Then HEMO would be scheduled by a call to PINIT. If *Acute* is on, the logical array is checked further and if at least one primary signal is available, HEMO is started by the system subroutine PINIT and the trigger program exists (Figure 4, line 56).

Subroutine PINIT is used to schedule other programs from applications programs in execution. The calling sequence includes: the bed to be associated with the scheduled program, the name of the scheduled program, the desired interval between executions, and a parameter through which data may be passed to the scheduled program. A positive value for the execution interval represents that interval in seconds. A zero or negative value specifies the delay (in seconds) prior to running the program once only.

If no primary signals are available, according to the

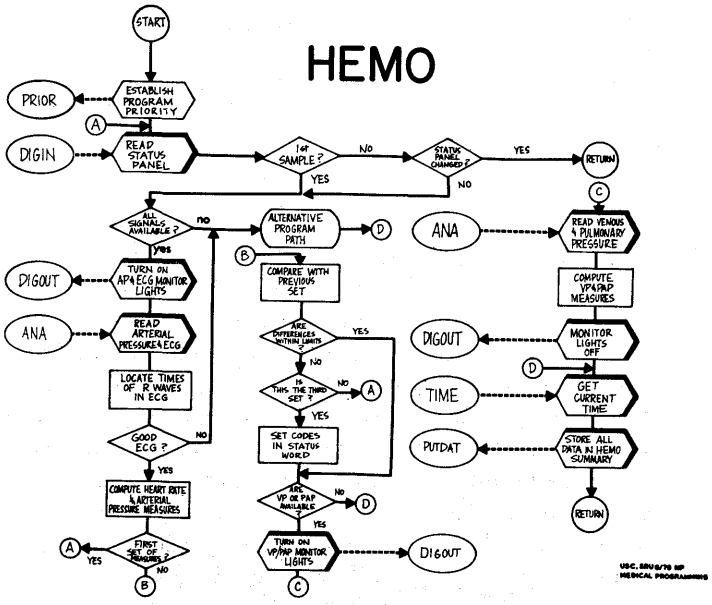


Figure 5—Diagram of data flow in the SRU Patient Monitoring System

status panel, appropriate codes are stored in a hemodynamic summary of the list structured patient file which resides on the disk. To determine the presence and location of information in the patient file, each file is accompanied by an outline which is separate from but descriptive of the file. The flow of patient data is illustrated in Figure 5. This outline serves as a unique table of contents for the particular patient's data. The headings in the outline are the names of the summaries. The subheadings comprise the names of the individual items to be stored in the file under that summary. Figure 6 shows some of the summaries in a typical patient file, as represented in the hardcopy

```
PATIENT # 1057     BED  1   *******
********

HEMODYNAMIC TIME 14/0530

ARTERIAL PRESSURE (MMHG)
  SYSTOLIC        75
  MEAN            65
  DIASTOLIC       59
  DELTA SYSTOLIC   5
  MAXIMUM DP/DT   -0
  MEAN DP/DT      -0
  PULSE DEFICIT    0

VENOUS PRESSURE (MMHG)
  MEAN            13
  DELTA RESP      -0

PULMONARY ART PRESSURE (MMHG)
  SYSTOLIC        -0
  MEAN            -0
  DIASTOLIC       -0

ELECTROCARDIOGRAM
  HEART RATE              97
  MAX R-TO-R INTERVAL     64
  MIN R-TO-R INTERVAL     62
  SD OF R-TO-R INTERVALS   0.0

RESPIRATION RATE  16

********

TEMPERATURE (DEG C) TIME 14/0530
  RECTAL          36.5
  LEFT TOE        31.9
  RIGHT TOE       22.9
  AMBIENT         22.7

********

URINE OUTPUT (ML) TIME 14/0531
  TOTAL OUTPUT   363.4
  LAST 5 MIN       .7
  LAST 60 MIN    30.3

********

CARDIAC OUTPUT TIME 14/0526

  CARDIAC OUTPUT L/MIN        3.41
  BODY SURFACE AREA MSQ          2
  CARDIAC INDEX L/MIN/M       2.09
  APPEARANCE TIME SEC            8
  MEAN CIRC TIME SEC            21
  CENTRAL BLOOD VOLUME ML        1
  STROKE VOLUME ML              35
  HEART WORK KGM/MIN          2640
  RES 1 (MAP-CVP)/CO          1597
  RES 2  MAP/CO               1339
```

```
PATIENT # 1057     BED  1   *******
********

HEMODYNAMIC TIME 14/0544

ARTERIAL PRESSURE (MMHG)
  SYSTOLIC        91
  MEAN            69
  DIASTOLIC       58
  DELTA SYSTOLIC   9
  MAXIMUM DP/DT   -0
  MEAN DP/DT      -0
  PULSE DEFICIT    0

VENOUS PRESSURE (MMHG)
  MEAN             6
  DELTA RESP      -0

PULMONARY ART PRESSURE (MMHG)
  SYSTOLIC        -0
  MEAN            -0
  DIASTOLIC       -0

ELECTROCARDIOGRAM
  HEART RATE             109
  MAX R-TO-R INTERVAL     57
  MIN R-TO-R INTERVAL     55
  SD OF R-TO-R INTERVALS   0.0

RESPIRATION RATE  17

********

TEMPERATURE (DEG C) TIME 14/0540
  RECTAL          36.5
  LEFT TOE        32.2
  RIGHT TOE       22.8
  AMBIENT         22.6

********

URINE OUTPUT (ML) TIME 14/0546
  TOTAL OUTPUT   365.4
  LAST 5 MIN       .6
  LAST 60 MIN    17.5
```

Figure 6—Example of hard-copy patient file output showing several summaries at two different times

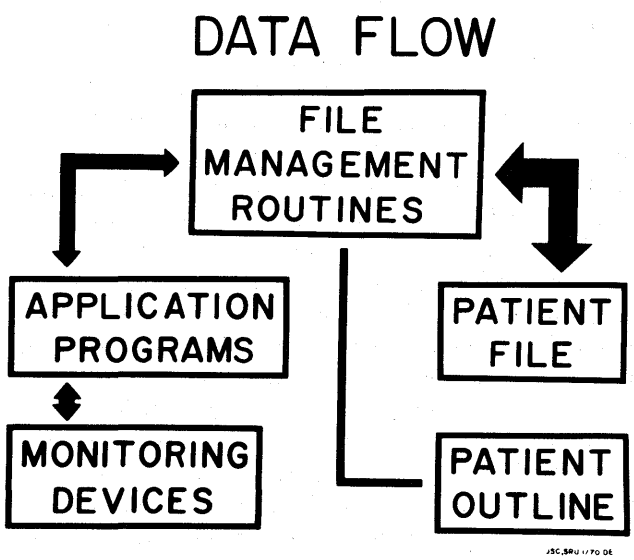


Figure 7—Flow diagram of the applications program which monitors hemodynamic signals

output. Data storage is accomplished by a call to the subroutine PUTDAT (Figure 4, line 49). Symbolic labels for the items to be stored are contained in an array which is an argument in the call, as is the array of corresponding values. The time of day, another parameter, serves as a sequencing element in each summary.

If the *Acute* button were not on (Figure 4, line 36), indicating 5 minute monitoring, the most recent hemodynamic summary would be retrieved from the patient file through a call to GETDAT. The parameters in the call to GETDAT are all analogous to those in PUTDAT, except Time. This parameter may be an actual time of day, in which case the values of the most proximate instance of a summary are returned. Or it may be a code requesting an instance of a summary by its position in the file, (e.g., the first, last, previous, or next instance). The current time of day, obtained by a call to the subroutine TIME, is used to calculate the elapsed time since the last HEMO instance. If this were greater than or equal to 5 minutes, the hemodynamic program would be scheduled; otherwise FUZ1 exits.

Just prior to starting HEMO, the trigger program also starts a time code output program, TIMO. Using calls to TIME, DIGOUT and DELAY, this program generates a series of variable width pulses representing the binary coded decimal 24 hour time. This signal is recorded on a multi-channel analog tape recorder along with the primary physiological signals.

HEMO proceeds as shown in Figure 7. The status panel is checked and the appropriate analog channels

to be read are determined by reference to the bed number parameter with which the program was started. A call to DIGOUT, with a bed and line number, turns on a light at the bedside indicating the signal being monitored. The arterial pressure signal and the R-wave trigger output of the ECG preprocessor are sampled simultaneously for 10 seconds at the rate of 100 samples per second (the R-wave trigger generates a pulse corresponding to each heart beat found in the ECG signal). Data acquisition is accomplished by a single call to ANA which stores 2000 digitized points as 16 bit numbers (half words), in the assigned buffer. On completion of the A/D conversion a second call to DIGOUT extinguishes the monitoring indicator light. The array containing the digitized ECG data is then scanned to locate the heart beats, the relative positions of which are stored in a table. The inability to detect a sufficient number of them in the 10 second sample, or the presence of excessive noise in the ECG signal causes the program to take an alternative path.

Assuming a good ECG, the arterial pressure data points are then scanned with reference to the R-wave table. The maximum and minimum arterial pressure values detected between each pair of corresponding heart beats are stored. Anomalous arterial pulse beats are located on the basis of pulse height criteria and transient values which exceed physiological limits are eliminated. The remaining maxima and minima are averaged and temporarily stored as systolic and diastolic pressures. The entire arterial pressure data array is averaged to compute the mean arterial pressure. The maximum, minimum, average, and standard deviation of the intervals between heart beats is computed from the R-wave table. DIGIN is called again to read the "status panel". If any switch positions have been changed, the program exits. Otherwise an additional 10 second sample of AP and ECG is read and analyzed as above. The results of the two analyses are then compared; if the differences fall within preset limits, the second set of data is retained and the program continues; if not, a third set is read. The new data are either accepted as consistent or, ultimately, are stored in the patient file with a code indicating the inconsistency of the sequential samples.

Venous and pulmonary arterial pressure signals are then sampled by another call to ANA. From these data, mean venous pressure, average pulse height, and systolic, diastolicand mean pulmonary arterial pressures are computed. Throughout the program the values are assessed as to whether they lie within reasonable boundaries. If not, or if the primary signal associated with that variable is unavailable, a value of -0.0 is assigned and an appropriate missing value code is stored in a status word. All of the data, including the status word, are stored in the patient file associated with the bed number. The array of data is stored by a single call to PUTDAT ,as an instance of the summary named HEMO, identified by the current time of day.

SUMMARY

The HEMO program is one of 61 applications programs currently in the library. It is representative of the complexity of processing required in patient monitoring programs. Some examples of other applications programs and their characteristics are shown in Table II.

An additional dimension of complexity is contributed by the requirement of simultaneous monitoring of more than one patient. Thus several programs such as HEMO may be in various stages of execution concurrently. Some of these programs may be executed on a scheduled basis while others are executed upon request of the clinical staff, making it difficult to predict the demands to be placed upon any subsystem of the computer. Simultaneous collection of data from many analog devices, storage and retrieval of data from patient files, and display of information on patient status on a variety of devices with varying transmission rates is a common occurrence. Some of these tasks can be deferred or delayed in their execution, but others with crucial responsibilities, such as controlling the infusion of fluids and medications, cannot tolerate interference and may be required at any time.

Unlike a process control situation where the responses of the monitored activity are well defined, patient monitoring relies significantly upon clinical intervention. Thus when programming a particular application module for such a system, the programmer cannot foresee the computer environment in which execution will occur.

The extensions to the executive system and the interface routines described above free the applications programmer from the logical complexity usually entailed in writing programs for such a multiprogramming environment.

Since the patient monitoring system is simply a collection of independent applications programs, wherein the implications of concurrency are resolved dynamically, the expansion or modification of this library can be done with only minor regard for such concurrency. This is a feature which is pleasing to those who have experienced difficulty incorporating additional features into an operating real-time system. This openendedness becomes a necessary feature in the medical research environment where the system must often be reshaped to respond to changes in care methodology. Finally, the ability to write applications programs in

TABLE II—Characteristics of Selected Applications Programs

	Program Name	Function	Devices/Facilities used	Primary Input Signals	Data Sampling rate (samples/sec)	Execution Interval (minutes)	Derived Variables
1	FUZ1	Schedules execution of HEMO	Patient file system(PFS), digital input (DI), digital output (DO), analog input (AI), program initiation (PI)	Cardiotachometer	50	1	—
2	HEMO	Monitors hemodynamic signals	PFS, DI, DO, AI	Arterial pressure, venous pressure, pulmonary artery pressure, ECG	100	1-5	Systolic, diastolic, mean arterial, mean venous, and mean pulmonary artery pressures, heart rate, pulse deficits, heart rhythm, DP/DT etc.
3	TEMP	Monitors patient temperatures	PFS, DI, AI	Thermistors	10	15	Rectal, left and right toe, and ambient temperatures
4	URIN	Monitors urine output	PFS, AI, DI, DO	Collector tube fluid column height	10	5	Total output, output/5 min., output/60 min.
5	PLT2	Writes 8 variable time-trend plots on storage scopes	Keyboard/display (K/D), PFS, AO, DO	—	—	Demand	—
6	MENU	Displays highest level list of user options	K/D, PI	—	—	Demand	—
7	HEDS	Displays Hemodynamic variables	K/D, PFS	—	—	Demand	—
8	DFIL	Generates lineprinter listing of a patient file	PFS, Lineprinter	—	—	Demand	—
9	CARD	Monitoring and calculation for cardiac output procedure	K/D, AI, AO, DI, DO, PI, PFS, storage scope, ext. interrupts	Densitometer	10	Demand	Cardiac output, stroke volume, central blood volume, peripheral resistance, heart work, appearance time, mean circulation time
10	LACT	Determination of blood lactate	K/D, PFS, TTY, PI	Laboratory determination entered through K/D	—	Demand	Lactate
11	PUMP	Automatic control of pump operation	K/D, DI, DO, AO, PI, PFS	—	—	Demand	—

FORTRAN, aside from the obvious advantage of ease of documentation, encourages the clinician to participate in the development of monitoring algorithms.

ACKNOWLEDGMENTS

The authors wish to express their appreciation to Max Harry Weil, M.D. and Herbert Shubin, M.D., Project Director and Co-Director for their support and encouragement, and to David H. Stewart for his invaluable suggestions. Particular thanks are due Miss Cecilia Pasos for her skill and patience in preparation of the manuscript.

REFERENCES

1 M A ROCKWELL H SHUBIN M H WEIL
Shock III: A computer system as an aid in the management of critically ill patients
Communications of the ACM Vol 9 No 5 May 1966

2 D H STEWART D H ERBECK H SHUBIN
Computer system for real-time monitoring and management of the critically ill
AFIPS Conference Proceedings Vol 33 December 1968

3 J P SINGER
Computer based hospital information systems
Datamation May 1969

4 D H STEWART N PALLEY
Monitoring and real-time evaluation of the critically ill
Invited Paper—Journées Internationales d'Informatique Medicale Toulouse France March 1970

5 H SHUBIN M H WEIL M A ROCKWELL
Automated measurement of arterial pressure in patients by use of a digital computer
Medical & Biological Engineering Vol 5 pp 361-369 1967

6 H SHUBIN M H WEIL M A ROCKWELL
Automated measurement of cardiac output in patients by use of a digital computer
Medical & Biological Engineering Vol 5 pp 353-360 1967

7 S T SACKS N A PALLEY A A AFIFI H SHUBIN
Concurrent statistical evaluation during patient monitoring
AFIPS Conference Proceedings Vol 37 1970

Decision making with computer graphics in an inventory control environment

by J. S. PROKOP

Office of the Secretary of Defense
Washington, D.C.
and
F. P. BROOKS, JR.

University of North Carolina
Chapel Hill, North Carolina

INTRODUCTION

Computer-driven displays have long been thought to help decision making. But the justification for using these devices in decision-making has been long on intuition and short on quantitative analysis. To see if this intuition was right, we conducted an experiment.

Eighteen interested and experienced decision-makers in the inventory control field met for twenty class hours of instruction in advanced inventory control techniques. Near the conclusion of the short course, we measured the participants' decision-making ability while using a computer-driven display device. This measurement was compared to their decision-making ability while using information presented to them on paper.

To provide a vehicle for the investigation, a simulator was written to apply certain inventory control policies to a hypothetical inventory system handling 34 items. This inventory system faces a randomly derived set of orders, price changes, replenishment of stock and other transactions. The simulator has two sets of input parameters: one governs the distributions of transactions; the other establishes the management policy for inventory control. The simulator investigates twelve distinct policies at one time over a simulation cycle of 24 months.

Each participant was given, as examination problems, the statistics resulting from two different simulation runs. Each decision-maker, or *decider*, made a series of decisions; at the end of each simulation month, he ranked the twelve policies in order of desirability. Each decider used printer output on one problem and graphic display presentation on the other.

The statistics from the simulation program were displayed in a variety of formats on standard computer printer paper. The statistics included:

(1) percent availability of stock,
(2) number of purchase orders generated,
(3) lost sales,
(4) total dollar investment in inventory.

The output was in the form of tabular listings and bar graphs. Also, for each problem, an IBM 2250 cathode ray tube display unit was programmed to show the same data as appear on the printer output, in fundamentally the same listing and graphical formats. Thus the experiment did not attempt to establish the relative comprehensibility of listed versus plotted data. Instead it investigated the relative comprehensibility of printed versus display *presentation* of such data, both printed and plotted. The display unit was under the control of the decider, who was able to specify which of the twenty-four months and the kind of displayed data which he wanted to see. In both display and printer output cases, he was expected to start at the first month and proceed through the months in order, with freedom to review.

The course participants were divided into two test groups, each group consisting of nine individuals chosen at random. Tests were made to answer:

1. Are decisions made earlier? The simulated calendar month (out of the twenty-four months of simulation) in which the decider feels he has enough data to commit himself to a ranking for future action was tested.
2. Are decisions made faster? The elapsed clock time to decide on a ranking of policies which the decider would be willing to commit himself to for

future action was tested. The elapsed clock time to complete the remainder of the problem was also tested. After committing himself to a ranking, the participant continued the problem through the twenty-fourth month.

3. Are decisions made more consistently? That is, does the decision made at the month of commitment agree better with that made at the twenty-fourth month, after all data are known?
4. To what degree do the members of each group agree among themselves in regard to the rankings?

In summary, the decider made two basic kinds of decisions. He decided on a ranking at each month in turn, and he decided whether or not to indicate at this month that he felt confident enough of his ranking to recommend that it be used for governing action.

The resulting ordered lists of policy sets and the preparation times were analyzed in two basic ways. First, a 2 x 2 Latin square design with repeated measures was used in an analysis of variance. Rank order statistics were then used to test the consistency of each individual decision-maker and the concordance of a group of decision-makers.

The statistical analysis of variance allocated the observed differences into (1) differences between graphic and printed data presentation (treatment), (2) differences between the first and second problems done by each participant (order), and (3) differences between the performance of the two groups the class was divided into.

The ranking of the policies at the decision month and the ranking at the end of the entire twenty-four month simulation were compared by means of the Spearman

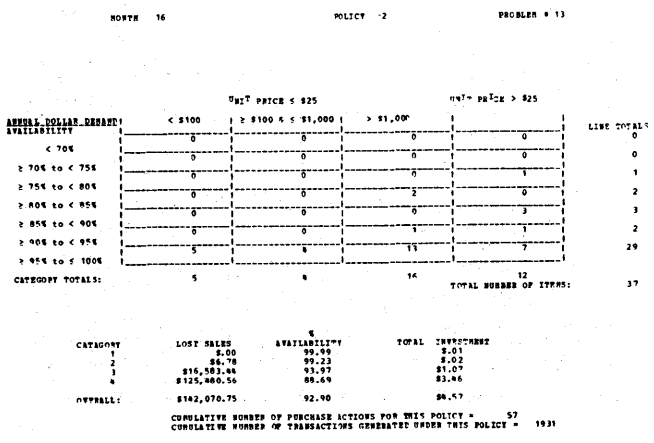


Figure 1—Detailed listing of simulation results by policy within month

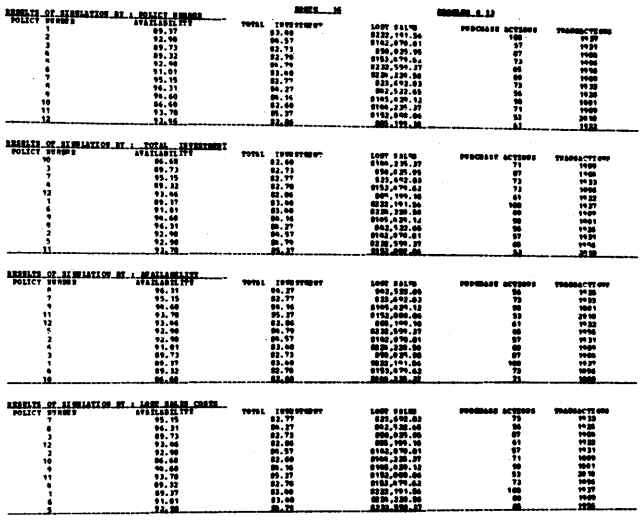


Figure 2—Sorted listing of simulation results by policies within month

rank correlation coefficient. A Kendall's coefficient of concordance was computed to measure how well the rankings produced by each test group on each problem agreed among themselves.

The results may be summarized as follows:

1. The time to make a decision was decidedly reduced in favor of the display presentation.
2. A decision could be made on the basis of less information by using the display device; that is, action decisions were made earlier in the simulated two years.
3. More consistent decisions were arrived at by using this display device. Even though action decisions were made earlier, with less information, using the display device, the decisions made matched better those made with all data available.

THE PRESENTATION TO THE DECISION-MAKER

The simulation statistics for one month are shown in Figures 1, 2, and 3. The total investment figure is scaled by 100,000. A total investment of $423,000 is, therefore, represented as $4.23.

Figure 2 is a table which is ordered by several of the statistics of interest. Figure 3 shows one statistic across all policies as a bar graph. The printed bar graphs are produced monthly for availability, value of lost sales, purchase order activity, and total investment.

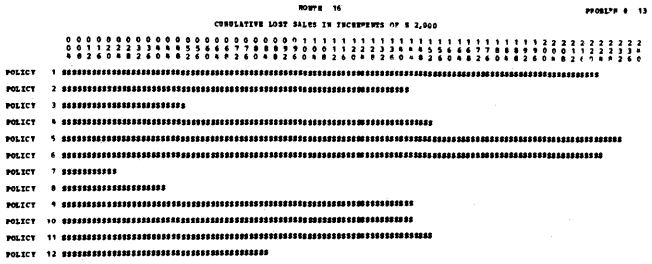


Figure 3—Bar graph of simulation results by month across all policies

Once the printer output was specified, these presentations were mimicked on the IBM 2250 cathode ray tube.

Some exploration of the power and utility of such a display device was made while maintaining similar presentation formats for sake of the experimental design. Figure 4 is the display device representation of Figure 2, for example, and shows some rearrangement of material due to display character and line number limitations. Figure 5 shows the first marked departure from the printer output of Figure 3. The layout is essentially the same, but the presentation is dynamic. A programmed timer advances all lines of the bar graph simultaneously at one second intervals. If the user specifies a month at which he wishes to examine the data, and then specifies the bar graph he wishes to see, the graph begins at month 1 and then advances at one-second intervals to the month requested. The user may alternatively specify half-second intervals. The movement of the bar graph gives a feeling for the

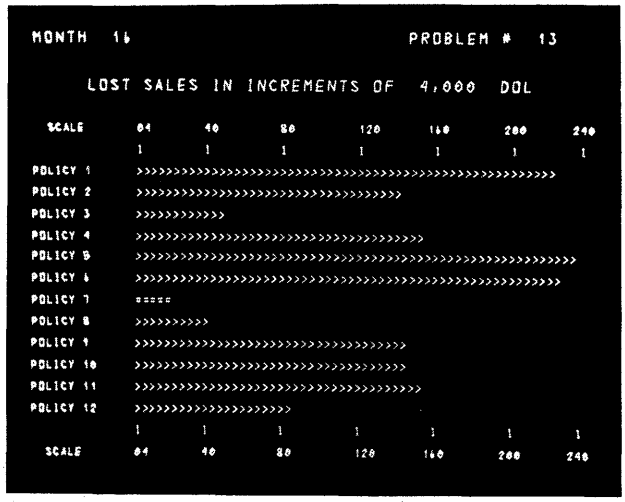
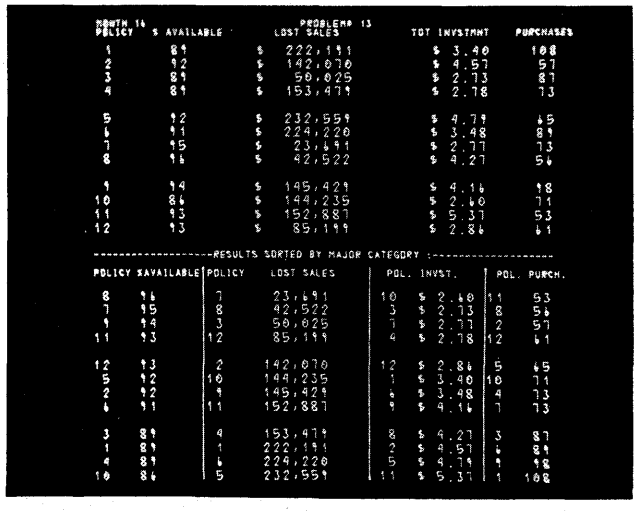


Figure 5—Display of Figure 3

history and current derivative of the statistic under scrutiny. In addition to the movement of the bar graph, the symbols <, > and = are used on the display unit to indicate whether the value of a particular bar has increased, decreased or remained equal in comparison with the previous month.

Figure 6 has no printed correspondent in this experiment. While the data for cost of lost sales, availability, total dollar investment and purchase action activity are available as individual bar graphs as represented by Figure 5, all four of these are available



MONTH 16 PROBLEM # 13

POLICY	% AVAILABLE	LOST SALES	TOT INVSTMNT	PURCHASES
1	89	$ 222,191	$ 3.40	108
2	92	$ 142,070	$ 4.57	57
3	89	$ 50,025	$ 2.13	87
4	89	$ 153,479	$ 2.78	73
5	92	$ 232,559	$ 4.79	65
6	91	$ 224,220	$ 3.48	89
7	95	$ 23,691	$ 2.77	73
8	96	$ 42,522	$ 4.27	54
9	94	$ 145,429	$ 4.16	98
10	86	$ 144,235	$ 2.60	71
11	93	$ 152,887	$ 5.37	53
12	93	$ 85,199	$ 2.86	61

RESULTS SORTED BY MAJOR CATEGORY

POLICY	%AVAILABLE	POLICY	LOST SALES	POL. INVST.		POL. PURCH.	
8	96	7	23,691	10	$ 2.60	11	53
7	95	8	42,522	3	$ 2.13	8	54
9	94	3	50,025	7	$ 2.77	2	57
11	93	12	85,199	4	$ 2.78	12	61
12	93	2	142,070	12	$ 2.86	5	65
5	92	10	144,235	1	$ 3.40	10	71
2	92	9	145,429	6	$ 3.48	4	73
6	91	11	152,887	9	$ 4.16	7	73
3	89	4	153,479	8	$ 4.27	3	87
1	89	1	222,191	2	$ 4.57	6	89
4	89	6	224,220	5	$ 4.79	9	98
10	86	5	232,559	11	$ 5.37	1	108

Figure 4—Display of Figure 2

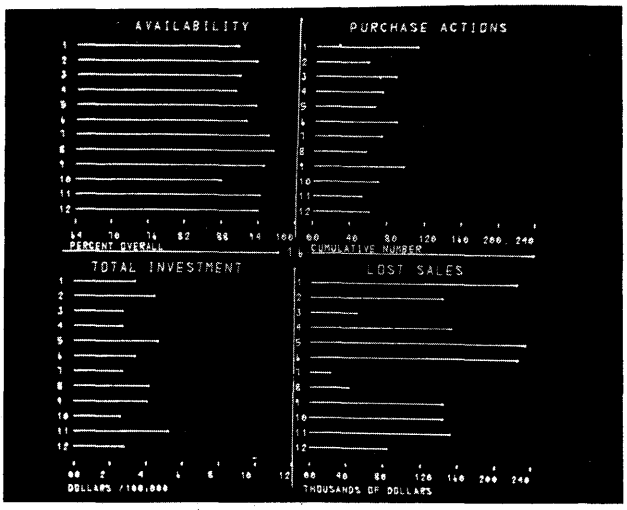


Figure 6—Quadrant graph of simulation results by month across all policies

at the same time in the Quadrant Graph. All bars move under the same program timer control as the individual graphs.

The user indicates by a Programmed Function Keyboard (PFK) to the computer what he wishes to see on the display device. The user is allowed to refer to any *past* month's data. He may not, however, move *ahead* in simulated time by more than one month at a time.

THE SUBJECTS OF THE EXPERIMENT

The experiment was designed to examine the decision-making processes of experienced, practical and interested professional administrators of inventory control. Hence the subjects were solicited carefully. A short course in advanced inventory control techniques was designed as a graduate-level course which would attract only those who were interested in the subject matter and who were prepared to understand the material. The simulation model was the experimental vehicle and was used in examples and problems which were integrated into the course. The participants came to the course in statistical inventory control for their own professional advancement, under the auspices of the firms for which they worked. The participants were not informed of the experiment which was being conducted, and they received the full measure of instruction for which they came.

We solicited participants from nearby manufacturing companies specifically for this short course. Candidates for the course had to have experience in inventory control and had to be in a decision-making position in the company. No company was allowed to sponsor more than two candidates.

The participants attended the course for two hours every weekday morning for two weeks. The mornings were chosen specifically to ensure that the attendees would have the active support of their sponsoring firms.

The subjects were divided randomly into two groups of nine people each. After a substantial part of the course material had been given, the participants were given either a problem book with twenty-four months of simulation output (Figures 1, 2, and 3) or assigned to work a problem on the display unit at individual laboratory sessions. The second problem was assigned later in a similar manner except the groups now used the presentation method that they had not used on the first problem. It was explained that congestion on the display unit precluded everyone working both problems in this way. There was no attempt made to identify individuals by group membership or emphasize this distinction.

The participants were not graded on these problems and were encouraged to work them as part of the education process of the course to investigate some of the fundamental properties of the inventory control policies. Although all of the inventory control policies simulated for the two problems were discussed in class, the specific problem policies were not identified, in order to avoid obvious bias. The participants, then, were asked to make their judgment based solely on the simulation results. The two problems were of equivalent difficulty.

An operational setting was postulated for the problem. Higher-level management had presented the participant with the output data for the set of twelve inventory control policies, and had requested a recommended ranking so that an implementation of policies could be decided upon. The participant was encouraged by management to present his recommended ranking as soon as possible, but cautioned that implementation of the recommended policies could have serious repercussions in the firm, so his best professional judgment was required. Participants were reminded to keep accurate figures on elapsed time spent studying each month's data, and were also reminded to mark the decision month at which they would have presented that ranking as an action recommendation to higher level management.

In such an experiment there are two possible approaches to the problem of evaluating the quality of the decisions made. The most commonly used approach is to furnish the subjects with the criteria which are to be used in evaluating the decisions. For example, in many cases the subjects are first given instruction, then tested to ascertain how well they perform as measured against the pre-established criteria.

This method of measurement may be dependable when the material is objective and the criteria are easily established. In inventory control (and many other problems) the weightings to be used in reaching a decision are highly individual. For example, one firm may emphasize high availability, whereas another may give heavy weight to low investment. When the experimental subjects are experienced in the field and have developed their own criteria for decisions, the method of measuring against instructor-set criteria is unsuitable. One cannot know the extent to which the subjects followed the instructor's criteria and the extent to which they followed their own.

In this experiment we followed an alternative approach to the problem of evaluating the quality of decisions The subjects were experienced in decision-making and were familiar with and interested in the substance of the course. Therefore, they were given no weightings, no criteria of excellence by the instructor. Instead they were explicitly told to apply their own several diverse sets of criteria. The quality of their

decisions was then measured by the consistency of each subject's own ranking at the month of commitment with the later ranking when full information was available.

Although this second approach almost eliminates the meaningfulness of comparisons which show how well the subjects agree with each other, such measures were taken as a matter of interest. The use of self-consistency rather than artificial criteria does, however, improve the credibility of the other statistics. It also substantially improves the generalizability of the results, for it shows the effect on decision-makers when deciding by their own standards, rather than by those dictated by a simplified theoretical model. In subtance, one attempts thereby to examine the decision process as it works under operational, rather than classroom, constraints.

THE DESIGN OF THE EXPERIMENT

Two main analyses were performed on the data: an analysis of variance of the Latin square design, and a computation of rank order statistics.

In the Latin square design of this experiment, it should be noted that instead of experimenting on different subjects in each cell of the square, the same experiment group was involved in both cells of a row. In effect, each group acted as its own control group. Experiments in which the same subjects are used under all q treatments require q observations on each subject, and are called *experiments with repeated measures.* In this experiment $q = 2$ since we are dealing with two treatments and each subject is observed twice.

The 2 x 2 Latin square with repeated measures of Table I was used to analyze the data for:

1. Simulated calendar time to reach a decision (decision month),
2. Actual elapsed time to complete the rankings through the decision month,
3. Actual elapsed time to complete the problem after the decision month,
4. Correlation coefficients for the consistency of the committed decision with the final ranking,

TABLE I—Latin Square Design

	Order 1	Order 2
Group I (9 Individuals)	Display	Printer
Group II (9 Individuals)	Printer	Display

in order to produce an analysis of variance of these statistics. When we discuss the effects of *treatment* in the following pages, we will be discussing whether the display device or the printer demonstrated more influence in our measurements. The effects of *order* will refer to the influence of the participants' doing the problems in the order given to them.

The particular design was chosen in part to isolate that variation due to natural differences between the experimental groups. The separation of the source of variation due to group differences allows a better measure of that variation due to the form of data presentation, which is the statistic of real interest.

Since the values of the decision month are positive integers, they were, as is standard, transformed by taking the square root, to make the variances more homogeneous and at the same time to normalize the distribution. It is this square root value which is used in the analysis of variance computations, and reported in the results.

It is of interest to compare the rank order of the inventory control policies at the decision month and the rank order at the twenty-fourth month for each individual. This comparison gives a measure of each individual's consistency between the ranking decision made at the decision month and that ranking decision made when full information was available. These two rankings for each participant are compared by means of a Spearman rank correlation coefficient (r_s) test. Once computed, these values for r_s are treated as data for analysis in the Latin square design previously discussed. Since the r_s values are known to have a skewed distribution, a standard transformation was made on each of the r_s values in order to place the data on a symmetric scale, so as to normalize the distribution.

In order to find the extent to which the members of each group ranked the policies the same way under the same conditions, Kendall's W coefficient of concordance was computed for each cell of the Latin square. From this, an *average* r_s value of Spearman's coefficient was computed for each of the four cells. This gave a measure of the homogeneity of decisions under the conditions of each cell. These values of *average* r_s indicate a degree of concordance, or how well the group members agreed among themselves as to the rankings. In the interpretation of the result of this computation it should be noted that this experimental group was not attempting to apply a common criterion of excellence in making their decisions. Individual best judgment and experience in the decision process guided the problem solution. The course which the subjects attended neither taught nor encouraged uniformity in decision-making or in performance goals.

The difference between the two values of r_s just

discussed is that the single value of *average* r_s derived by way of Kendall's W statistic is a measure of agreement among all nine participants within a cell; whereas the individual values of r_s derived from Spearman's test are a measure of each person's consistency.

RESULTS

The data analyzed and the detailed results of the analyses are available from the authors. This section of the paper will highlight some of the results.

The usual method in statistical hypotheses testing involves setting the significance level of the test in advance of obtaining the data. The convention used in the analysis of variance deviates somewhat from this formality. The value of F (the significance level of the variance under investigation) is reported to exceed a specified percentile by a comparison with tables of critical values. This allows each reader to establish his own significance level and to judge the results thereby. We will consider a conclusion to be more surely established if the probability of its truth is higher. This methodology does not allow a measure of the power of the test; however, the procedure is valid for estimating the probabilities of the observation in relationship to the assumed sampling distribution. In this investigation we would have been encouraged to find significance at the .05 level, and the significance levels actually attained are noted in this discussion.

The decision month

In this analysis of variance, we were principally interested in investigating the effect of the *treatment* on the subjects. Our first question was: How early in simulated time can a decision be made?

We found that there was less than one percent probability that the observed differences between treatments were due to chance for this question. We can say with a high degree of assurance that the mean decision month arrived at by using the display unit was indeed less than that using the printer output. From another viewpoint, the participants on the average needed to look at less data with the display unit to make a committing decision than they did with the printer output. No effect could be attributed to *order*.

The time to make a decision

The elapsed clock time for a participant to commit himself to a ranking was analyzed next. The mean time to decision using the display unit was, with great certainty ($>.999$), less than the mean time to decision using the printer output. This indicates that the amount of time spent in making a decision was significantly reduced in this experiment by using the display device, confirming the intuitive belief that a person can assimilate a large quantity of data and correlate these data by using display techniques, as opposed to printer output. The convenience of having virtually instantaneous recall of data displays by using the Programmed Function Keyboard is certainly a consideration in the interpretation of the results. Pushing buttons is just inherently faster than paging through a book of data, however well arranged and indexed the book may be. However much or little this consideration affected the results, they show that data can be correlated faster and retained better from a properly programmed display unit.

Unsolicited comment from individual participants supported this conjecture without exception. We observed that the dynamic graphs gave the participant a much better intuitive feel for the situation, and that he was more likely to retain this impression and not have to refer to past data repeatedly. A major help seems to have been the program control which always started the dynamic graphs at month 1 and brought them up to the current month in increments of one month per second. This forced a continual review of the history and derivative of the measure under consideration and undoubtedly reinforced past impressions. It was seldom that a participant asked that the graph be stopped at a month *prior* to his current month so that he could review the static situation as of that past instant. We noted when the experiment was well under way that the participants with more experience as inventory managers used the dynamic graphs extensively, where the less experienced participants relied on the tabular listings presented on the display unit.

The time to finish the problem

The analysis of variance of the total elapsed time to complete the problem after the decision month was examined. One would expect some speed-up by participants after they had made their commitment decision, simply to get to the end to see how well they did. There is some evidence of this speed-up in the time data. How much of this is due to increased familiarity with the problem at hand and how much is due to impatience to get to the final result is difficult to say. The elapsed time after the decision month was tempered by the requirement that the data be ranked at each month. From personal observation, the participants appeared to be conscientious about following the spirit and the letter

of the instructions, but relieved that the big decision had been reached, and were in a hurry to finish the twenty-four months to check their final ranking against their decision month ranking.

The *order* factor was significant at the .01 level, which is reasonable and consistent with our previous comments on the effects of order. In this test the *treatment* factor was completely without significance which is also a reasonable result in view of the observation concerning the impatience to finish the problem.

TABLE II—Values of *Average* r_s Arranged in Latin Square for Final Month

	Order 1	Order 2
Group I	Display .639	Printer .856
Group II	Printer .485	Display .947

Individual consistency in decisions

The transformed Spearman rank correlation coefficients, r_s, were then subjected to an analysis of variance. We found a significance level of .05 for the *treatment*, and we obviously do not have the clear mandate that our other *treatment* factor statistics have given, but the evidence is that the mean correlation coefficient is higher using the display unit than using the printer output. The values of r_s give a correspondence between the participant's ranking of the policies at the decision month and his ranking at the last month of the simulation data, month 24. This, then, is a measure of the consistency between these two rankings. It is also a measure of the participant's discrimination ability—that is his ability to decide whether he has enough information to commit himself to a ranking or not. A decision to commit too soon in relation to each individual's ability and ranking criteria would, in most cases, result in a poor correlation coefficient, whereas being overly cautious and waiting beyond the point where he had sufficient information could not be expected to materially improve the correlation coefficient. Thus, we might say that one interpretation of a low r_s would be that the participant committed himself too early. Other interpretations are, of course, that he simply used poor judgment in his ranking, or that he materially changed his ranking criteria after the decision month. Participants were cautioned to use a consistent ranking schema throughout. As an extreme example, we pointed out to the class that to rank the policies based only on lost sales data through the decision month, then to abandon that schema and to rank the policies only on number of purchase orders generated would not be showing responsible judgment. On the basis of these comments, we should be able to narrow our consideration of a principal cause of low r_s to either too early commitment or poor ranking judgment at some point. Both of these essentially are measures of decision-making ability and we can accept either one or both as reasonable interpretations of a low correlation coefficient. Observe that the greater consistency observed for the display results is in spite of the *earlier* month of commitment. This earlier commitment would be expected to make consistency worse.

For the transformed Spearman correlation coefficient, r_s, the *order* factor was highly significant; the significance was at the .001 level. Thus we accept the hypothesis that maturation would appear to play a large role in the decision-making consistency that is being measured. That is, the ability to meaningfully rank a set of policies grows with practice.

Group consistency in decisions

The results of the decision process at the end of the twenty-fourth month, when the decider had full information available to him, were next examined for *average* r_s. The resulting average r_s is represented in our usual Latin square format in Table II. Here the effect of Group II going from printer output at Order 1 to display unit output at Order 2 is pronounced. There was a moderate increase in the *average* r_s for Group I going from display to printer output, which may be ascribed in part to maturation. However, the *average* r_s almost doubling in Group II when going from printer output to display output may be more than can plausibly be ascribed to maturation alone. In the case of these rankings at the twenty-fourth month, the values of *average* r_s are a valid measure of concordance. Remember that in the rankings at the twenty-fourth month, the deciders all had the same amount of information available to them, which would not have been the case for a decision month ranking. Additionally, the participants at this point were concerned only with the ranking process, and not the additional question of whether or not to indicate a decision month.

CONCLUSIONS AND REMARKS

The strongest result statistically was that actual *time to make a decision* was reduced in favor of the display presentation.

A second statistically significant result was that a decision could be made *earlier*, or on less complete data, by using the display unit. This is the *most* significant result *economically*. And a one percent result is very strong, statistically.

The mean time over both problems to reach the decision month using the display device was 52 minutes. The mean time using printer output was 82 minutes. This result points to the use of a display presentation when economy of time or simply volume of printed output is a serious constraint on the system or the decision-maker. The mean decision month was 9.2 while using the display device and 11.3 for the printer output.

While the two results reported above have the respectability of high statistical significance, the next results to be discussed are at least as important in the evaluation of the experiment. These are the results which answer the question of whether a *better decision* can be made with a display device. We will claim that a decision at the month of commitment that is more consistent with the final decision is a better decision.

The results from Kendall's W statistic and from Spearman's rank correlation statistic show that the subjects when using the display tended to make decisions more *consistent* with themselves, and even with their group.

The economics of a system of display devices for decision-making will not be explored here, however, it is evident that the very specialized research equipment used for this experiment is both expensive and unnecessarily elaborate for an operational system.

The minimum display unit for implementation of an information system of this general type should have alphanumeric display capabilities and a programmed function keyborad, or equivalent means of easy display selection. The size of the display face is crucial to the extent that it must be able to contain enough material to be of interest and still allow character size and spacing to enhance readability. For instance, the IBM 2250 used in this experiment has a display face twelve inches square with a maximum capability of fifty-two lines of seventy-four characters each. The information in the displays for this investigation is rather densely packed and is digestable only by someone sitting in the console chair immediately in front of the display face. A smaller display face would mean that displays would have to be segmented; the same information in smaller characters on a smaller display face would be the wrong compromise. With segmented displays, more programmed function keys would be needed and the problem of how to ask for a particular display becomes more complicated for the user. It is unfortunate that the great majority of available alphanumeric display units have small display faces—eight inches by six inches appears to be a popular size. Other features of the IBM 2250, such as an alphanumeric typewriter keyboard, line-drawing capability, and a light pen, are unnecessary for this application.

In addition to the display device proper, this experiment used other system facilities. The display unit had a self-contained buffer of 8,192 characters. Of this, a maximum of 2,000 characters of buffer storage were used at any one time. The display program in the main storage of the IBM 360 Model 40 used approximately 13,000 characters for program and 21,000 characters for tables. An additional 46,000 characters of disk storage were used for table overlays.

The display system evidently achieved the objective of presenting a complex situation, which involved many inter-relationships, in a manner such that the key concepts and fundamental correlations were clearly understandable. The display system appeared to facilitate interpretation and extrapolation of the relevant data. The reduction of reaction time of top-level decision-makers in this environment is both an interesting result and an important objective of any executive display system.

One point of great interest for further work would be the exploration of the differential cost or savings of decisions using display units and printer output. This is a rather difficult area to define, since dollar values and weightings must be assigned not only to the reduction in inventory valuation and the cost of lost sales, but also to the generated purchase actions, availability of material, timeliness of decision, system cost and other factors.

Statistics on the frequency of use of the various displays should be collected, both automatically and by experimenter observation. The correlation of the frequency of use by display type with the individual's consistency of decision would be most important for the design of extensions of this system.

Whatever the extension of the experiment, there should be the capability for the decider to request a hard copy of any display he wishes. If line drawing capability is used, this, of course, implies the availability of the equivalent of a line plotter for hard copy output. This requirement is more operational than experimental. We have no doubt as to the utility of such a feature for the decider in an operational environment, and if a display unit has a line drawing capability, it should be used with this requirement in mind.

SELECTED BIBLIOGRAPHY

Computer characteristics quarterly (second quarter)
Keydata and Adams Associates Inc Watertown Massachusetts
1968

R G BROWN
Statistical forecasting for inventory control
McGraw-Hill Book Company New York 1959
D S BURDICK T H NAYLOR
Design of computer simulation experiments for industrial systems
Communications of the ACM 9 May 1966 pp 329-339
W G COCHRAN G M COX
Experimental designs
John Wiley & Sons, Inc New York 1962
W L HAYS
Statistics for psychologists
Holt, Rinehart and Winston New York 1963
IBM system/360 operating system graphic programming services for IBM 2250 display unit
International Business Machines Corporation Form C27-6909-4 December 1967
IBM system/360 component description, IBM 2250 display unit model 1
Form A27-2701-1 January 27 1967
H R LUXENBERG R L KUEHN eds
Display systems engineering
McGraw-Hill Book Company New York 1968
T H NAYLOR J L BALINTFY D S BURDICK C KONG
Computer simulation techniques
John Wiley & Sons New York 1966
K WERTH T H WONNACOTT
Methods for analyzing data from computer simulation experiments
Communications of the ACM 10 pp 703-710 November 1967
G W PLOSSL O W WIGHT
Production and inventory control
Prentice-Hall Inc Englewood Cliffs NJ 1967
H SACKMAN W J ERIKSON E E GRANT
Exploratory experimental studies comparing online and offline programming performance
Communications of the ACM 11 pp 3-11 January 1968
R G D STEEL J H TORRIE
Principles and procedures of statistics
McGraw-Hill Book Company New York 1960
B J WINER
Statistical principles in experimental design
McGraw-Hill Book Company New York 1962

Concurrent statistical evaluation during patient monitoring*

by S. T. SACKS, N. A. PALLEY and H. SHUBIN

University of Southern California School of Medicine
Los Angeles, California

and

A. A. AFIFI

University of California
Los Angeles, California

BACKGROUND

The Shock Research Unit, a specialized clinical research facility, has been developed by the University of Southern California's School of Medicine for the purpose of rendering intensive care to seriously ill patients. Included is a medium-sized digital computer with a real-time system which monitors the critically ill patient. In addition, the system is being used to study the underlying mechanisms of the disease process and for developing new techniques of evaluating and treating seriously ill patients.

The Shock Research Unit was started in 1962 and since that time approximately 1,000 patients have been treated at the unit. In mid-1963 a computer was obtained and a system developed for monitoring patients.[1] In 1968 the Shock Research Unit obtained a third generation computer and applications programs for a much-enlarged system of real-time monitoring were written.[2]

"The patient in circulatory shock is an example of the need for immediate response to clinical observations. Low blood pressure and reduced blood flow are characteristic of the patient in shock. He may become stuporous or comatose due to the inadequate circulation to the brain. His kidney may cease to function and his respiration may fail.

Assessment of the circulatory and respiratory status of such a critically ill patient requires measuring a number of variables: arterial and venous pressure; blood flow and volume; electrocardiogram; blood gases such as oxygen and carbon dioxide and blood constituents such as potassium. Repeated assessment of these variables is required since the critically ill patient is not in a steady state, but may undergo rapid and often unpredictable changes in status".[3] A number of other patient monitoring facilities employing digital computers are currently in operation across the country. Two such units engaged in the simultaneous monitoring of multiple variables are at the Pacific Medical Center[4] in San Francisco and the Latter Day Saints Hospital[5] in Salt Lake City.

At the Shock Research Unit, a combination of sensors and transducers is used for measurement of vital signs and additional parameters of clinical importance, and the data is processed to derive numerical information helpful to the physician. Information which is supplied directly by sensors applied to the patient's body include eleven primary measurements and twenty-five parameters which are recorded and displayed with a frequency ranging from once a minute to once every twenty-four hours. All numeric and textual data are stored in an on-line patient file organized by type of data and by time. The data may be retrieved by the user through a bedside K/D and by current scheduled applications programs. Given this large amount of sequential data, the physician must have some means of combining it into a meaningful evaluation of changes in the patient's status.[6] All present commercially available patient monitoring

* The research programs of the Shock Research Unit are supported by grants from the John A. Hartford Foundation, Inc., New York, and by the United States Public Health Service research grants HE05570 and GM16462 from the National Heart Institute and grant HS00238 from the National Center for Health Services Research and Development.

systems depend on alarm limits which are manually set and may not be adaptable to the particular clinical situation. These alarms are based on absolute values of single variables and do not take into account relationships among variables. When these univariate alarm systems sound too frequently, the usual response of the person in charge is to set wider alarm limits. Such actions have no statistical basis and may be detrimental to patient care.

The availability of an on-line computer monitoring system allows the examination of many variables simultaneously as well as the observation of their interrelationships. The purpose of this paper is to describe the development and application of an automated screening procedure for evaluating physiological data obtained from continuous monitoring of critically ill patients. Three criteria, motivated by clinical experience, are used in the evaluation of the patient's data. The first is the absolute value of the monitored variables. These values contain the information necessary to determine the current status of the patient and to assure that the monitoring equipment is operating properly. The monitoring interval then may be modified appropriately. However, in a dynamic system the information of interest to the clinical staff is contained in the sequential changes in the measured variables. These may reflect sudden, unexpected changes in status or the expected responses to treatment and constitute the second criterion. Variations are sometimes more meaningfully interpreted relative to the absolute values of the variables. The third criterion, therefore, is the proportionate change in the monitored variables.

Many of the variables which are routinely monitored in a critically ill patient are highly correlated and the clinical significance of a given measurement becomes more apparent when examined in the context of the remaining measurements. For example, Figure 1 illustrates a scatter diagram of a typical random sample of simultaneously measured systolic (SP) and diastolic (DP) pressures. The data tend to accumulate in an elliptical region with the highest density near the center. While the vertical and horizontal dashed lines determine the 95 percent confidence intervals for SP and DP individually, the elliptical region includes the same region for the pair of measurements. Point A illustrates a measurement which lies within the individual normal limits but exceeds the bivariate limits, because low systolic pressure is combined with high diastolic pressure. Thus normalcy is determined by knowing both distance and direction from the mean. The region inscribed by the ellipse may be expressed by the inequality

$$D^2 \leq C$$

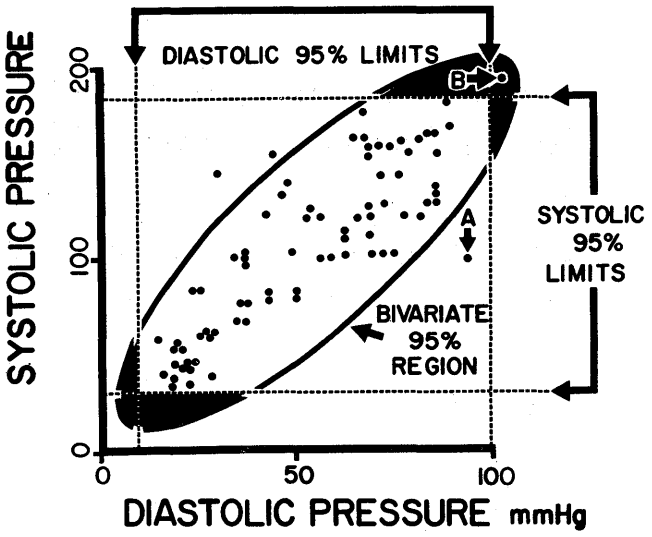


Figure 1—Scatter diagram of a set of values of systolic and diastolic pressures indicating univariate and bivariate 95 percent confidence regions

where C is a constant and D^2 is the Mahalanobis distance for the two variables SP and DP.[7] This Mahalanobis distance may be computed for any number of variables. If Y is a vector of r observations, $\bar{Y}$ is the mean vector, and S is the covariance matrix, then

$$D^2 = (Y - \bar{Y})' S^{-1} (Y - \bar{Y}).$$

Because D^2 incorporates interdependencies among the variables, unusual combinations of variables or changes result in abnormally large D^2 values, even though the variables or changes when regarded individually may fall within normal limits.

Using data sampled from patients previously monitored in the Shock Research Unit, these ideas provided the basis for a system to respond in real-time to changes in patient status. The responses of the system include:

1. Informing medical personnel of unusual changes in the set of monitored variables.
2. Selecting data for display.
3. Selecting data for recording on a permanent file.
4. Running of special analysis programs.

These responses will be discussed in detail below.

Development of method

Forty-one patients were chosen randomly from the 750 patients monitored in the SRU up to August, 1967.

Only those patients who were observed for at least four hours were included.

Seven variables were selected from the many which are routinely monitored: systolic pressure (SP), diastolic pressure (DP), mean arterial pressure (MAP), mean venous pressure (MVP), rectal temperature (RT), heart rate (HR), and respiratory rate (RR). Each patient's record was examined and that four-hour period which had the least missing data was chosen. Measurements on sets of these variables were recorded on each patient at five-minute intervals over the selected four-hour period. Estimates for any missing data, which accounted for less than 5 percent of the total data, were made by interpolating between the nearest recorded values before and after the missing observation. From these absolute value sets the following were calculated: 5-, 10-, 15-, 30-, and 60-minute changes, proportionate 5-, 10-, 15-, 30-, and 60-minute changes.

Figure 2 illustrates the successive changes in a single variable over time. It should be noted that this choice of intervals was made in order to account for both long-term and short-term variations. An example of the calculation of 5-minute changes and proportional changes in a set of variables is illustrated in Table I. The value of the Mahalanobis D^2 was calculated using the appropriate sample mean vector and covariance matrix for each vector of absolute values, x-minute changes and proportionate x-minute changes ($x=5$, 10, 15, 30, 60). In the remainder of this paper frequent reference will be made to these eleven types of measurements, namely, absolute values, the five x-minute changes, and the five x-minute proportionate changes.

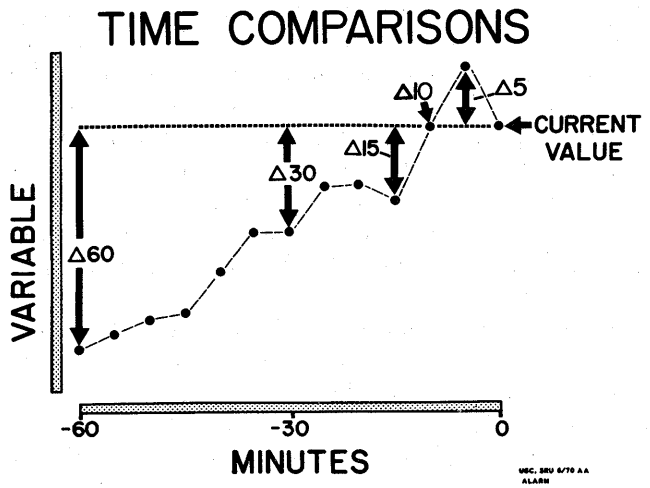


Figure 2—X-minute changes in a given variable for $x=5$, 10, 15, 30 and 60

TABLE I—An Example of 5 Minute Changes and Proportional Changes in the 7 Variables Set

Variable Name	Value at Time t	Value at Time $t-5$	5-minute change at Time t	5-minute proportional change at Time t
SP	122	122	0	0. = 0/122
DP	80	87	−7	−0.088 = −7/80
MAP	94	98	−4	−0.043 = −4/94
MVP	1	1	0	0. = 0/1
RT	37.7	37.8	−0.1	0.003 = 0.1/37.7
HR	98	100	−2	−0.020 = −2/98
RR	21	22	−1	−0.048 = −1/21
D^2	12.9	8.5	11.8	43.1
Region	B	A	B	C

This procedure yielded eleven distributions of D^2. Percentiles necessary for setting monitoring limits were then calculated for each of these D^2 distributions.

In monitoring a critically ill patient, it is inevitable that, on occasion, values for one or more variables are missing. This may occur if a measurement is taken while a catheter is being flushed or if the EKG leads are accidentally disconnected or simply because monitoring of all variables is not started simultaneously. A preliminary screening procedure will exclude such invalid measurements. Consequently, tables of D^2 for incomplete vectors of observations are necessary. It is possible to empirically derive and store the 1309 tables of D^2 percentiles necessary to handle all combinations of missing variables. However, in order to minimize storage requirements and allow the system to operate in real-time, an alternative procedure was developed as follows.

If the population mean vector and covariance matrix were used in computing D^2, and if the observation vector were normally distributed, the distribution of D^2 would be that of a Chi-square variable. Since the variables under consideration are not exactly normally distributed and sample estimates were used instead of population parameters, the empirical distribution of D^2 deviates from that of Chi-square. However, they do remain similar in shape. Thus it was hypothesized that the ratio of a percentile of the Chi-square distribution to the corresponding empirical percentile of D^2 for a given type of measurement is independent of the number of components in the vector of measurements. That is, for absolute values, x-minute changes and proportionate x-minute changes, we assume that:

$$\frac{p\text{th percentile of } D^2 \text{ distribution based on } r \text{ components}}{p\text{th percentile of Chi-square distribution with } r \text{ d. f.}}$$

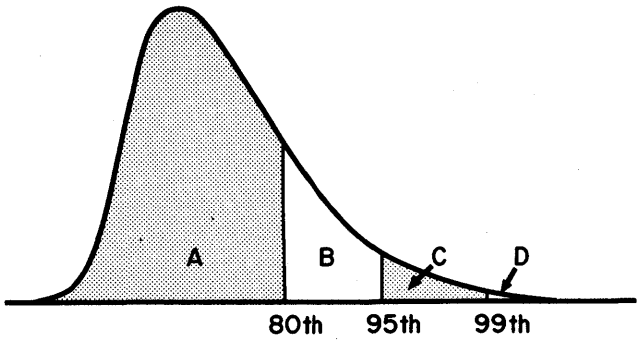


Figure 3—Monitoring regions with alarm limits based on the empirical distribution of D^2

$$= \frac{p\text{th percentile of } D^2 \text{ distribution based on 7 components}}{p\text{th percentile of Chi-square distribution with 7 d. f.}} \quad \text{for } r = 2, \ldots, 6.$$

From this assumption, approximate empirical percentiles for the D^2 distribution based on any number of components were generated from the D^2 distribution percentiles based on seven components.

Each D^2 distribution is divided into regions designating the degree of normalcy of a set of measurements as shown in Figure 3. These limits are based on the empirical distributions of each set of values in the base sample. In the figure the limits are taken as the 80th, 95th, and 99th percentiles. Region A may be classified as the *normal* region, containing values which are expected to occur 80 percent of the time. Region B, the *moderately abnormal* region, contains the values expected to occur between the next 20 percent to 5 percent of the time. Region C, the *abnormal* region, includes the 95th to 99th percentiles, and Region D may be interpreted as the *highly abnormal* region since it contains the extreme values which would be expected to occur only one percent of the time. Sets of three limits are computed for 0, 1, . . ., 6 missing variables. The total number of tables to be stored is thus reduced from 1309 to 77, and with 3 limits per table, only 231 values are stored (11 change vectors × 3 region limits × 7 possible numbers of missing values). In addition eleven 7 × 7 covariance matrices based on the sample data are stored.

Note that the original choice of the three percentile limits is arbitrary and may be modified as experience is gained with the system.

System algorithm

The basic algorithm for this system comprises the following steps:

1. Each five minutes, the most recent set of observations is retrieved from the patient file.
2. The patient file is searched sequentially for the measurements taken 5, 10, 15, 30 and 60 minutes previously and also for alarm information recorded at those times. For those variables having a valid present and past value, changes and proportionate changes are computed. The appropriate stored covariance matrix, or a reduced matrix corresponding to the variables present in each change vector, is then used to calculate D^2. In some cases all components necessary for computing D^2 may be missing.
3. For each of the computed D^2 values, the appropriate set of three region limits is retrieved from the tabled values on the disc file. The index of the set of limits to be retrieved is determined by the change vector being considered (1, . . ., 11) and its dimension (1, . . ., 7).
4. Using the region limits, a cateogry is assigned to each of the available D^2 values.
5. The identity of the most extreme D^2 and the category to which it is assigned are stored in the patient file.
6. Appropriate system action is taken as described below.

Uses of the system

The action taken by the patient monitoring system in response to the computed D^2 depends upon the most extreme category detected. Any D^2 value falling in the B, C or D regions will result in a signal to the ward staff. The signal is coded to indicate the degree of abnormality. A red and a yellow signal light and a chime, under computer control, are mounted near the status display. The chime signals a D^2 value detected in any of the three regions. In addition, the red light accompanies region D while the yellow light accompanies region C.

Some measurements such as heart rate and respiration rate, are usually read from the output of preprocessors which derive the information from the raw data. When an alarm occurs, the digital analysis program reads the original wave-form signal directly to verify the abnormal value encountered. The EKG analysis program, which may be run on demand, is also triggered in response to an alarm.

In many cases, the cause of the alarm is immediately obvious by noting the values on the status display, as shown in Figure 4. If the physician wishes to know in detail the values causing an alarm, he may call a program on the keyboard display which reports the current and previous values of the variables used in computing D^2. As illustrated in Figure 5, this program indicates the time over which the extreme changes have taken place and the magnitude of the changes. The physician may then indicate the probable cause of the alarm. The cause may be a change in the normal course of the patient's progress or a specific treatment such as medication, fluid infusion, or adjustments made to the respirator. Changes from such causes, although resulting in an alarm, do not necessarily reflect emergent situations. In addition, the alarm may have been caused by an artifact in one of the monitored variables. This may occur if a catheter becomes clogged or is inadvertently flushed while the signal is being read or if EKG leads become dislodged. This information is stored in the patient file and at the same time the signal light is turned off.

Summaries of alarm information are stored in the patient file. These summaries contain the following information: the time of the alarm, its category and the specific change or proportionate change causing the alarm. In addition, if the ward staff responded to the alarm through the keyboard display, the summary will contain the time of the response and may thus contain a flag indicating that the cause of the alarm was an artifact and the particular variable or

19/0517 STATUS	BED 1	BED 2
SYS/DIA	64/37	96/58
MAP	45	72
VEN	6	8
HR/PDEF	84/ 3	98/ 0
RESP	17.	22
RECT/AMB	34.6/25.4	37.1/25.4
TOE L/R	24.8/24.3	24.9/25.3
URS/UR60	6/39	0/28
DAY/TIME	19/0201	19/0317
CI	1.9	1.8
AT/MCT	14/28	11/25
RESIST	1347	2227
HCT/TIME	29/0210	19/0329
PH/PCO2	7.31/51	7.49/24
PO2/SAT	71/96	278/100
PV/TIME	58/0224	29/0346
RCM/TIME	22/0224	19/0346
LAC/TIME	3.4/0217	2.9/0338

Figure 4—Bedside status display showing values of the monitored variables for two patients

```
ALARM SUMMARY BED 1

FICIAL, ARTHUR I.
SRU# 1234

TIME OF ALARM: 04/0105        REGION: C (LESS THAN 5 PERCENT)
SOURCE OF ALARM: 5 MINUTE CHANGE

     TIME      0100      0105      DELTA 5

1    SP         122       122         0
2    DP          80        87         7
3    MAP         94        98         4
4    MVP          1         1         0
5    RT        37.7      37.8       0.1
6    HR          98       100         2
7    RR          22        21        -1

CAUSE OF ALARM: 0) UNKNOWN, 1) TREATMENT, 2) ARTIFACT  ? 2

VARIABLE ? 2 (DP)

0) UNKNOWN ORIGIN
1) CATHETER DAMPED OR DISCONNECTED
2) FLUSHING
3) AMPLIFIERS
4) OTHER ELECTRONICS

CODE ? 2                    SUMMARY STORED
```

Figure 5—Keyboard/CRT display alarm summary showing that the set of 5-minute changes falls in monitoring region C. The attending physician indicated that the value for variable 2 was an artifact caused by flushing

variables involved. If the information was supplied by the physician, the clinical event related to the alarm condition is also coded.

The alarm summaries are utilized by the patient monitoring system in a variety of ways. As indicated in Step 2 of the system algorithm, the patient file is searched so that changes may be computed. Summaries of alarms stored at the same time as any of the five previous sets of values are also retrieved. If the summary indicates that any value in the set used for calculating the D^2 was the result of an artifact, that value is deleted from computation. This prevents a single artifact from causing several alarms as that set of values is successively retrieved 5, 10, 15, 30 and 60 minutes later.

During the patient's stay, the physician often reviews sets of data on the bedside keyboard display. Since on any display frame only six sets of simultaneous measurements may be viewed, as shown in Figure 6, it is desirable to eliminate redundant observations.

This may be achieved by displaying only those sets of data falling in abnormal regions. In the same way the permanent hardcopy record, which is printed upon patient discharge, can be specified to contain only significant information. Under conditions of intensive monitoring where measurements are made every minute, this may represent a valuable reduction in the bulk of the hardcopy record to be reviewed. Those sets of variables containing values previously identified as artifacts may be deleted from both the keyboard

HEMODYNAMIC DATA SUMMARY BED 3

PATIENT NAME- FICIAL, ARTHUR
HOSPITAL NUMBER- 911676
SRU NUMBER- 1040

DAY/TIME HOURS	DD/HHMM ON WARD	10/2343	10/2348	10/2353	10/2358	11/0003	11/0007
SYS	MMHG	95.0	95.0	95.0	94.0	94.0	91.0
DIA	MMHG	54.0	54.0	54.0	54.0	54.0	52.0
MAP	MMHG	70.0	70.0	70.0	69.0	69.0	67.0
VEN	MMHG	8.0	8.0	8.0	8.0	7.0	8.0
HR	/MIN	120.0	120.0	117.6	120.0	120.0	120.0
PR	/MIN	118.0	118.0	118.0	118.0	118.0	118.0
RESP	/MIN	13.0	15.0	17.0	18.0	17.0	15.0

1) FIRST, 9) LATEST 4) PREVIOUS, 6) FOLLOWING, OR DAY/TIME
ENTER CODE

Figure 6—Hemodynamic data summary showing six sets of values of seven variables

display presentation and the hardcopy output. This procedure is also used with the storage oscilloscope at the bedside which displays a trend plot of monitored variables. By referring to the alarm summaries the program which produced this display deletes invalid data points from the plot.

In order to have the patient file accessible in case of system failure, teletypes, remote to the ward, produce hardcopy output containing the latest measurements. While the status display is automatically updated every 30 seconds, the teletype, because of its low speed, cannot be updated at the same rate. In order to make efficient use of the teletypes and fulfill their function of preserving data of interest, the teletype records are updated every half-hour or whenever abnormal values are encountered.

SUMMARY

The alarm system described in this paper both depends on and augments the capabilities of the digitally controlled patient monitoring system. It utilizes multivariate techniques to compute statistics which are sensitive to relationships among the monitored variables. The degree of abnormality of a computed statistic is evaluated in terms of empirical distributions. These distributions were derived from a population of critically ill patients similar to that being presently monitored by the system. The actions taken by the patient monitoring system in response to the alarm depend on the severity indicated by the alarm. The occurrence of the alarm and the cause, if known, become part of the patient's file and are accessible to applications programs involved in data display. The stored summary of the alarm information assists the clinical staff in case review, and provides a basis for evaluating and modifying the alarm system itself. Such modifications might include the redefinition of the percentiles which define the alarm categories or even the number of such categories. Extensions of this alarm system might enable a small centrally located digital processor to evaluate sets of data from a number of analog monitoring modules at various bedsides.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Doctor Max Harry Weil, Director of the Shock Research Unit, whose able guidance and support made this work possible. We would also like to thank Doctor William Rand who, during his employment at the Shock Research Unit, contributed greatly to the collection of the data in the preliminary discussions leading to the formulation of the system. Our thanks also go to Miss Cecilia Pasos for typing the manuscript.

REFERENCES

1 M A ROCKWELL H SHUBIN M H WEIL
Shock III: A computer system as an aid in the management of critically ill patients
Communications of the ACM Vol 9 No 5 May 1966

2 D H STEWART D E ERBECK H SHUBIN
A computer system for real-time monitoring and management of the critically ill
AFIPS-Conference Proceedings Vol 33 December 1968

3 IBID

4 J J OSBORN J O BEAUMONT J C RAISON J RUSSELL F GERBODE
Measurement and monitoring of acutely ill patients by digital computer
Surgery Vol 64 pp 1057-1070 December 1968

5 T A PRYOR H R WARNER
Time-sharing in biomedical research
Datamation Vol 12 pp 54-63 April 1966

6 A A AFIFI W RAND H SHUBIN M H WEIL
A method for evaluating changes in sets of computer monitored physiological variables
Submitted for publication

7 T W ANDERSON
Introduction to multivariate statistical analysis
John Wiley and Sons New York 1958

Associative capabilities for mass storage through array organization*

by ARNOLD M. PESKIN

Brookhaven National Laboratory
Upton, Long Island, New York

THE ASSOCIATIVE MEMORY PROBLEM

Since computers first came into wide usage, digital systems designers have been intrigued by the possibilities of associative or content addressable memories. The concept is quite easy to understand; whereas, in the conventional case, the address is furnished to the memory and the data stored at that location is the expected result, in the associative reference, the data is furnished and the expected result is a list of all addresses which contain matching data. Up to now, however, the physical systems which exhibit the requisite symmetry to realize this concept have been necessarily very costly because the commonly used, low cost, random access memories do not easily lend themselves to this new operation. Those digital systems designers who predicted widespread use of associative memories by the late 1960's are found in retrospect to have seriously underestimated the difficulty in implementing thin magnetic film or superconducting memory systems, on which these forecasts heavily depended.[1,2,3]

Logic designers and programmers have each been called upon to simulate the associative memory for specific applications, but implementation in logic has incurred an almost prohibitive cost-per-bit and software search subroutines expend equally unattractive lengths of time.[4] The advent of large scale integration and the rapidly decreasing cycle times of new computers promise to make either recourse somewhat more attractive but at this time associative capability is still unthinkable for anything but a small block of information.

A CONTENT ADDRESSABLE N-CUBE

One approach to this problem is to expand the associative capability from a small block of data to a larger storage array. If the small block possesses the associative property but the large array does not a method of data base organization and array interconnection is possible so that the larger array begins to exhibit all the properties of the smaller one including reference speed and content addressability. A geometric interpretation of this approach is depicted in Figure 1.

A cubic storage array is shown, only one plane of which has associative capability; this is the $z=0$ plane. A match resulting from an association in this plane will define another plane, perpendicular to the first. Assuming that the memory reference requires that a second association be made on the data in this newly selected plane also, its contents may be loaded into the plane with the content addressable capabilities, while the original contents of that plane are temporarily stored in the buffer of Figure 1. The association may then be completed and the array restored. For purposes of this discussion, it is assumed that the loading and unloading capability is implemented by adding to each bit the necessary hardware to make it a unidirectional fast shift register.

It is then possible to select or "find" any given word in the cube in the time required to perform two associative references and the requisite loading and restoring of the $z=0$ plane. What this scheme accomplishes can be demonstrated by examining its cost-performance characteristics. Symbols will be defined in the following manner:

w = word length: number of bits per dimension

C_a = associative portion of cost per bit

C_c = conventional portion of cost per bit

t_a = time duration of a two-dimensional associative reference

t_s = time required to shift a bit "w" places.

For use in approximation, the following relationships

* Work performed under the auspices of the U.S. Atomic Energy Commision.

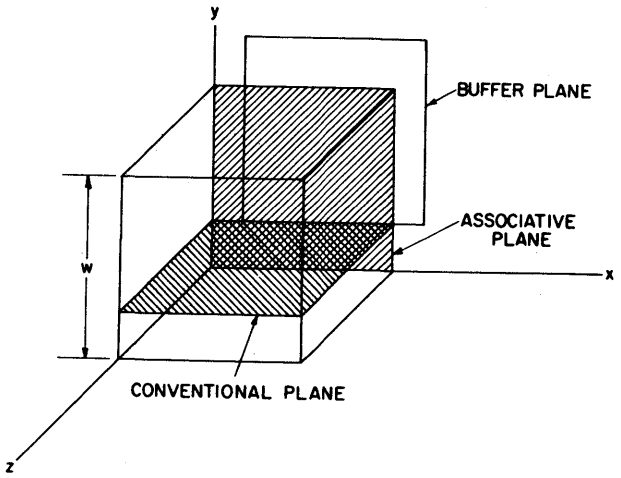


Figure 1—A content addressable 3-cube

are assumed to exist

$$C_a = 10C_c$$

$$t_a = (20/w)t_s.$$

These relationships state that the content-addressable and shift portion of the cost is ten times the cost of a conventional memory of the same size, and that the time required to do a memory reference in an IC memory system is twenty times that of a one bit shift in logic of the same family. These constants vary with the technology and mode of implementation used, of course, but experience indicates that these are reasonable assumptions.[4] What is most important, nevertheless, is that these relationships are expressable by constants, and not what the specific values of these constants are.

In two dimensions, then,

$$\text{total cost} = (C_a + C_c)w^2$$

$$= 11w^2C_c.$$

With the three-dimensional scheme of Figure 1, there are now w^3 bits, each of which may be thought of to have associative capability, but only one out of every w bits incurs the associative portion of the cost. This is accomplished at the expense of a reduction in the total speed of reference. Specifically,

$$\text{total cost} = w^3C_c + w^2C_a$$

$$= (w^3 + 10w^2)C_c$$

$$\text{duration} = 2t_a + 2t_s$$

$$= t_a[2 + (w/10)].$$

There is no reason to stop at three dimensions, because as each new dimension is added:

1. the associative capability increases exponentially;
2. the conventional portion of cost only follows exponentially;
3. the associative portion of the cost does not increase;
4. the duration of a reference increases linearly.

Summarizing for an n-cube:

Number of associative elements $= w^n$

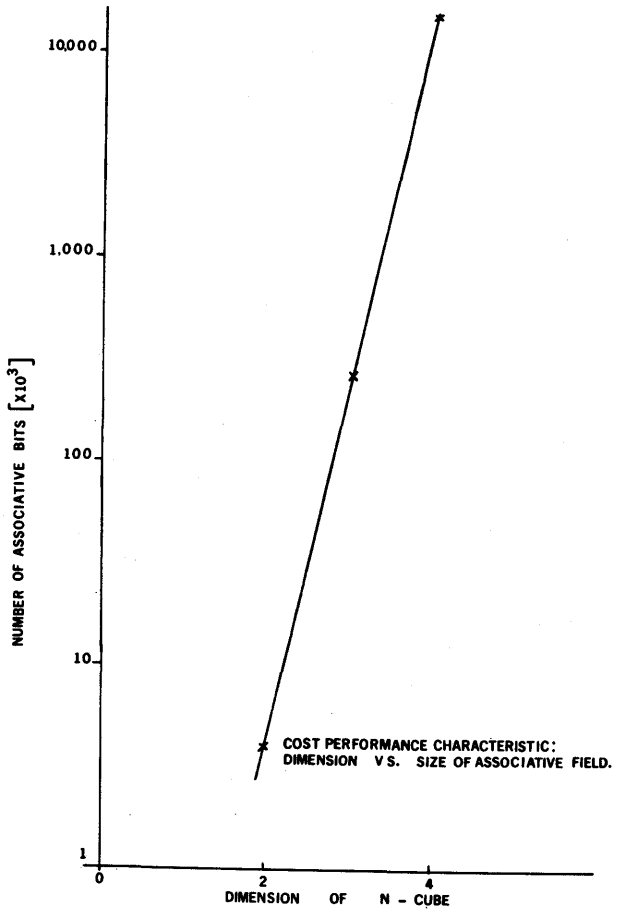
Total cost $= (w^n + 10w^2)C_c$

Duration $= (n-1)t_a + 2(n-2)t_s$

$$= t_a[(n-1) + w(n-2)/10].$$

These relationships are summarized in Figure 2.

Whether or not the n-cube is a practical design approach is a difficult question to answer in view of the current flux in integrated circuit technology. IC's are becoming available which provide the associative, but not the shift capability. Providing a bussing structure

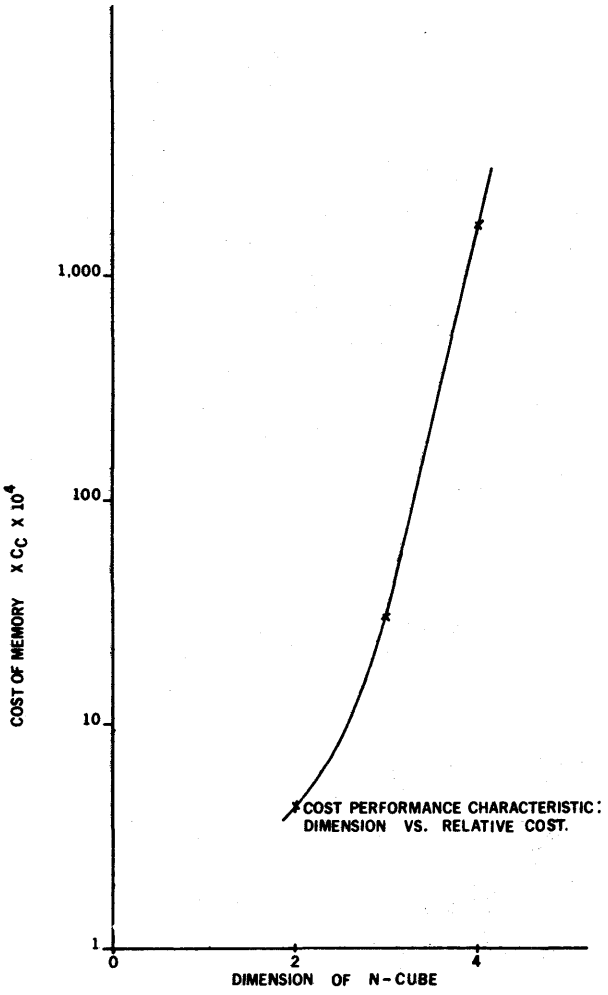


so that the information of each plane can find its way to the $z=0$ or associative plane may add considerable overhead to the system. Also, the question of how to handle multiple matches in an early plane reference can only be addressed to the specific application for which the n-cube is being used. However, one can attempt to take advantage of the exponential capability increase vs. linear time duration increase with a two-dimensional associative array interfaced to a properly partitioned conventional random access storage bank.

A PRACTICAL FOUR-DIMENSIONAL APPROACH

Figure 3 shows a computer configuration featuring a large random access bank of core storage which can be referenced by one or more central processors through the mass storage controller. This configuration is typical of many large computer installations. Also connected to the mass storage unit is a small two-dimensional content addressable memory, which interfaces to the mass storage controller through its own transfer coupler, which in turn is controlled programmatically through the I/O interface of the computer.

The two-dimensional content addressable memory (2DCAM) consists of sixty-four 64-bit words which may be accessed either conventionally by address or



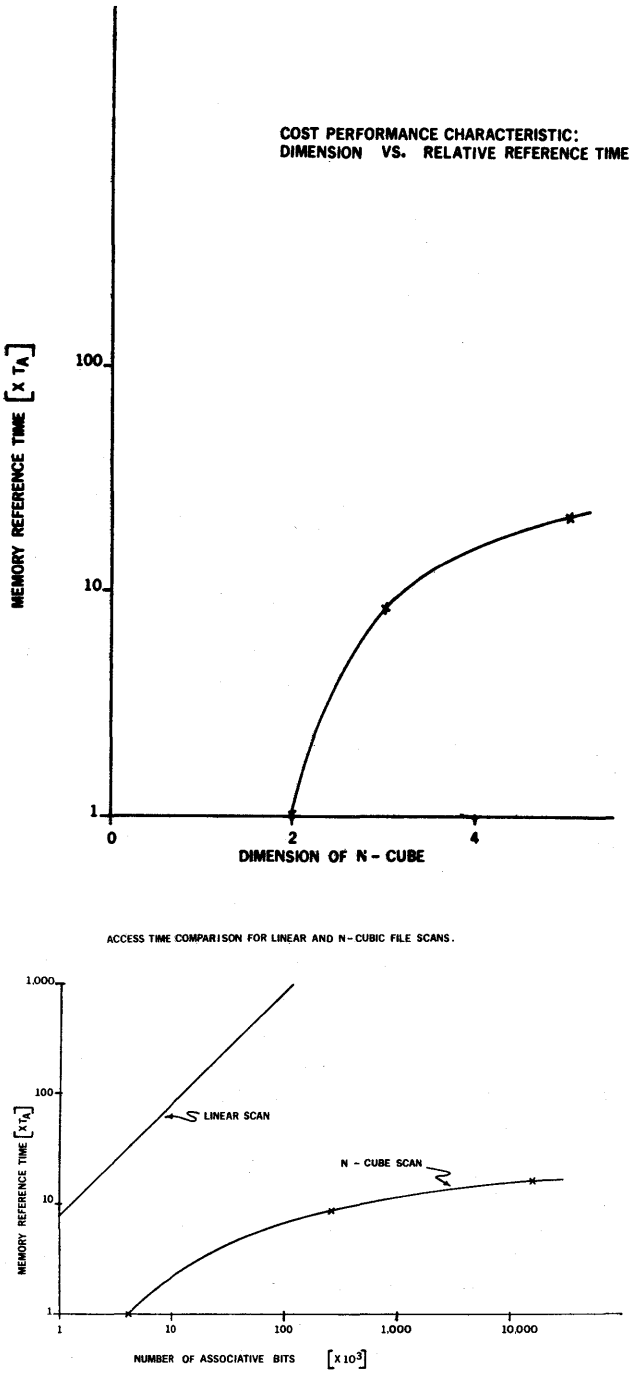


Figure 2—Cost-performance characteristics of a content-addressable N-cube for $w=64$ bits per word

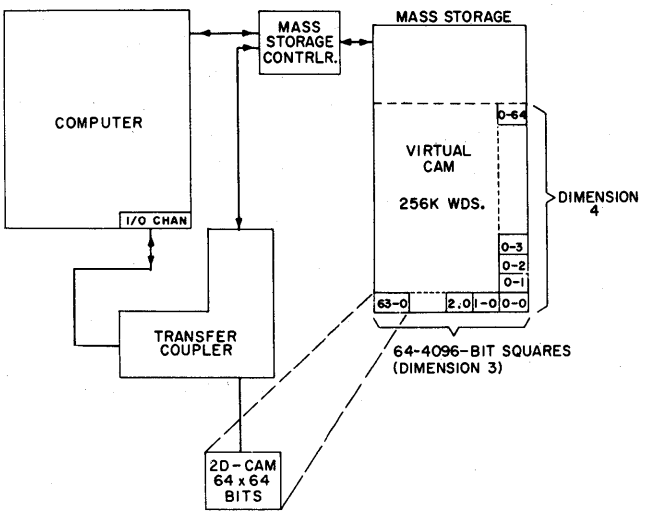


Figure 3—Proposed 4*D* configuration

by content; that is, exhibiting the associative property. The field of association can be specified under program control by loading a mask register, thus increasing the 2DCAM's versatility. This device may be built around integrated circuit content addressable memory modules, such as are becoming available in various logic families from several different manufacturers. It is also important to implement a burst load/unload capability for the entire 2DCAM, analogous to the perpendicular shift capability in the n-cube configuration. In the TTL or ECL logic family, storage cycle times of less than 100 nanoseconds are now commonplace.

The portion of the mass storage which is dedicated to the 2DCAM will for many applications, virtually exhibit the associative property itself. If the mass storage is partitioned as suggested in Figure 3, there will be four dimensions of associativity, each 64 bits long, or over 256 kilo-64-bit words capable of exhibiting the associative property in little more than the time required to load the 2DCAM twice. There will be one 64 word block which can be thought of as the interfile index. It is the block in which the first association takes place and is thus analogous to the $z=0$ plane of the n-cube. A match resulting from an association in this table will determine which one of sixty-four sectors of conventional storage contains the information that is ultimately sought. An index for that sector is then loaded into the 2DCAM and a subsequent association is performed, this time to determine which 64 word block will be required for the final associative reference. Thus a partially ordered data field of 16 million bits has been scanned in the time required to load the 2DCAM twice and perform a content addressable reference three times, which together would total an elapsed time of less than twenty microseconds.

As is shown in the graph of Figure 2d, this method compares quite favorably to the time required to scan 256,000 64-bit words programmatically, especially when it is realized that while the association takes place, the central processor is free to perform other tasks. Compared to the most advanced techniques of binary searching, the n-cube associative search over this data base can be performed approximately five times faster.

THE BROOKHAVEN CONFIGURATION

In the Central Scientific Computing Facility of Brookhaven National Laboratory, the computer of Figure 3 is one of Brookhaven's Control Data 6600 processors. The mass storage system is a one-million word Extended Core Storage (ECS), also a Control Data product. The ECS controller, designated the 6640, can service up to four 6000 series computers with a maximum throughput rate of 600 million bits per second.[5]

The transfer coupler of Figure 3 is represented in the Brookhaven configuration primarily by a branch of the Laboratory's remote computer network, known as Brooknet, as shown in Figure 4. Brooknet can provide on-line service to eight remote areas, any of which can be up to 5,500 feet away from the central facility. Each remote area, in turn, may have eight or less remote computers on-line within a radius of 1000 feet from a remote Brooknet multiplexer. As shown in Figure 4, Brooknet provides selectable data paths for the remote from either the computer I/O interface or the ECS controller, to which Brooknet logically appears as a third 6600. The I/O interface is utilized for status and fixed format control messages and the ECS path is used for block data transfer of files. Because it can provide a high speed interface between a special pur-

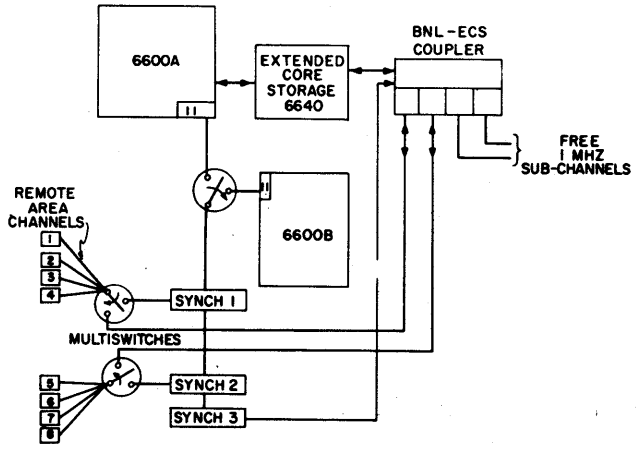


Figure 4—Brooknet

pose device (such as the 2DCAM) and ECS, a Brooknet link meets all the requirements for a transfer coupler. That is, ECS would suffice as the large conventional storage, the 2DCAM would reside at the remote system, and the requisite swapping of the 64 word blocks of information are performed as normal Brooknet data transfers just as if they were input or output files from a remote batch station. For this system, in order to maintain compatibility with the Control Data equipment, the word size must be sixty, rather than sixty-four bits long.[6]

The two-dimensional content addressable memory (2DCAM) designed for Brooknet has the block diagram of Figure 5. This device consists of sixty-four 60-bit words, expandable to 256 60-bit words, which can perform associative or conventional memory references. There are six functions defined for it:

1. Load mask register;
2. Write (conventional);
3. Read (conventional);
4. Associate and count matches;
5. Associate and present addresses;
6. Multiload.

The multiload function implements the burst loading feature required to perform successive associations efficiently. The entire memory can be loaded in 6.3 microseconds.

To render the 2DCAM hardware and software compatible with Brooknet, a modest control computer is required to engage in the Brooknet dialogues, and two more pieces of interface equipment are needed for level conversion and logic translation. A 2DCAM system

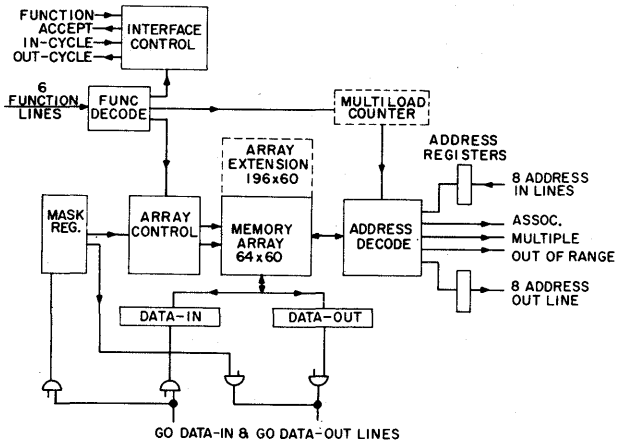


Figure 5—Content addressable memory block diagram

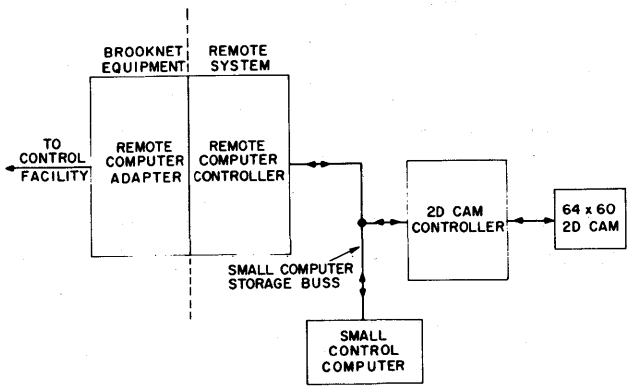


Figure 6—Remote system for 2 DCAM

which is attachable to Brooknet appears in Figure 6. This configuration represents a departure from the initial intent of the Brooknet system, which was to extend the considerable resources of the 6600 computer to smaller remote processors for controlled periods of time. In this case, however, it is the remote which possesses the desired capability of which the central processors would like to avail themselves.

APPLICATIONS

The multidimensional approach presented can be adapted to many large computer installations provided that a mass core storage device is included. The Brookhaven configuration was offered as an example which utilizes facilities which were already available at that particular computing center. This scheme for providing content addressable capability has applications wherever a conventional two-dimensional configuration was previously considered, provided that the decrease in speed incurred is not a severe restriction.[7]

By partitioning the associative field of the memory word into sections one can structure his data array so that multiple associations are required, and thus take advantage of the multi-dimensional approach. For instance, if a word has an eighteen bit associative field, it can be partitioned into three 6-bit operands. A match in the first plane will define a new plane wherein the associative field of all words have the same first operand as produced the match. Now it is no longer necessary to scrutinize the first operand so the match operation is performed on the second operand and so on until the unique word is found. Implementation of such a procedure presupposes a certain order of structuring when the data block was first introduced to the virtual CAM area. Self sorting may be accomplished by using part of the data word as the address and preserving

the old address as a sequence number when first storing the data into the mass memory. Applications in pattern recognition, storage paging, and table look-up routines lend themselves to such structuring quite readily, and, if the application is such that the associative file must be loaded once and then referenced many times, the presorting requirement does not introduce an objectionable amount of overhead at all.[8] Indeed, in the Illiac 3 pattern recognition processor, the Iterative Array and Transfer Memory taken together achieved the n-cube effect in three dimensions.[9]

The multidimensional approach appears particularly attractive to those applications which can be thought of as requiring a hierarchy of associations, such as finding a record in a complex paging scheme (i.e., a file within a page, a sector within a file, and a record within a sector), or where the file itself is multidimensional in structure, such as for graphic or mechanical information where $f(x, y, z)$ varies with time. In addition, new applications might open up where the amount of source information is so large that up to now consideration of a truly associative system would have been prohibitively expensive.[4,10]

CONCLUSION

Despite some restrictions on its class of applications, the cost performance characteristics of this approach have been considered attractive enough to warrant implementation, especially in large multiprocessing systems where the structure of system tables and the traffic in file accessing represents a considerable portion of the processing time. The implications of large scale integration and emergence of more and more complex modules suggest that other capabilities besides content addressability may benefit from a similar approach to effective utilization.

This system is currently being implemented at Brookhaven National Laboratory.

ACKNOWLEDGMENTS

The author wishes to thank Dr. Y. Shimamoto of Brookhaven National Laboratory and Dr. J. Robertson of the University of Illinois for their suggestions and encouragement.

This system is currently being implemented by Niels Schumburg and Bernard Garfinkel of the Applied Mathematics Department engineering group at Brookhaven National Laboratory.

REFERENCES

1 W F CHOW
Plated wire content-addressable memories with bit-steering technique
IEEE Transactions on Electronic Computers Vol EC-16 No 5 October 1967

2 P M DAVIES
A simplified superconductive associative memory
Proceedings SJCC May 1962

3 W L McDERMID H E PETERSEN
A magnetic associative memory system
IBM Journal of Research and Development January 1961

4 ASPINALL KINNITMENT EDWARDS
An integrated associative memory matrix
IFIP Congress August 1968

5 *6400/6500/6600 Extended core storage system reference manual*
Control Data Corporation Publication No 60225100 1968

6 K R FANNIN
Brookhaven digital communications network
AEC Computer Information Meeting Rice University Houston Texas April 7 1967

7 DUGAN GREEN MINKEN SHINDLE
A study of the utility of associative memory processors
Proceedings—ACM National Meeting 1966

8 R R SEEBER A B LINDQUIST
Associative memory with ordered retrieval
IBM Journal of Research and Development January 1962

9 B H McCORMICK
The Illinois pattern recognition computer—Illiac III
IEEE Transactions on Electronic Computers Vol EC-12 No 5 December 1963

10 ASPINALL KINNITMENT EDWARDS
Associative memories in large computer systems
IFIP Congress August 1968

Interrupt processing with queued content-addressable memories

by JERRY D. ERWIN and E. DOUGLAS JENSEN

Southern Methodist University
Dallas, Texas

INTRODUCTION

One of the most significant problems in designing high performance computing systems is the complexity of the associated supervisory software. This is especially true in multi-user environments: the software overhead involved in user communications and resource allocation normally absorbs a great percentage of the system's computing power.

An often-proposed solution to this problem is to remove some of the time-consuming executive functions from the software and perform them in hardware. Numerous examples of this operation are visible throughout the history of computer development.[1] These attempts have met with varying degrees of success because as the tasks to be transplanted become more and more comprehensive, it becomes less and less obvious exactly what sort of hardware structures are needed for their efficient implementation.

One of the most vital ingredients in an on-line computer is a powerful and flexible priority interrupt system. More than any other single feature, the interrupt structure determines the capability of the machine to respond quickly to both internal and external stimuli. Whether the computer is used for control or data processing, its effectiveness is frequently measured by how rapidly it is able to react to conditions in the user environment.

A fundamental characteristic of an interrupt system is how many interrupting sources it can handle. Few of today's computers are designed to accommodate interrupts from a large number (say hundreds) of devices, despite the growing requirements for such facilities. It is increasingly common to find cases where a machine's processing power is sufficient to satisfy a great many concurrent demands, but its interrupt system is incapable of supporting the corresponding volume of service requests. This limitation lies not only in the hardware restrictions on the number of interrupt lines, but also in the software's inability to effectively deal with the interrupts it does get. It is apparent, then, that in such cases a brute force extension of current interrupt concepts is inadequate. Instead, a new approach needs to be formulated which provides improved performance from both hardware and software.

It is well recognized that on-line computers generally cannot afford the inefficiencies inherent in scanned or multiplexed interrupt structures; a multi-level hierarchical system is preferable for such applications.[2] Frequently the on-line interrupts do not all have distinct priorities, so they can be separated into priority classes. Each class can then be assigned to some priority level which is shared by all interrupts of that class. Unfortunately, most machines do not possess an efficient means of identifying different interrupt sources within a priority level, and so the user must either degrade his system performance or undergo the expense of additional priority level hardware.

The assignment of priority levels to particular functions can be a complex task since the interrupt requests must be ordered on the basis of their interaction, not merely on their relative importance.[2] However, it is difficult to accurately forecast the exact nature of these interactions in advance, especially since the context tends to vary widely during system operation. The resulting assignment compromises can be avoided by supplying the freedom to dynamically reallocate priority levels, a powerful tool which enables the executive software to accurately establish the computer's response to changes in its environment. This capability is currently approached through some combination of arm/disarm commands, program-generated interrupts, multiple level assignments, and hardware switching matrices.

One of the disabilities of conventional priority

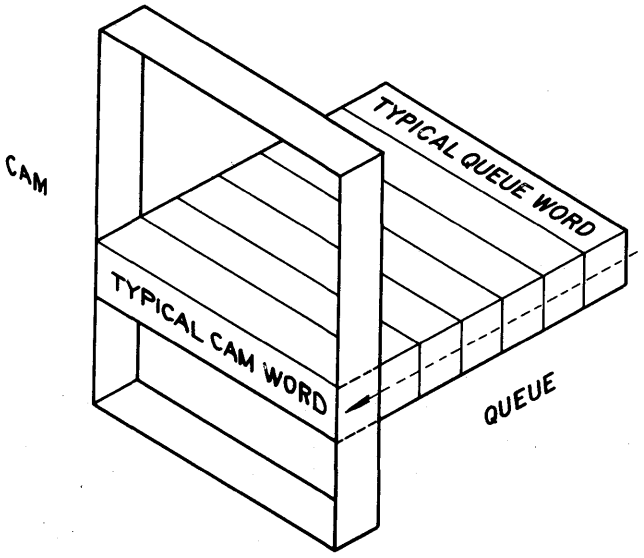


Figure 1—Queued CAM

interrupt schemes is that if an interrupt at some level is waiting, or is being serviced, or is suspended due to higher priority requests, subsequent interrrupts at that same level will be ignored. It is obviously necessary for a system to be designed so that the time required to respond to an interrupt is less than the *average* time between occurrences of that interrupt. But it is also highly desirable that the system not become saturated by occasional bursts of interrupts at any level. This tolerance to interrupt asynchronism greatly eases the problem of compatible priority assignments, and lessens the risk of disrupted communications between the machine and its environment.

These considerations have all been successfully addressed in the design of a special purpose Interrupt Processor (IP). The IP functions as an autonomous element of medium to large scale on-line multiprocessing systems, and facilitates the computer's interaction with users, peripherals, and control interfaces. All of the functions associated with detecting, acknowledging, and scheduling interrupts have been incorporated into the IP hardware, providing centralized routing of interrupts to all other processors. The remainder of the task assignment mechanism may be software in the destination processor[3,4] or a hardware extension of the IP.[5]

The IP monitors a large number of interrupt lines on which signals are automatically identified by source within software-defined priority levels. When interrupts are received they are organized and stored on the basis of priority and order of occurrence. This assures optimum priority response, and reduces supervisory software overhead leaving more processing power for the users.

The IP is organized around a special unit called a queued content-addressable memory, which forms its primary storage and processing facility.

THE QUEUED CONTENT-ADDRESSABLE MEMORY

The concept of a queue is a familiar one to hardware and software designers alike. As modeled in software, queues are usually variable length, each new word being added directly behind the last one. Hardware queues, on the other hand, are fixed length as exemplified by a special shift register in which every new entry is inserted at one end from where it moves up to the forward-most empty position.

Content-addressable memories (CAM's) are also well-known, but not so widely found, due to technological limitations which are now being overcome.

Queues and CAM's are not necessarily disjoint structures, but can be combined into one entity having both kinds of characteristics. As shown in Figure 1, this results in a wordwise 2-dimensional memory consisting of a CAM with a queue behind every word. To enter a new item into the memory, the fronts of the queues (i.e., the CAM words) are associatively searched for a current entry having a key that matches that of the new item. If there is none, the new word is placed directly in a vacant CAM slot. If a word with a matching key is found in the front of some queue, the new item is entered into the rear of that queue and allowed to ripple forward. To read from the memory, the CAM is interrogated for a word with the desired key. If one is present, it appears on the output lines. If an entry is removed, the remaining words in the corresponding queue all move forward one position, filling the vacated CAM position.

INTERRUPT PROCESSOR ORGANIZATION

For purposes of discussion, it is assumed that the IP is part of a 32-bit computer, and that it interfaces with both the Central Processor (CP) and the main memory. Only one CP is mentioned, but the IP is readily adapted to multiple CP's in the fashion described in References 3 and 4, or 5.

The basic IP configuration shown here provides for 64 levels of priority with 16 hardware-identified sources per level, for a total of 1024 interrupts. This may be

expanded in increments of 64 levels to a maximum of 256 levels and 4096 interrupt sources.

A block diagram of the IP is illustrated in Figure 2. The major components are a priority structure, a random access scratchpad, and a queued CAM.

Priority logic

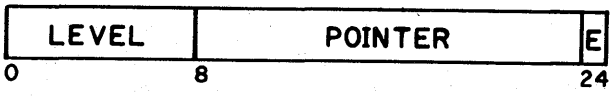
A fully nested priority tree monitors the 64 to 256 interrupt lines. This tree determines the order in which interrupts are accepted and stored for processing, as contrasted with the program-controlled priority in which the stored interrupts are serviced by the CP.

The priority logic incorporates a holding register that frees the device from having to maintain the interrupt signal. Also included is an arm register with a bit for each line which may be set and reset both unitarily and in groups.

The highest priority line which is armed and active is encoded by the tree into an 8-bit binary number. This number is transferred to the Scratchpad Address Register (SAR), and the corresponding 4-bit source identification code is stored in part of the queued CAM input data register. A latch is then set to return an acknowledge signal to the interrupting device. This latch also removes the interrupt line from the priority tree until the interrupt is reset by the device.

Scratchpad

The scratchpad is a high speed random access memory which contains one word for each of the 64 to 256 interrupt lines. The words are 25 bits long and formatted as follows:



The 8-bit level field indicates the priority assigned to the associated interrupt line, and can be altered under program control. More than one interrupt line can be placed on the same level if desired, and under certain conditions the waiting requests from a given line may have more than one priority.

The 16-bit pointer is used by the CP to form the address of the appropriate interrupt service routine. This frees the computer from having fixed interrupt entry locations in main memory.

The E bit is both a unitary and group enable bit. Considered as part of a scratchpad word, it can be set or reset unitarily. However, the scratchpad is also sideways addressable in its least significant bit position. Viewed together as a vertical word, the 64 to 256 E bits are divided into 8-bit groups. Each group may be set to all ones or reset to all zeros according to the values of the corresponding bits in a 32-bit word supplied by a CP instruction. Interrupt requests received on a line which is armed but disabled will be accepted and queued up, but will not be processed until the line is enabled.

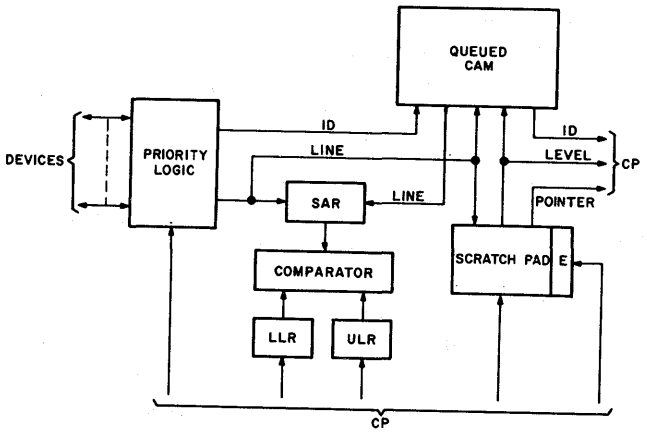


Figure 2—Interrupt processor block diagram

Scratchpad read requests come from either the priority logic or the queued CAM, with preference given to the latter in the event of conflict. In all cases, the scratchpad is addressed by line number.

There are two means of altering the scratchpad's contents (in addition to the sideways addressability of all the E bits). A single word may be replaced, or any sequence of contiguous entries may be reloaded from an image in main memory. In either case, only one CP instruction execution is required. During block updates, the boundaries of the affected area are defined by a pair of 8-bit upper (ULR) and lower (LLR) limit registers. Once a transfer commences, the upper address remains fixed while the lower address is incremented as each new word is written. A comparator not only detects the end of the operation but also monitors the scratchpad address register. All attempts by the priority logic and queued CAM to reference within the boundaries of the limit registers are subject to restrictions specified in the CP instruction which initiated the scratchpad modification. The scratchpad may be copied into main memory without imposing access restrictions on the priority logic and queued CAM.

Queued CAM

The heart of the IP is a queued CAM, as illustrated in Figure 1. It contains at least eight words of content-

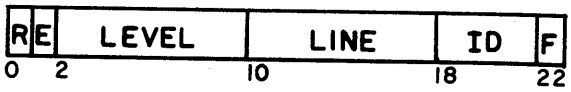
addressable memory, expandable in increments of four words to a maximum of 64. Every CAM word is backed up by an 8-word queue, each of which is individually expandable to 32 words in groups of eight.

Each CAM word and its affiliated queue is dedicated to a particular interrupt level as long as that level is armed and has an active or waiting interrupt request. Multiple interrupts on a single level, whether from the same or different sources, are lined up in the queue for that level in order of occurrence. Thus, the CAM size represents the maximum possible number of simultaneously active levels, and the length of a given queue represents the maximum possible number of simultaneously active sources at that level. The relationships between these figures and the number of implemented interrupt levels are parameters dependent on the specific system application and performance requirements.

Each queue entry is 21 bits wide with the following format:

The priority level is obtained from the scratchpad; the line number and source identification (ID) are received from the priority logic. The F (Filled) bit is set if the word contains meaningful data, and is the mechanism by which entries are automatically shifted forward in the queue. When a word is accessed in the CAM, it may remain there or it may be deleted by resetting its F bit.

The CAM entries are 23 bits wide, as follows:



The R (Running) bit being set signifies that the service routine for this level has been activated in the CP. The E (Enable) bit in the CAM is updated as each new request enters the CAM and whenever its counterpart changes in the scratchpad. The other fields are the same as in the queue.

Almost all searches in the CAM are conducted on the level field (or the combined E and level fields), which eliminates the possibility of multiple matches since CAM entries are uniquely allotted on the basis of priority level. The CAM may be interrogated by line number in order to clear the waiting interrupt requests generated by a particular line. It is possible to get multiple matches in this situation if the priority of the specified line has been recently changed. However, the multiple matches are of no consequence since they are all simply cleared. When the CAM F bits are associatively tested to locate a vacant CAM word, the choice among multiple candidates is made by logic which selects the responding location having the highest conventional address.

When the CAM is interrogated on the juxtaposed E and level fields, what is sought is not an exact match as in other cases, but rather the word having the largest numerical value in those fields. This process is accomplished in the fashion described in Reference 6.

INTERRUPT PROCESSOR OPERATION

IP operation can be divided into three independent phases: auxiliary functions such as changing the scratchpad contents (described earlier) or arming and enabling interrupts; inputting interrupts to the queued CAM; and outputting service requests to the CP.

Arming and enabling interrupts

Interrupts are armed and enabled by line number rather than level since priority levels are program-assigned. Interrupt lines may be armed, disarmed, enabled, and disabled both unitarily and in groups.

Unitary arming and disarming is accomplished with the arm register in the priority logic. The arm register bits are individually addressable and may be altered by transmitting one or more words from main memory.

The arm register can also be controlled in 8-line groups. Each 8-line group is represented by a single bit in a 8-bit (for the minimum 64-line configuration) to 32-bit (for the full 256 lines) control word. This allows all 32 8-line interrupt groups to be armed and/or disarmed in any combination with a single CP instruction execution.

Interrupts are enabled and disabled both unitarily and in groups through the scratchpad (and indirectly through the CAM). The scratchpad entry for each line includes an Enable (E) bit, so unitary enabling and disabling is performed by rewriting scratchpad words as described earlier.

Since the scratchpad is also sideways addressable in its LSB position, all 32 8-line interrupt groups can be enabled and/or disabled in any combination with a single CP instruction execution.

The Enable (E) bit in the CAM is updated whenever its counterpart in the scratchpad is altered. If a word is changed in the scratchpad, its line and level fields

are used to search the CAM for a corresponding entry. If one is found, its E bit is modified to agree with the new value of the scratchpad E bit. In the case of a group operation on the combined scratchpad E bits, all those levels in the groups affected by the instruction would be subject to inspection.

Since the IP components function autonomously, facilities are available to provide program control over the activity of those lines whose parameters are being altered in the scratchpad and the arm register. The modifications (such as enable/disable, arm/disarm, priority reassignment, or a new pointer) are immediately reflected in any waiting requests. Alternatively these requests may be cleared, the modification taking effect only with subsequent new interrupts. The CP instructions which incur these changes include the capability to specify the following modes of operation:

a. Normal operation—the queued CAM continues to accept and process interrupt requests on all armed lines as usual.
b. Clear affected entries—each line changed in the scratchpad or arm register is looked up in the CAM, and if found the corresponding queue is completely cleared.
c. Hold all inputs—the queued CAM continues to process the waiting interrupt requests, but no new requests are accepted on any lines until all initiated changes have been completed. This is accomplished by inhibiting the priority logic response to interrupt signals.
d. Hold affected inputs—the queued CAM continues to process the waiting interrupt requests, but no new requests are accepted on any lines for which changes have been initiated but not completed. A comparator on the scratchpad address register causes those lines which fall between the current values of the scratchpad upper and lower limit registers to be considered disarmed.
e. Hold all outputs—the queued CAM continues to accept new interrupt requests, but no further CP service notifications are made for any lines until all initiated changes have been completed.
f. Hold affected outputs—the queued CAM continues to accept new interrupt requests, but further CP service notifications are made for lines for which changes have been initiated but not completed. The scratchpad address register comparator inhibits the IP from signaling the CP in response to interrupt requests on those lines which fall between the current values of the scratchpad upper and lower limit registers. Any request which is inhibited this way also blocks all lower priority requests.

These CP instruction options are coded to allow combinations of input/output modes. The "hold" modes are applicable principally to cases where the scratchpad is being loaded from an image in main memory, or where a group enable in the scratchpad causes several CAM updates.
input/output modes. The "hold" modes are applicable principally to cases where the scratchpad is being loaded from an image in main memory, or where a group enable in the scratchpad causes several CAM updates.

Detecting and acknowledging interrupts

When an interrupt signal is detected, it is loaded into the appropriate bit of the holding register. If the line is armed, it is gated to the priority tree where the highest priority line is encoded into an 8-bit scratchpad address. The source ID is stored in the queued CAM input data register, an acknowledge signal is returned to the device, and the line is disconnected from the priority tree until the interrupt is reset. Since the scratchpad gives precedence to the queued CAM, the priority logic may have to wait one cycle for service. When access is granted, the program-assigned priority level is retrieved and included with the source ID and line number in the queued CAM input data register.

Storing and scheduling interrupts

The word in the input data register is entered into the queued CAM by first associatively checking the level of the new interrupt request against those already there. If a match is found, this level is already assigned to a word in the CAM, and the new request is loaded into the rear of the corresponding queue. If there is no match, the request is placed in a vacant CAM slot. The CAM is then associatively searched for maximum on the combined E and level fields. If the entry retrieved is enabled but inactive (i.e., E set and R reset), the CP is trapped to initiate the new routine. Otherwise, no further action is necessary.

Any time that data entry is attempted and the CAM or queue is filled, a high priority executive trap occurs in the CP. The supervisory software then may arrange to simulate a portion of the queued CAM in main memory to handle the overflow at reduced rates, or it may decide to reduce the I/O traffic to within the queued CAM's hardware capabilities.

Initiating CP service

There are three conditions under which the IP notifies the CP to initiate the processing of an interrupt service routine. The first occurs when a new interrupt request being loaded into the CAM is both enabled and higher priority than the currently active request. The second case is when the CP enables a level which is higher priority than the currently active level and which has a request pending in the CAM. Last is the completion of a higher priority interrupt routine when a lower priority request is enabled and waiting.

When one of these events occurs, the relevant line number from the queued CAM is put in the scratchpad address register. When access is granted, the IP signals the CP, sending it the priority level, source ID, and service routine pointer. These parameters presume a software supervisor in the CP, but are readily extended to interface with hardware-resident schedulers.[5]

Terminating CP service

When execution of an interrupt service routine is completed, the CP returns the priority level of that routine to the IP. The corresponding CAM entry is deleted by resetting its F bit, allowing any pending request in the queue behind it to move into the vacated position. The E bit of the new request is updated from the scratchpad, and a new associative search for the highest priority waiting request is then initiated as described earlier.

Interrupt processor implementation

The IP consists primarily of memories which are inherently regular and thus readily lend themselves to economical batch fabrication. Implementation is entirely feasible with the level of integration available today in TTL, and will be further facilitated by impending improvements in semiconductor technology.

An MSI 8-input priority encoder circuit simplifies the task of detecting and recognizing interrupts. Three levels of these encoders are cascaded to attain the maximum 256-line fan-in. A high speed multiplexing scheme could be employed instead of the priority logic since both approaches establish a somewhat arbitrary order in which interrupts are accepted and stored. However, cost/performance tradeoffs favor the hierarchical technique with the logic functions currently available.

The scratchpad is conveniently constructed from internally decoded 64-word by 4-bit random access memory modules. The associated address, data, and limit registers, as well as the comparator, are also composed of standard MSI circuits.

A substantial reduction in hardware was achieved by adapting the CAM to take advantage of an existing associative memory element. Six of these 4-word by 4-bit blocks are needed for every 4-word by 23-bit CAM increment, plus some auxiliary logic to tie them together. The auxiliary logic would be included on the chip if a custom LSI circuit were fabricated for this application.

The 8-word by 21-bit queue behind each CAM position comprises sixteen 10-bit register packages plus a more general 8-bit register for the F bits. Again, the extra logic needed could be integrated into a special queue module.

Control and interface logic constitutes the remainder of the IP, or about 10 percent of the total hardware.

A maximum 256-line configuration having a 32-word CAM with 8-word queues consists of under 1500 IC's. Maintaining the same relative sizes of the CAM and the queues, a minimum 64-line IP would include only about 500 IC's. This suggests that even an expanded IP could be purchased for less than the price of a conventional I/O channel on many computers. The probable impact of eventual LSI implementation would be to reduce the chip count by at least an order of magnitude.

INTERRUPT PROCESSOR PERFORMANCE

There are four criteria commonly used to judge the performance of an interrupt system: reaction time, overhead, optimum priority response, and saturation.[2]

Reaction time is the time between the occurrence of an external interrupt signal and the commencement of the first useful instruction in response to that signal. (This interrupt is understood to require that the CP pursue a higher priority task than is presently under way.) A maximum of 5.0 microseconds elapse in a 256-line IP from the moment an interrupt occurs until the CP is notified. This period is nominally 2.5 microseconds but the higher figure arises from worst case synchronization of the priority logic. Subsequent interrupts can be accepted by the IP every 1.4 microseconds and can also be sent to the CP at the same speed. The remainder of the computer's reaction time will be contributed by status preservation activities in the CP. A contemporary machine having a multilevel priority structure may be able to alert the CP this quickly, but it cannot so easily divorce itself from the CP's participation. When the liabilities of conventional interrupt hardware are compensated for in

the software, the supervisory bookkeeping can far outweigh the hardware delays.

Similar considerations apply to the interrupt completion procedure which is the second half of the interrupt system overhead. This overhead is defined as the difference between the time needed to completely process the incoming request and the execution time of all useful instructions. No more than 1.4 microseconds transpire when a 256-line IP terminates an interrupt, for a total IP overhead of 2.8 microseconds per complete interrupt cycle. Thus the IP can support a sustained throughput rate of almost 400,000 interrupts per second. Additional overhead factors in the machine are diminished by the power of the IP, but will be determined by the integrated hardware and software design of the CP's executive structure.

The independence of the IP's input and output functions implies occasional conflicts in accessing the scratchpad. These are resolved in favor of the queued CAM but the priority logic will not have to wait more than one scratchpad cycle since the CAM cannot supply consecutive CP service requests at the scratchpad cycle rate. In the worst case only one scratchpad cycle time (about 50 nanoseconds) is added to the IP input time.

The priority logic and queued CAM must also contend with occasional scratchpad updates by the CP. This is the lowest priority scratchpad function unless the acting CP instruction specifies one of the hold modes during updating. However, some interference may be precipitated by examination/modification of the CAM E bits as a result of scratchpad E bit changes. This alteration occurs in one CAM cycle time (about 100 nanoseconds) per affected entry.

Optimum priority response is a measure of the extent to which the computer is always executing the most important task as determined by the environment. The utilization of an autonomous IP for centralized control of interrupts assures that the other system processors are always devoted to the highest priority jobs without diverting their efforts to evaluate and manipulate interrupts.

To maintain accuracy in the priority scheduling, it is necessary that the lines which require very fast reaction time be attached to higher positions in the priority tree. Once in the queued CAM, all interrupts are served on the basis of their program-assigned priorities.

Saturation occurs when the system cannot respond quickly enough to all of the interrupts causing some of them to be disregarded. Protection against this is inherently supplied by the queuing that occurs in the IP.

CONCLUSION

A unique Interrupt Processor has been described which uses hardware to perform many of the interrupt handling functions heretofore dealt with by software in large on-line multiuser systems. The operation of this unit has been described and a brief look at its implementation and performance has been given.

The most significant aspect of the IP is the queued content-addressable memory which provides an efficient interrupt organization and storage facility. This queued CAM concept should also prove to be highly effective in many other hardware solutions to supervisory system problems normally handled by software.

REFERENCES

1 S ROSEN
Hardware design reflecting software requirements
Proc FJCC 1968 pp 1443-1449

2 E R BORGERS
Characterics of priority interrupts
Datamation June 1965 pp 31-34

3 R J GOUNTANIS and N L VISS
A method of processor selection for interrupt handling in a multiprocessor system
Proc IEEE December 1966 pp 1812-1819

4 R J GOUNTANIS and N L VISS
Methods of interrupt routing in a multiprocessor system
IEEE Twin Cities Section November 1967

5 B W LAMPSON
A scheduling philosophy for multiprocessing systems
CACM May 1968 pp 347-360

6 M H LEWIN
Retrieval of ordered lists from a content-addressed memory
RCA Journal June 1962 pp 215-229

7 J G BENNET
Letter
Datamation October 1965 p 13

8 R V BOCK
An interrupt control for the B5000 data processor system
Proc FJCC 1963 pp 229-241

A language-oriented computer design*

by CLAY McFARLAND

First Business Computing Corporation
Dallas, Texas

INTRODUCTION

Learning to program in a general-purpose, high-level language is a formidable task for a person who simply wishes to use the computer to solve his problems. He must learn how to express his problems in algorithmic form in the language, the constraints and idiosyncrasies of the language, and the mechanics of running a program on his computer. If he wishes his programs to be efficient, he must learn which constructions in the language use the machine effectively and which do not. This is complicated by the unpleasant fact that effectiveness in the language may not correspond to effectiveness in the machine. A concise, well constructed statement may use much more machine time than an ungainly structure that does the same thing.

Part of this learning process is quite beneficial; for example, being forced to state a problem precisely enough for a computer solution can cause a significant increase in understanding of the problem. System programmers can ease the user's task by providing a language whose terminology is close to that of the types of problems to be solved, or by constructing a simple job-control language. Difficulties in using computers are often the result of either poor system design or a mismatch between the language and the computer being used. Thus, while it is the user's responsibility to produce good algorithms for solving his problem, he is entitled to assume that a good construction in the language he is using will produce an efficient program in the machine.

Computer users can be roughly divided into four classes:

1. Non-programmers
2. Occasional users
3. Professional programmers
4. System implementors

Historically, very little has been done to make the computer usable by class 1 and class 2 users; questionnaire programs and languages such as JOSS and a subset of APL are changing this to some degree. Computers and systems have traditionally been more or less optimized for the class 3 user, although this is probably accidental in most cases. Historically, the only consistent optimization has been the minimization of machine hardware.

Unfortunately, on most systems the class 4 user is forced to design his structures using bits and fields in machine words, and to design his systems to be written in assembly language. Special languages such as APL, LISP and the various compiler-writing languages have been developed for this user, but these languages have been designed for a narrow range of information structures and a slightly wider class of problems. The only general-purpose languages that have met the needs of the sophisticated user are Burroughs extended ALGOL and PL/I. While it is feasible to write an operating system in PL/I, there has been a large loss in efficiency; this loss is justifiable only in a system of the order of magnitude of MULTICS, which simply could not have been written in assembly language. Systems written in extended ALGOL, on the other hand, are among the most efficient in existence. Table I shows the results of a comparison between the Burroughs B5500 and IBM 360/40 for an ordinary numerical problem. The test was run by W. M. McKeeman. The reason for this performance difference is straightforward: the structure of PL/I does not match the structure of either System 360 or the GE 645, while the Burroughs machines faithfully mirror the structure of extended ALGOL.

This last class of user is often producing systems that will be used by many other users of all classes. It is therefore of primary importance to optimize the machine hardware for this user. To do this, the hardware,

* A portion of this research was supported by Air Force Avionics Laboratory Contract F-33615-68-C-1682, at Texas Instruments, Inc., Dallas, Texas.

TABLE I—Operating System Comparison
Burroughs B5500 vs IBM 360/40

	Debug	Read	Compile	Link-Edit	Run
B5500 (ALGOL)	2 days	<1 sec.	3 sec.	none	7 sec.
360/40 (PL/I)	2 months	22 sec.	180 sec.	69 sec.	22 sec.

	Memory Cycle	Word Length	Floating Multiply
B5500	6 μs	48 bits	30 μs
360/40	2.5 μs	16 bits	80 μs

operating system, and primary languages of a system should be designed as a unit. This implies that machine hardware should be designed to make it very easy to produce powerful programming languages, and to allow the production of an operating system that presents a simple, understandable face to the user. The hardware-language-system combination should be designed to minimize the total cost of production of the application programs of the class 3 user; this cost includes programming time, machine time, and the cost to a project to any delays in debugging, etc. As a side effect, a system designed in this way will provide aid and comfort to the class 1 and class 2 users by reducing the cost of their answers.

The hardware in a total system design would have the following characteristics:

1. Information structures used by class 3 and class 4 users would be basic structures accessible to a machine language; operators normally used on these structures would be built in as machine instructions.
2. Control flow and data access methods in the machine would be the same as in its basic languages and its operating system.
3. The system would perform implicit functions for users, but would avoid producing unexpected side effects. Further, any implicit functions could be overridden by the user. In other words, the system would appear simple to the casual user, and as complex as desired to the sophisticated user.

The remainder of this paper will describe a language and computer structure designed to meet the criteria discussed above. The programming language for the system is called simply "the programming language" (TPL); the computer system is named the Hydra. Both the computer and language are experimental and in a state of continual change. The system is described as it is conceived at this writing.

TPL

The aim of the TPL design is to produce a language that will be an aid to the programmer in thinking about his problem and the processes he is using to solve the problem. Our purposes are quite similar to those of the ELF project of Cheatham, Fischer and Jorrand.[3] However, we are not constrained, as they were, by the necessity of producing a language compatible with current computing systems. Many of the constructs in TPL are the same as or similar to the corresponding BASEL (the base language of ELF) construct, but the parts of the languages that deal with their machine environment are quite different. The operator structure of TPL is a generalization and expansion of the Iverson notation data operators[9] and the control operators of EULER[18] and Linear C.[11] The data operators are essentially those in the appendix of *A Programming Language*, and the control operators have been extended to all compatible data structures.

Several departures from the traditional structure of expression-oriented languages have been made.

1. The *go to* operator has been eliminated. This should make programs easier to read and good programs easier to construct. Without the *go to*, labels are unnecessary, and without labels the syntactic type ⟨statement⟩ is unnecessary.
2. The assignment expression has been restricted. An assignment can change the value of variables defined in its block at its own level and no others.

These two changes have many good effects. Their

only bad effect is making coroutines more difficult to construct (though possibly easier to understand after they are constructed). Although programming without *go to*'s is unnatural when first tried, the resulting programs are invariably superior in both structure and readability to conventional programs.

3. A process is a basic data type, and includes the concepts of macros, subroutines, and procedures. Functionally, these structures differ only in binding their variables to a fixed reference at compile-time, load-time and run-time, respectively. The differences in binding time can be explicitly controlled by the programmer. Processes can be called implicitly, i.e.,

   $$A \leftarrow B+C$$

   where B is type *real* and C is type *process*, would cause C to be executed and the value it returns checked for compatibility with B. If it were compatible, A would be marked *defined*, and assigned the value and attributes of the result. If the value of C were a *process*, it would be executed, and so on.
4. The operator-operand structure of TPL is quite regular. Operators are valid only for strictly compatible data types (i.e., ">" and "<" are not defined for logical variables, *true* is not equivalent to the number 1, etc.), although obvious automatic conversions such as *integer* to *real*, *real* to *complex*, etc., are provided. If an expression is valid, it remains valid if variables of any structure class are replaced by
   A. A process, block, or expression whose value is a variable of the same type.
   B. An implicit process call, if a variable of type *process* is not compatible.
   C. A variable of a different structure class that is compatible mathematically with the expression.

The strict application of (3) and (4) create some collisions. For example, suppose we write

a←*begin new* c . . . *end*

and we want a to be type *process*, with the value given by the program on the right of the assignment. However, the program will be executed and its value assigned to a. To allow the former option, we have added an operator, "...," which protects its operand from immediate execution. The statement

a←"*begin new* c, . . . *end*"

would produce the former result.

Program structure

The syntax for TPL is shown in the Appendix. In TPL, declarations are expressions to be evaluated at compile-time. A simple declaration, such as

new b,

establishes the scope of b as starting with the block containing the declaration; storage for b will be allocated when b is given a value, and freed when the block is no longer active. If a value or attributes are assigned to a variable, as

new integer c←6,

storage is allocated for c when it is given a value (in this case, at compile-time), and freed when the main program of which c is a part is no longer active.

Expressions that are not part of declarations are evaluated and replaced by their value as soon as their variables become fixed in value. If an expression's variables are fixed at compile-time, the expression functions as a macro. The only explicit transfers of control in TPL are a *repeat* operator, which repeats the innermost block or compound expression in which it is imbedded, and an *exit*, which causes execution to resume at the first instruction past its block or compound statement. Every expression in TPL has a value, which is placed on top of a system-value stack. Operators manipulate the stack implicitly, and the top of the stack (the value of the last expression) can be referenced explicitly with the reserved identifier *sysval*. An expression can be terminated with a semicolon, colon, or period, which respectively throw away, save, and execute the system-value.

Figure 1 is an example of an ALGOL-like TPL program, the Euler precedence parse. This is an exact copy of the Euler algorithm as presented in Part I of (15), with precedence relations represented correctly as matrix entries. The program in Figure 2 improves on the algorithm by producing a vector of character vectors, eliminating the necessity to scan both forward and backward for relations. The program in Figure 3 is the same as in Figure 2, but it uses the implicit TPL value stack to avoid explicit temporary storage.

Data

Each data item in a TPL program has a value and several attributes, including data *type*, structure *class*, *defined*, *size*, *length*, etc. The data types currently al-

```
    begin new vector s; new integer i; j; k;
(1)     s←p[0]; i←0; k←1;
(2)     While p[k] ≠ '⊥' do
        (i←j←i + 1; s[i]←p[k]; k←k + 1;
(3)       while m[s[i] : p[k]]= 3 do
(4)           (while m[s[j-1]; s[j]]= 2 do j←j-1;
                s[j]←leftpart (ω(i-j+1)/s); i←j;
           )
      )
    end
```

1. Initialize string indices
2. Until end symbol (⊥) increment pointers to s get next input symbol, increment input index.
3. Perform the next statement only if s[i] ◇p[k]
4. Back up j to last ◇, reduce resulting string, reset i.

Relations in Figures 1,2,3 are coded

◇ = 1

≐ = 2

◇ = 3

Figure 1—The EULER parsing algorithm

lowed in TPL are:

1. Logical
2. Integer
3. Real
4. Character
5. Reference
6. Process
7. Pattern
8. Mixed.

The first four types are conventional. A variable of type *reference* simply points to another variable. A variable of type *process* is a macro, subroutine, or procedure. A variable of type *pattern* has attributes but no value, and is used in creating other similar structures. A *mixed* variable is a structure other than *scalar* where the elements may vary in type. These types may be expanded (although not necessarily in TPL itself) to include other types desired by the programmer.

The structure classes currently allowed in TPL are:

1. Scalar
2. Vector
3. Matrix
4. Array
5. Tree
6. List
7. Set.

Mathematically, there is a lot of redundancy in this set of structures, but the redundancy mostly disappears when the machine and operating system in which the language is imbedded are considered. For example, a *matrix* may be sparse or triangular, and the storage allocation may differ for these special cases. A *set* is a collection of elements with no implication as to ordering or any other relationship between the members of the *set*. A *set* is stored in lexical order to make set operations more convenient. A *tree* is stored in end-order with an index, and storage allocation methods for a *list* have special provisions for self-containing lists, etc.

Each variable has a definition attribute. A scalar may be *defined* or *undefined*; a variable with more than one element may also be *partially defined*. Other attributes are *length* of each element of a variable in units appropriate to the variable, *size* of each dimension of a variable, and various special attributes for variables of type *process*.

```
    begin new vector s; integer i; k; x;
(1)     process parse←"case x←m[ω(1)/s[i-1]: s[i]] of
(2)       i←i + 1;
(3)       s[i-1]←s[i-1], s[i];
(4)       (s[i-1]←leftpart (s[i-1]); i←i-1;)
                  end
          if x = 3 then repeat;"
        s←p[0]; i←1; k←1;
        while k<size p do
        (s←s, p[k]; parse . k←k+1;)
    end
```

1. parse is a case statement on the precedence relation code
2. ◇ , leave symbol as a single vector element, increment pointer
3. ≐, concatenate symbol with previous vector element
4. ◇, s[i-1] can be replaced with its leftpart (the case is then repeated to compare s[i-2] and leftpart (s[i]).

Figure 2—Modified EULER parsing algorithm

Attributes are manipulated both implicitly and explicitly. Any operation includes checking of its operands to insure that they are compatible with the operation, and any conversions that may be necessary. In

a←b+c

where b is *real*, c is *integer* and a is *undefined*, the system will float c, add the result to b, allocate storage for a, store the result of b+c in a, and set a to *defined, real.* If c were type *logical*, an exit would be taken to the currently active error procedure. At the system level, an error- and interrupt-handling procedure is predefined. The programmer can change this for any block by writing

error←thiserr

and defining thiserr to be any error-handling mechanism he wishes. Error-handling procedures have access to all system parameters and to an error vector which indicates the error or errors that have occurred. The programmer can also directly access attributes by writing

size (a) (a structure containing an integer for each dimension of a),

length (a[i]) (an integer),

defined (b) (a *logical* variable), etc.

In general, an attribute for a structure will be a structure of the same class (except for *size*, which is always a vector). If an attribute is uniform throughout a structure, it will belong to the same class as the structure but will have only one element.

Any attribute of any variable defined in an active block is available to the programmer, but the memory address of a variable is never available. If a block is not active, access to its variables can be allowed by setting up a chain of variables of type *reference* going up the program tree structure, and procedure calls going down the tree structure. Any block can reach any other block, but only through a strictly regulated transfer of information. This structure more or less forces operating systems written in TPL to be of the type proposed by Dijkstra.[5] In fact, an operating system is being designed in conjunction with the language and machine, called the TPL-Hydra-Operating System Environment (the THOSE multiprogramming system).

Some interesting games can be played with the definition attribute. If a structure is *partially-defined*, it will have a map of identical structure to it containing the

```
    begin new t; x; y; k;
(1)     next←"y←if x = 3 then t else (k←k+1; p[k])"
(2)     p[0]: k←1; x←1;
          while k<size p do
(3)           case x←m[ω(1)/sysval: next] of
(4)             y:
(5)             sysval, y:
(6)             t←leftpart(sysval) ;
          end
    end
```

1. next gets the next test symbol and saves it in y
2. Place p[0] on top of stack, initizlize indices
3. Find relation code of last element of vector on top of stack, save in x.
4. Place symbol on top of stack
5. Concatenate symbol with top of stack
6. Save leftpart of string on top of stack in t, then pop stack.

Figure 3—Modified EULER parsing algorithm using implicit stack

definition status of each element. Consider the situation where we are writing a program to define a structure element by element, and we have several algorithms for defining elements. Each algorithm may be recursive or iterative, and may define some unknown number of elements on each pass. We can call each algorithm in succession, with the exit condition from the algorithm being a comparison between the number of defined elements before a pass and after a pass. This allows the programmer to say, in effect, "Do this until it doesn't do any more good."

Any of the complex structures in TPL may contain any arbitrary collection of other structures as elements. It is possible, for example, to have a *vector* whose elements are *trees* of *processes.*

Operators

The basic operator set of TPL is that contained in the Iverson notation.[9] All the Iverson data operators are included, but the operators on the program stream are written quite differently.

The inclusion of the *process* as a basic data type allows the Iverson data operators to be used for con-

trol operators. For example

1. *if* u *then* b *else* c
   is equivalent to
2. /c, u, b/
   and
3. *case* d *of* (s_1; s_2; s_3; s_4;)
   is equivalent to
4. x [d], x = (s_1, s_2, s_3, s_4).

The more familiar operators were included for two reasons. First, the data structures must sometimes be expressed differently; in (1) b and c could be vectors and u scalar boolean, while in (2) b and c would have to be one-element vectors with vector components to be compatible with u being scalar boolean. Secondly, readability of programs is usually better with the control operators written out, and many programmers will find this notation easier to think in than the more abstract Iverson forms.

One departure from Iverson that must be allowed for is the incompatibility of *integer* and *logical* variables. The compatibility of *integer* and *logical* causes anomalous results in Iverson, PL/I and other languages. As Cheatham noted, the PL/I expression

$$7<6<5 \quad (</(7,6,5) \text{ in Iverson})$$

has the value *true* (1 in Iverson), which is certainly not what one expects. In TPL, $7<6<5$ will produce an error, since $(7<6)$ is *logical* and 5 is *integer*. However, the epxression

$$</(7,6,5)$$

is defined. A reduction with a type-changing operator-operand combination such as a relational and integers is defined differently. For example, if R is any relational and the vector to be reduced is not *logical*,

$$R/(V_1, V_2, \ldots V_n) = (V_1 R V_2) \wedge (V_2 R V_3) \wedge \ldots (V_{n-1} R V_n).$$

This produces the value one expects, i.e., *false* for $</(7,6,5)$.

The Iverson principle of extending scalar operands to vectors and matrices, etc., is followed with respect to the control operators. In the expression

if <expression-1>*then* <expression-2> *else* <expression-3>

<expression-1> may have any structure of booleans as a value. The structure of the values of <expression-2> and <expression-3> must be the same as that of <expression-1>, (if <expression-1> is not a scalar boolean), and the *if* operator is applied to corresponding elements of the structures.

In the expression

case ⟨index-valued-expression⟩ *of* ⟨expression-list⟩

the expression operand may have any structure, and the value of the case is a structure of the same class as the ⟨index-valued-expression⟩, with the indices replaced by the corresponding members of the ⟨expression list⟩. Similarly, any structure class can be executed, so long as all elements of the structure that are actually executed are of type *process*. Likewise, *for* and *while* operators may have compatible structures as their operands.

In all instances where these structures are composed of *processes* to be executed, there is a question of order of evaluation of the elements of the structure. The rules for each structure for evaluation order are as shown below.

Structure	*Order of Evaluation*
vector	left to right
matrix	left to right on rows, top to bottom on columns, no other implied order (i.e., a_{12} and a_{21} have no implied order).
array	low indices to high indices on each dimension, no further implied order.
tree	pre-order
list	pre-order
set	no implied order

In all cases where there is no implied order, there is, of course, a further implication that the *processes* can be executed in parallel.

THE HYDRA COMPUTER

Introduction

ALGOL-like languages have many features in common which are not adequately supported by conventional hardware. Among these features are

1. Run-time binding of variables, including dynamic storage allocation.

2. Procedure calls, particularly recursive calls and procedures which have variable-size arrays as formal parameters.
3. Block structuring, which requires a subroutine call for *begin* and *end* statements on conventional machines.

Since TPL allows the type and structure class of a variable to vary dynamically, it requires even more run-time checking than ALGOL or PL/I. The class of problems for which TPL would be an effective language is therefore quite small on a conventional machine. In order to make TPL (and other ALGOL-like languages) effective, the host machine must be designed to provide hardware support for the features described above, as a minimum.

Run-time binding of variables implies the need for descriptor information associated with each variable. A variable's descriptors should be modified automatically by the system, and the descriptor information should be available to the programmer. For maximum efficiency, descriptors should be kept in a fast register descriptor file while a variable is active. Descriptor manipulating routines can be hard-wired, as in the Burroughs B6500-7500 machines,[2] or microprogrammed, as in the 360 implementation of Euler.[17] Dynamic storage allocation should be done interpretively; producing the code in-line, as is done in some PL/I implementations, slows down the compiler drastically. Storage allocation can be done much more efficiently in microcode than with higher-level machine instructions. It would seem that a system must have at least its basic allocation routines microcoded to be really efficient.

Procedure calls can be handled quite easily with a hardware program stack, as in the Burroughs machines. The stack maintenance routines must be either hard-wired or microcoded; machine instruction subroutines will lose most of the speed gained by putting the stack structure in hardware.

The necessity for subroutines to implement block structuring of programs is partially eliminated by the program stack. The remainder of the block mechanism is handled by coding each variable reference as a two-component address of the form

(level, variable number).

The level number is used to get a pointer into a descriptor file from an address table, as shown in Figure 4. The variable number is then used as a displacement from the pointer.

Several other forms of hardware support for TPL are desirable. Run-time optimization of programs al-

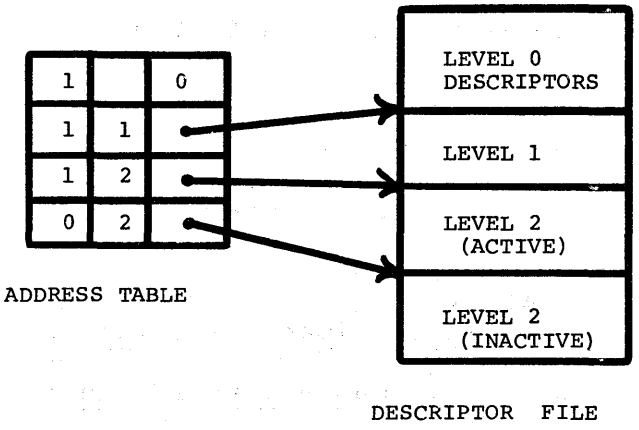


Figure 4—Descriptor addressing

lows programmers to express algorithms in a natural, readable manner and have them executed in an efficient manner. The production of code from compilers is also much easier if run-time optimization is done. A hardware mechanism to allow user-defined structures and operators without machine-language subroutines is also desirable. This would be implemented most flexibly with a writeable control store, but it is feasible to consider hard-wired extensions if an operator or structure is to be used often in a machine's job load. Finally, parallel (or at least partially overlapped) operation of checking and operand location with normal execution is desirable. This parallelism is particularly desirable since TPL contains many complex operators which will require long execution times. If the operations in a program are sufficiently complex, the operand checking and locating features may run with no overhead at all.

Processor organization

The Hydra computer has been designed to provide all of the hardware support features described in the last section. It is felt that the projected relative costs for hardware and software during the next few years justify including all desirable software support features in the machine hardware. Another design feature is the maximum use of microprogramming in the system. This was done for two reasons. First, microprogrammed as opposed to hardwired implementation replaces random logic with more regular memory hardware structures. It appears that the greatest cost savings available from MSI and LSI techniques will result from the use of regular structures, such as memories. The cost and dif-

```
begin new a; b; a←0; b←doit;
  a←a+x;
end
```

TPL program

```
(a, b
 load a load 0 ← ; load b
doit ← ; load a load a load x
```

Hydra program

Figure 5—Main memory program

ficulty of producing random logic will remain high, even in LSI. Second, it is quite difficult to determine in advance which operations will limit the speed of a system; the limiting factors will almost certainly be different depending on job mix. It seems a sensible design procedure to microprogram everything possible, and produce logic chips for the critical functions for each application.

The logical structure of the Hydra consists of four separate functional units operating in parallel. Each instruction is pipelined through the four units. The degree of parallelism required in any actual implementation would be a function of the relative time required for a logic state, a micro-memory access, and a main memory access. The four units of the Hydra are

1. Instruction Acquisition Unit (IAU)—
   Locates a program and its fixed variables and reads them into fast temporary storage.
2. Program Stream Unit (PSU)—
   Locates executable instructions and places them in a queue. It also performs some code optimization.
3. Operand Acquisition Unit (OAU)—
   Fetches the operands for each instruction. It also does type checking and initiates procedure calls.
4. Execution Unit (EU)—
   Performs the actual execution of operations, and stores the results in temporary storage.

We will give a brief description of the construction and operation of each unit, and its relation to the TPL language structure.

Instruction acquisition unit

A program in main memory has the structure shown in Figure 5. Instructions and their operands are kept in a reverse Polish string. Each instruction and pointer is one byte long (the size of a byte may vary in different implementations of the Hydra). The program is called by inserting its descriptor information into a set of hardware stacks which hold program parameters (the initial parameters for a program are kept in its descriptor entry), called the Status Stacks. The Instruction Acquisition Unit (IAU), Figure 6, then uses the Status Stack information to read the program into a fast memory for holding active programs, called the Instruction File (IF). The first item read is a pointer to the static-variable descriptor file entries for the program, which will usually be in main memory. The IAU uses this pointer to load the Static Variable Descriptor File entries for the program.

The Hydra is effectively interpreting a TPL program with microcode. The IAU structure decreases the main memory bandwidth required for the interpretation process.

Program stream unit

The Program Stream Unit (PSU), Figure 7, works on the IF representation of a program. Its output is a stream of executable instructions which are passed on to the Operand Acquisition Unit (OAU) through an Instruction Queue (IQ).

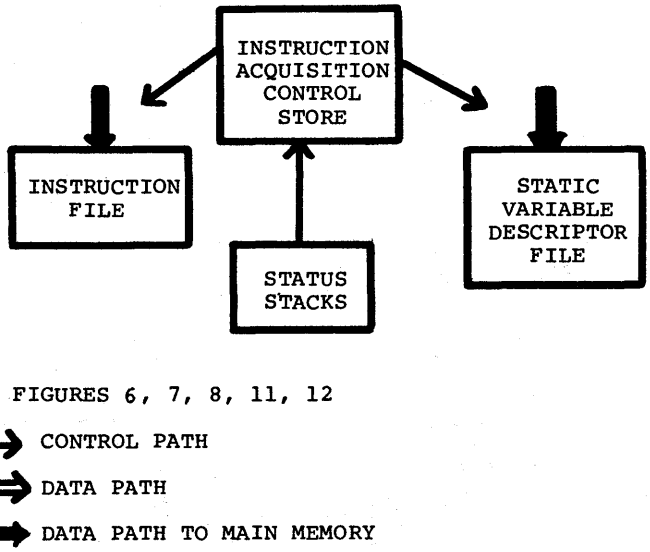


Figure 6—Instruction acquisition unit

The first time a program is processed, the PSU will simply pass all instructions that are not the operands of a conditional operator into the queue. If a sequence of instructions is the operand of a conditional, the sequence is given a Temporary Descriptor File entry, and a pointer to this entry is placed in the queue. This scheme allows code that may or may not be executed to be bypassed until its conditional is resolved, avoiding the necessity of stopping instruction lookahead or guessing the value of the conditional.

The PSU also examines each operator to determine if the value of its result is fixed during this activation of the program. If the value after the operation is executed is variable, the operator is placed in a fast memory which holds the expressions in a program that can have variable values, called the Dynamic Instruction File (DIF); if the value is fixed the operator is discarded after being placed in the IQ. If an iteration or recursion occurs, the reduced program in the DIF will be executed. This process amounts to a run-time compilation of each program. This allows a better code optimization than is possible at compile-time, and also allows programmers to put computations where they occur logically in a program, without worrying about introducing inefficiency at run-time.

The PSU provides hardware support for the run-time optimization described previously, and for the block structuring of TPL.

Operand acquisition unit

The OAU, Figure 8, removes instructions from the head of the IQ, and prepares them for the Execution Unit (EU). It locates operands, does type checking, executes procedure calls and load instructions, and maintains the descriptor files.

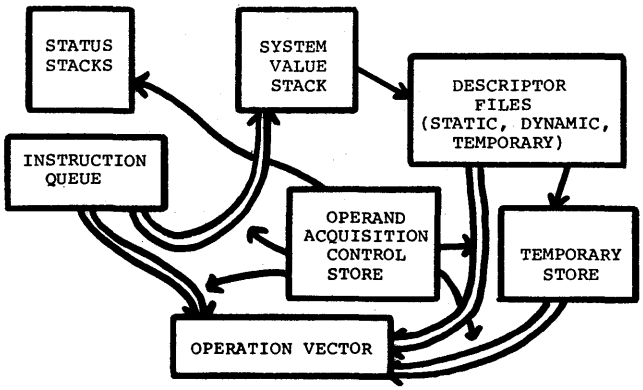


Figure 8—Operand acquisition unit

Each load instruction is followed in the instruction stream by a pointer to the descriptor file entry for the variable to be loaded. A load is processed by placing its pointer on top of the System-Value Stack, and setting a code associated with the pointer to indicate which descriptor file the pointer refers to (this is determined by the kind of load instruction).

An operator causes the OAU to build a vector for the Operation Unit, as shown in Figure 9. The operator is placed directly into the operation vector, and the *type* and *length* of the operands are obtained from the descriptor file entries for the operands. The operand entries are pointed to by the top n entries in the System-Value Stack for an n-ary operator. The *types* of the operands are checked for compatibility with each other and with the operation being performed. The result of the operation will be placed in temporary storage, with a pointer to a Temporary Descriptor File entry on top of the stack. The OAU creates the temporary descriptor entry, pops the operand pointers off the stack, and places the temporary pointer on the stack. It then

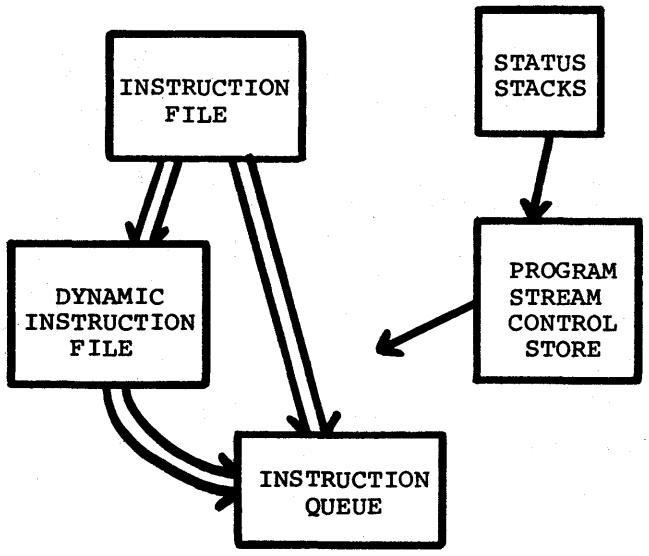


Figure 7—Program stream control unit

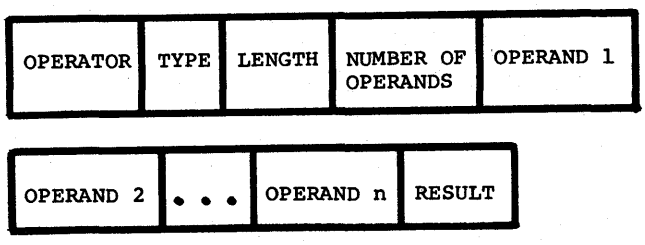


Figure 9—Operation vector

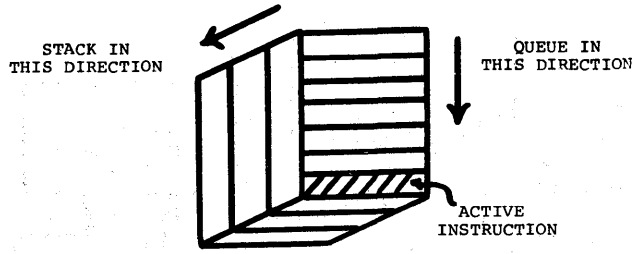


Figure 10—Instruction queue

marks the vector ready for execution (the operation vector will be queued if necessary).

If an execute instruction or an implicit procedure call appears, the OAU places the procedure's descriptor entries into the corresponding status stacks, and pushes the IQ stack. The Instruction Queue is a queue in one direction and a stack in another, as shown in Figure 10. Execution of the called procedure begins as soon as the IQ is pushed. A block end or an *exit* instruction causes the OAU to pop the status stacks pushed by the block (or procedure), and pop the IQ, thus restoring the previous machine conditions.

The OAU implements the self-describing data feature necessary to support dynamic binding of variables in TPL.

Execution unit

The EU, Figure 11, requests operands from main and temporary storage, breaks down complex operators not actually implemented in the Operation Unit, executes each instruction, allocates a block of temporary storage for the result and places the result in temporary storage.

The EU resembles the central processor of a conventional computer, with all functions associated with operand location and instruction sequencing removed.

The logical separation of the four Hydra units allows whatever parallel operation is necessary to be implemented, thus lowering the overhead associated with the procedure-call oriented structure of TPL. This separation also makes extending the language easier; new structures and operators will require new microprograms only in the EU, and new codes for descriptor entries used by the OAU.

SUMMARY

It has been recognized for several years that, since pure hardware cost is a small percentage of the cost of ownership of a computer, improved hardware technology should be used to decrease costs other than hardware. There has been quite a bit of discussion about implementing software functions in hardware, but very few concrete proposals have been made on exactly how this should be done. Outside of specific applications, it is quite difficult to assign portions of a system to hardware or software. The Hydra has been designed with the idea of putting operations and structures normally used in operating systems into the basic machine hardware. This has been done by designing a language with data structures and operators suitable for writing operating systems, and then implementing the language structure as directly as possible in hardware. It should then be relatively easy to build software functions into hardware in specific instances.

The Hydra processor design is currently being studied using a simulator being written in ALGOL-SIMULA for the Univac 1108. The simulator is necessary to answer questions such as:

1. Should we add a control store to handle storage allocation and related activities? If not, how should these routines be divided among the present control stores?
2. What are the optimum sizes for each of the fifteen memory units in terms of both performance and cost effectiveness?
3. Would the activity balance in each unit be improved by relocating some functions (for example, the OAU might handle decomposition of complex operations instead of the ECU)?
4. How does the Hydra's performance compare with
   a. conventional machines
   b. conventional machines with cache memory

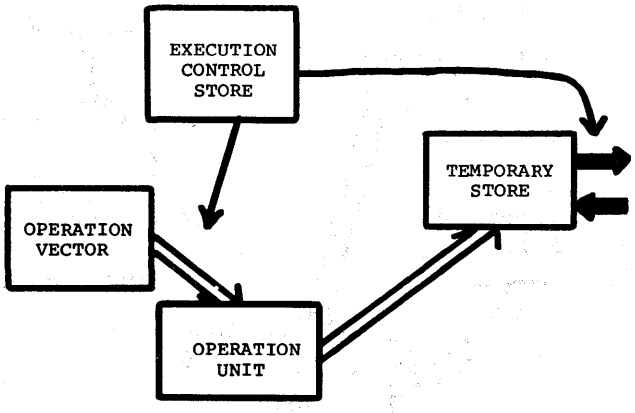


Figure 11—Execution unit

c. Hydra-type machines with cache memory instead of memories with assigned functions?

There are reasonably appealing intuitive answers to all these questions, but the simulation is vital for authoritative answers. Intuition has not proved very accurate when applied to very complex systems.

Work is also proceeding on a TPL compiler for the PDP-10. It is a reasonable assumption that there exist programs that require data structures that are sufficiently complex to allow TPL to be efficient on a conventional machine. The PDP-10 TPL will allow us to test this assumption and discover any hidden difficulties in using TPL.

ACKNOWLEDGMENTS

The author would like to credit the designers of the Burroughs $B \geq 5000$ machines[2,8] for providing several of the ideas used in this system; the designers of the EULER,[18] PL/360,[16] Linear C,[11] and LISP[10] languages and the Iverson notation for many interesting ideas; and to thank Bill McKeeman, Bob McClure, Tom Cheatham, Doug McIlroy, Dick Karpinski and Hi Young for many stimulating discussions which helped clarify some of the murkier portions of the design; and Cliff Hemming and Jerry Duval for their work on the simulation project.

The author would particularly like to thank Session Chairman Dick Watson for many suggestions that improved the readability of the paper, and Barbara McFarland for typing and illustrating the paper.

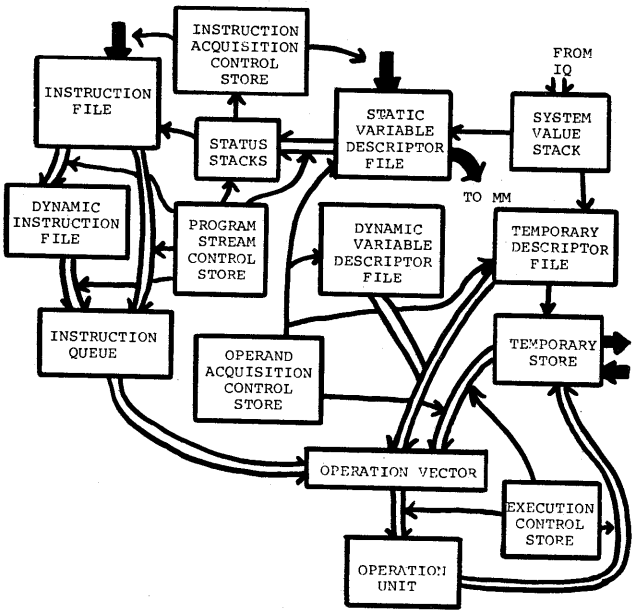


Figure 12—Hydra processor block diagram

REFERENCES

1 T R BASHKOW A SASSON A KRONFELD
System design of a FORTRAN machine
IEEE Transactions on Electronic Computers Vol EC-16 No 4 August 1967

2 *Burroughs B6500 reference manual*
Burroughs Corporation Detroit Michigan September 1969

3 T E CHEATHAM JR A FISCHER P JORRAND
On the basis for ELF—an extensible language facility
Proceedings of the Fall Joint Computer Conference 1968

4 T E CHEATHAM JR
The introduction of definitional facilities into higher level programming languages
Proceedings of the Fall Joint Computer Conference p 623 1966

5 E W DIJKSTRA
The structure of the "THE" multiprogramming system
Comm ACM 11/5 p 341 May 1968

6 B A GALLER A J PERLIS
Criteria for the design of programming languages
Univ of Michigan Advanced Programming Short Course 1966

7 B A GALLER A J PERLIS
A proposal for definitions in ALGOL
Comm ACM 10/4 p 204 April 1967

8 E A HAUCK B A DENT
Burroughs B6500/B7500 stack mechanism
AFIPS Conference Proceedings Vol 32 1968 SJCC

9 K E IVERSON
A programming language
Wiley 1962

10 J McCARTHY
Recursive functions of symbolic expressions and their computation by machine
Comm ACM 3 4 April 1958

11 R M McCLURE
The Linear C language
Unpublished technical memorandum Southern Methodist University Dallas Texas 1968

12 W M McKEEMAN
Language directed computer design
Proceedings of the Fall Joint Computer Conference 1967

13 W M McKEEMAN
An approach to a computer language design
Stanford University Report C5-48 1965

14 G H MEALY
Another look at data
Proceedings of the Fall Joint Computer Conference p 525 1967

15 R F ROSIN
Contemporary concepts of microprogramming and emulation
Computing Surveys 1/4 Dec 1969

16 C STRACHEY
Comments on "DL's"
Comm ACM 9 3 pp 165-166 March 1966

17 H WEBER
A microprogrammed implementation of EULER on IBM System/360 Model 30
Comm ACM 10/9 Sept 1967

18 N WIRTH H WEBER
EULER, a generalization of ALGOL, and its formal definition
Comm ACM 9/1 pp 13-23 Jan 1966 9/2 pp 89-99 Feb 1966

19 N WIRTH
PL/360, a programming language for 360 computers
J ACM 15/1 p 37 Jan 1968

APPENDIX

Syntax for TPL

⟨program⟩::=⟨block⟩
⟨block⟩::=begin ⟨blockbody⟩ end
⟨blockbody⟩::=⟨dec-list⟩⟨exp-list⟩
⟨dec-list⟩::=⟨declaration⟩ | ⟨dec-list⟩ ⟨ex-sep⟩ ⟨declaration⟩
⟨declaration⟩::=new ⟨dec-comp⟩ | ⟨declaration⟩ ⟨ex-sep⟩ ⟨dec-comp⟩
⟨dec-comp⟩::=⟨attribute-list⟩ ⟨expression⟩
⟨attribute-list⟩::=⟨null⟩ | ⟨attribute⟩ | ⟨attribute-list⟩ ⟨attribute⟩
⟨attribute⟩::=real | integer | logical | character | reference | process | scalar | matrix | array | tensor | tree | list | set
⟨exp-list⟩::=⟨expression⟩ | ⟨exp-list⟩ ⟨ex-sep⟩ ⟨expression⟩
⟨expression⟩::=⟨control-expression⟩ | ⟨*n*-expression⟩ | ⟨assn-expr⟩
⟨*n*-expression⟩::=⟨element⟩ | ⟨expression⟩ ⟨element⟩ | ⟨expression⟩ ⟨operator⟩
⟨control-expression⟩::=⟨if-expr⟩ | ⟨case-expr⟩ ⟨for-expr⟩ | ⟨while-expr⟩ ⟨I-O expr⟩ | repeat | exit
⟨assgn-expr⟩::=⟨expression⟩←⟨expression⟩
⟨element⟩::=⟨variable⟩ | ⟨compound-expr⟩ | ⟨block⟩ | (⟨expression⟩)
⟨compound-expr⟩::=begin ⟨exp-list⟩ end
⟨if-expr⟩::=if ⟨expression⟩ then ⟨expression⟩ else ⟨expression⟩
⟨case-expr⟩::=case ⟨expression⟩ of ⟨case-list⟩)
⟨case-list⟩::=(⟨expression⟩ | ⟨case-list⟩ ⟨expression⟩
⟨for-expr⟩::=for ⟨variable⟩←⟨expression⟩ ⟨for-compl⟩ do ⟨expression⟩
⟨for-compl⟩::=⟨null⟩|step⟨expression⟩ until ⟨expression⟩
⟨while-expr⟩::=while ⟨expression⟩ do ⟨expression⟩
⟨null⟩::=
⟨variable⟩ is as usual
⟨operator⟩::=+ | − | × | ÷ | div | rem | ∧ | ∨ | ¬ | = | ≠ | < | > | ≤ | ≥ | ↑ | ↓ | ↑ with subscript | ↓ with subscript | abs | , | /⟨expression⟩: ⟨expression⟩: ⟨expression⟩/ | ⟨operator⟩/[expression] | ⟨operator⟩ [expression] | | /⟨expression⟩:⟨expression⟩: ⟨expression⟩\ | ⟨operator⟩.⟨operator⟩ | [⟨exp-list⟩]
⟨ex-sep⟩::=;/:/.

Analog/hybrid—What it was, what it is, what it may be

by ARTHUR I. RUBIN

Electronic Associates, Inc.
Princeton, New Jersey

THE ZEROTH GENERATION

Introduction

The history of the analog computer goes back to antiquity, where tax maps were first reported being used for assessments and surveying. However, I shall confine this paper to the analog computer as it evolved from World War II to the present time. For those interested in the history of the analog computer, from antiquity to World War II, I refer the reader to an excellent introductory article by J. Roedel, Reference 1. The "Palimpsest" in which Roedel's history of the analog computing art is included is in itself an excellent history of analog computers in the early days dating from World War II to about 1954. From page 4 of the Palimpsest, I would like to show a diagram of computing devices as visualized by George Philbrick for an article in Industrial Laboratories in May, 1952. Of interest to us in this diagram on the analog side, is the separation, at the bottom, between *fast* and *slow analog* which I will discuss shortly. We will also note the presence of *hybrid* at the very top, and this article was written in 1952! Of course, Mr. Philbrick's "hybrid" was reserved for the use of the analog computer first to obtain a ball-park idea of a solution, then followed by a separate digital solution to obtain a more accurate answer to the same problem. I am certain that very few people thought of this as being hybrid computation at the time. However, consider this definition in the light of later work reported by Mark Connelly (Reference 2) in his use of a "skeleton" representation of a problem on the analog in conjunction with a more accurate representation of the problem on the digital.

It is interesting to observe the basic operations as defined by Roedel in Reference 1. This is shown in Figure 2. Note that the early practitioners of the analog art considered differentiation to be a basic linear element for the fast speed computers and did not show potentiometers, since the latter must have been taken for granted. Furthermore, an arbitrary function generator was also not shown. Apparently, that device, which is necessary to make analog computation capable of solving any problem, was developed later or was considered an oddball, along with the comparator (which is really represented by the dry friction element, provided that the output of the dry friction element is

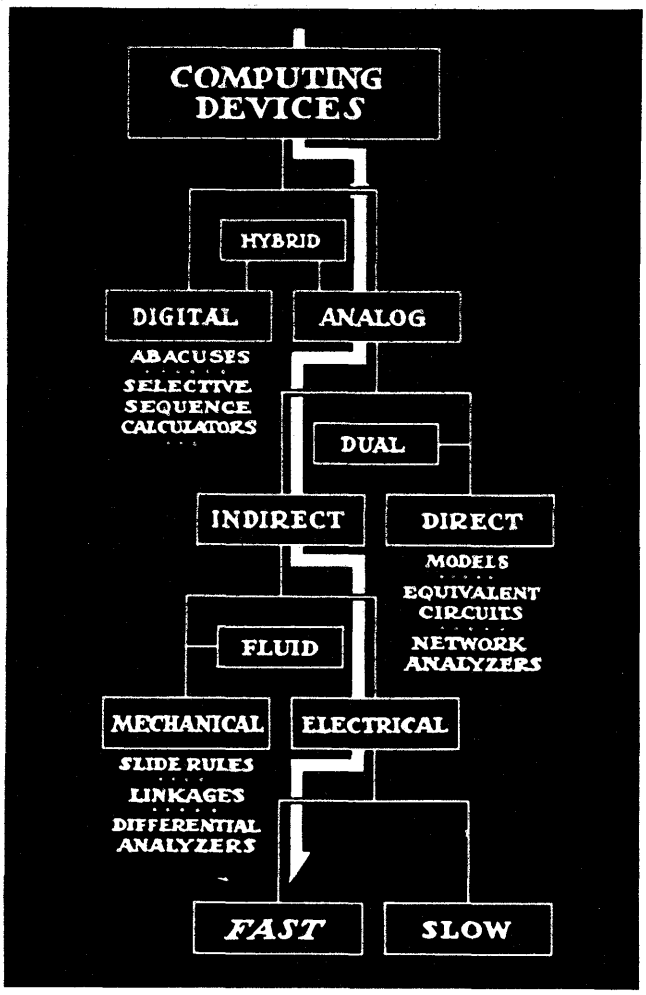


Figure 1—Structure of computing devices as visualized in 1952

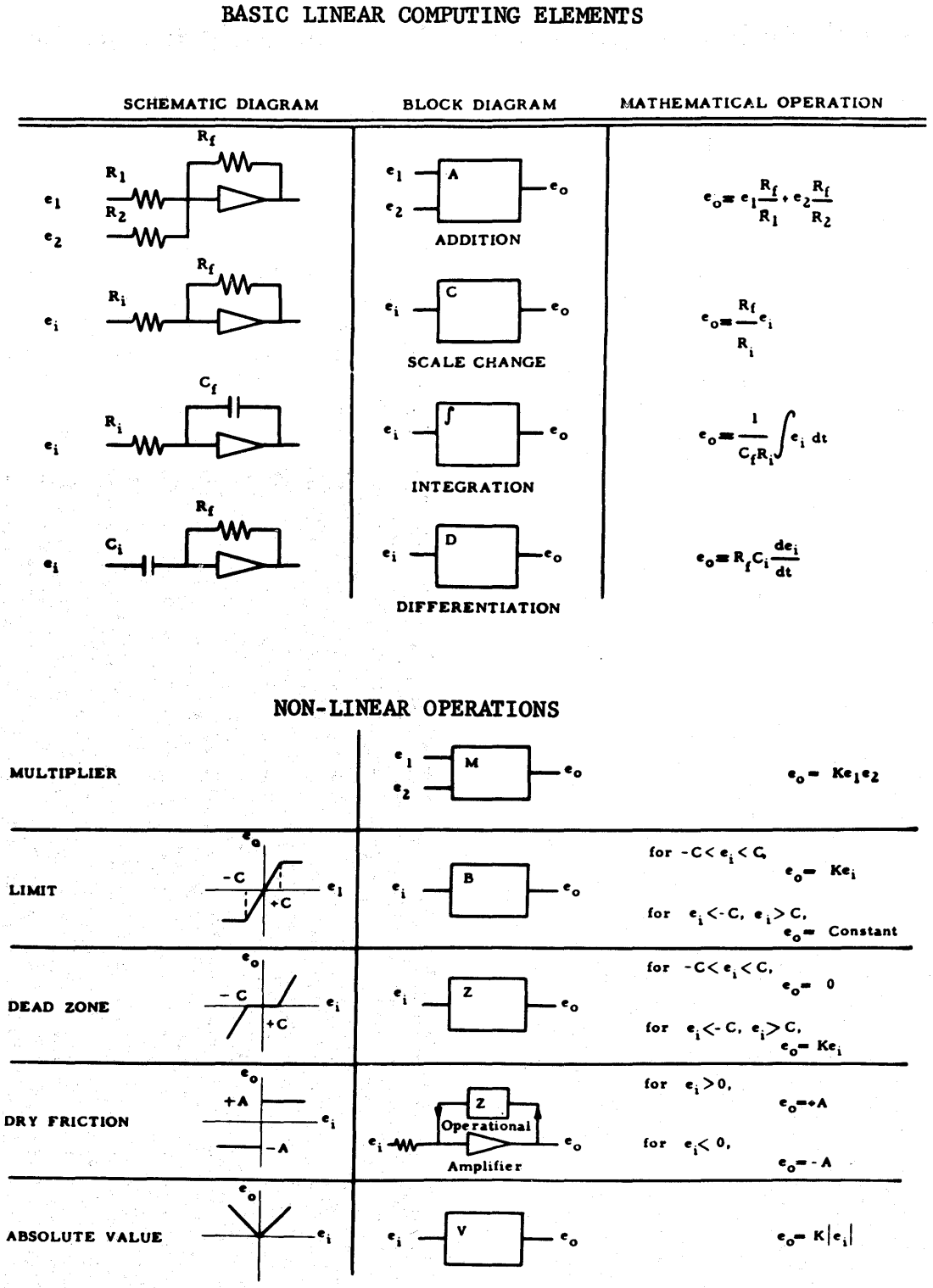


Figure 2—Basic linear and non-linear analog operation and components

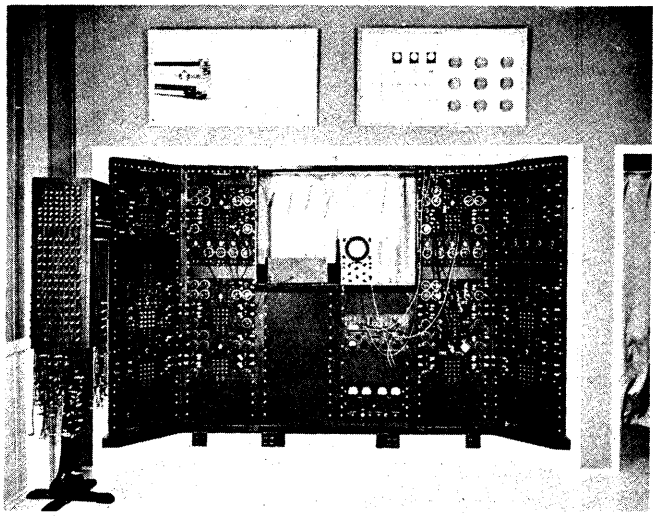


Figure 3—Pullman-Standard Car Manufacturing Company's analog computer installation

used to drive a switch or a gate connected to some other computing element). There was a great deal of emphasis in those days on the solution of linear differential equations, obviously because those required the simplest computing components. Perhaps also, because one could

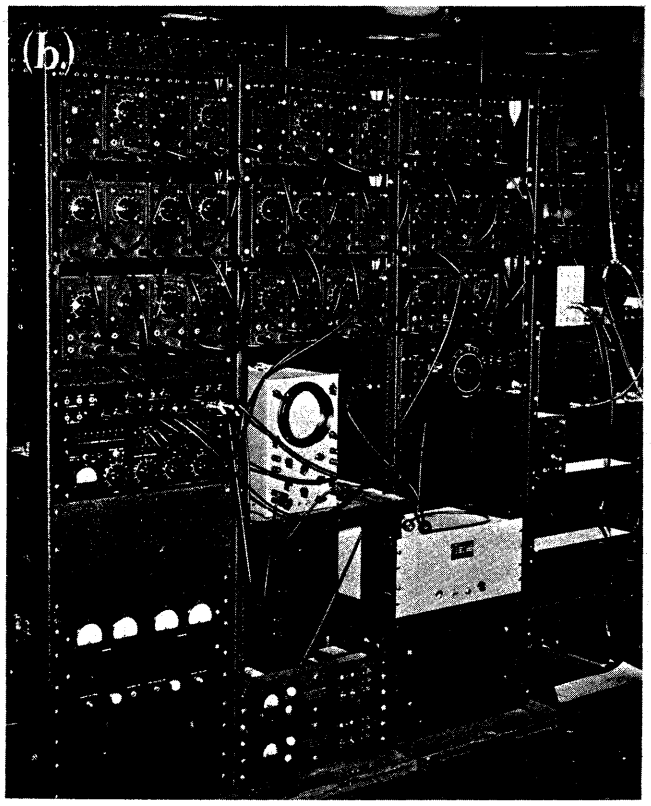


Figure 4—Computing equipment in a typical rack assembly

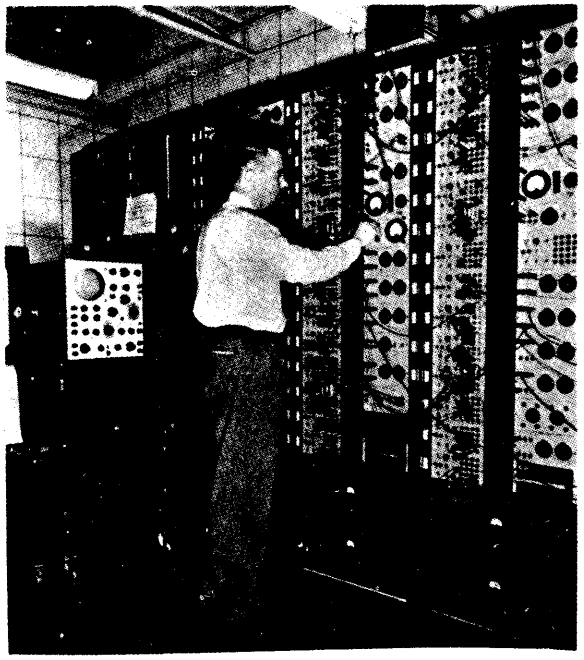


Figure 5—Boeing analog computer, courtesy of Boeing Airplane Co.

obtain check solutions to such equations with pencil and paper, and computers, being relatively new, could not yet be trusted.

Hardware

The major manufacturers during this initial period were the Boeing Company which made the BEAC computer, the Berkeley Scientific Computing Company which made the EASE computer (Berkeley subsequently became part of Beckman Instruments), the Goodyear Aircraft Company which made the GEDA, the IDA computer with which I am not familiar at all, the George A. Philbrick Research Company which made the GAP/R computer, and finally, there was the Reeves Instrument Company which made the REAC computer. Some pictures of these early analog computers are shown in Figures 3 through 8. Figure 3 shows a GAP/R installation while Figure 4 shows a close-up of how those computing components were interconnected. You will note an absence of a patchboard. Can you imagine checking this one out today?

Note the telephone jack panels on the Reeves computer and note also that the Berkeley and the Goodyear computers are the first ones with patch panels. These figures date from about 1952 or 1953. EAI, which was just beginning to build analog computers, does not even show. The typical size of computers in those days ranged from about 20 amplifiers up to 80

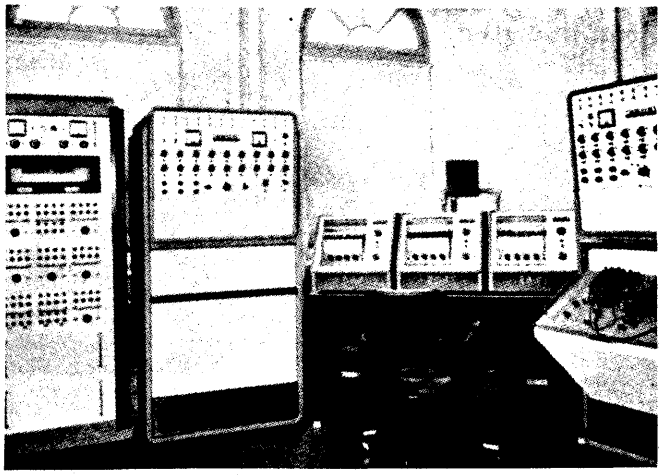


Figure 6—Goodyear GEDA computer installation, courtesy of Goodyear Aircraft Company, Akron, Ohio

amplifiers, which was considered to be fairly large. One manager, in fact, was proud of the fact that he could expand his 80 amplifier installation to 160 without requiring any additional wiring. The accuracy of the components was of the order of one percent (and that applied to resistors and capacitors as well as to the electrical components). Overall solution accuracies on what was then considered medium size non-linear problems was of the order of five percent. One final point of interest is that several of these manufacturers, mainly Boeing, Goodyear, and Reeves were primarily aerospace/defense manufacturers who saw the obvious need for such devices in the design of their own equipment, whether it was airplanes, electronic gear such as radars, or control systems. Philbrick, on the other hand, and possibly also the Berkeley Company, concentrated from the very beginning on the process control applications.

Applications

The 2nd page of the table of contents of the Palimpsest is reproduced here in Figure 9 and shows the wide variety of applications that were actively investigated in the early 1950's. You will note in particular the beginnings of an analytical attack on our environmental problems in the papers on the freezing and thawing of soils as well as flood routing. The analog equipment, especially that which did not have patch panels was generally purchased for a particular problem or problem type. For example, the Pullman Car Company would buy one for solving their "transportation equipment design" problem. An aircraft manufacturer would buy a computer to study the control system of a particular airplane. There was an almost complete lack of user conveniences leading to the ridiculous situation of being able to obtain a complete, single solution to a complex set of differential equations in 5 milliseconds, but having to wait several days, at least, to change to another problem, due to the lack of a patch panel and other amenities, such as a readout system. This type of inaccessibility (to the "next"

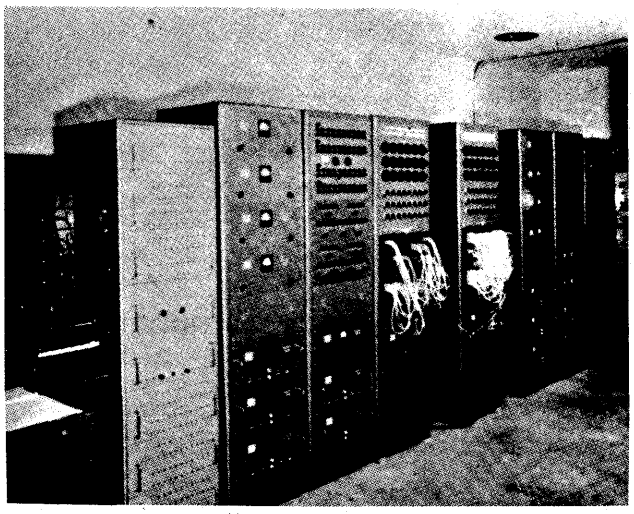


Figure 7—Berkeley EASE computer, courtesy of J. B. RAE, Los Angeles

Figure 8—Reeves computer installation, courtesy Reeves Instrument Company

problem) has been at the root core of the ailment in the analog field and has given the analog computer the reputation of being "inflexible." This ailment is still with us, albeit to a much smaller extent, and a cure is visible on the horizon, as we shall see later. For further information on techniques and methods that were expounded in the early years of analog computation, the reader is referred to References 3, 4 and 5. This by and large represents the first generation analog; however, since I seem to have too many generations, as we shall see later, I will term this the heroic age, or the zeroth generation. This generation coexisted with the heroic age digitals, such as the ENIAC, EDVAC, the ORDVAC, MANIAC, and the UNIVAC.

THE FIRST GENERATION

The next generation, here termed the first, more or less coincided with the arrival of EAI on the scene, with its establishment of the firm need for a patch panel and an integrated set-up and readout console as part and parcel of the analog computer. In other words, human factors entered into the picture, also, this generation saw the arrival of the .01 percent component, such as resistors and capacitors, which allowed linear problems to be solved more accurately than the solutions could be displayed on a strip chart recorder, X–Y plotter, or oscilloscope. The credit for this shift in emphasis on more accuracy and more user conveniences must go to the manufacturers who went against the ideas of some of the then old line users, who kept pointing to the problems that were being solved and observing that much of the input data was unknown perhaps even within a factor of two of the correct value. These old time analysts recognized that there was no need for obtaining very accurate solutions to such problems. However, they overlooked the crutch available to the insecure analyst if he can get a repeatable, accurate answer even though the model is not exact. This analyst then has fewer questions from his management, because when he goes back for reruns, he gets the same old answer to compare with at the same time, the solutions for the new set of parameter values. Thus, he and management both think they understand the problem.

(Aside—I learned this trick early in the game. In order to convince my management and customers as to the validity or correctness of a set-up to a problem, I always went back to a "standard" solution, if a check solution was not available. And if the standard or check didn't repeat, then I would hopefully "tune-up" the equipment to produce a "replica" of the check solution. In some cases, I must confess, I may have "de-tuned" the equipment to produce the so-called "check".)

First printing 1955
Second printing 1958
Third printing 1960
Fourth printing 1965

Figure 9—A portion of the table of contents of the Palimpsest

Conveniences such as a digital volt meter readout of amplifiers and all other components via push-button selectors, servo set pots as well as experiments with quarter-square multipliers and time division multipliers were introduced. The second phase lasted roughly from 1955 to 1960 and saw the rise of EAI from the position of young upstart to that of the major supplier of analog computing equipment. While EAI was rising, the period saw several companies such as Boeing, Goodyear, IDA (or perhaps Mid-Century) drop out of the industry. After these defections from the ranks of the manufacturers, the field of slow speed analogs was split amongst EAI, Berkeley, which by this time had become merged with Beckman, and Reeves Instruments. The high speed analog now had two manufacturers, the old Philbrick Co. and a newcomer to the high speed camp, the GPS Company. This period saw the 31R and the 131R and to lesser extent, the Reeves' C400 gain wide distribution.

1958 NATIONAL SIMULATION CONFERENCE

Clymer, "Operational Analog Simulation of the Viration and Flutter of a Rectangular Multicellular Structure"

Powell, "Distributed Parameter Vibration With Structural Damping and Noise Excitation"

Ladd and Wolf, "A Non-Real-Time Simulation of SAGE Tracking and BOMARC Guidance"

Miller and Enger, "Liquid Transfer and Storage System Simulation by Active Element Computers"

Azgapetian, "Some Aircraft Problems Simulated by Means of Z-forms"

Boxer, "Z-forms, and the Digital Simulation of Dynamics"

Nemerever, "A New Technique in System Performance Evaluation"

Gilbert, "Linear System Approximation by Differential Analyzer Simulation of Orthonormal Approximating Functions"

Brammer, "Solutions of Convolution Integrals by Analog Computers"

Rideout, "Some Applications of a High-Speed Analog Correlator"

Rawdin, "A Time Multiplexing Technique"

Heffron and Bristow, "A Method for Helicopter Rotor Performance Simulation"

Bush and Orlando, "A Perturbation Technique for Analog Computers"

1958 NATIONAL SIMULATION CONFERENCE, continued

Ehlers, "Standard Simulation Circuits"

Gilbert, "The Design of Position and Velocity Servos for Multiplying and Function Generation"

Sinker, "The Card Programmed Diode Function Generator"

Shen, "Multiplier Circuits Utilizing Squaring Property of a Triangular Wave"

Pfeiffer, "A Four Quadrant Multiplier Using Triangular Waves, Diodes, Resistors, and Operational Amplifiers"

Ehlers, "General Purpose DC Analog Computer with Transistor Circuitry"

Pritsker, Buskirk, and Wetherbee, "Simulation to Obtain Systems Measure of Air-Duel Environment"

Billinghurst and Single, "Extending the bandwidth of Precision Analog Systems"

Bekey and Whittier, "Generalized Integration on the Analog Computer"

Bruns, and Wilcher, "Transistorized Relay Amplifier"

Neshyba and Coffman, "Airborne Radar-Beacon Traffic Simulator"

Munson and Rubin, Optimization by Random Search on the Analog Computer"

Schwarm, "Computer Systems for Jet Transport Simulators"

1958 NATIONAL SIMULATION CONFERENCE, continued

Morrison, "APPR-1 Simulator Description"

Meilander and Hellman, "A Technique for Absolute Measurement of Analog Computer Capacitors"

Gerlough, "A Comparison of Techniques for Simulating the Flow of Discrete Objects"

Figure 10—1958 National Simulation Conference

The end of the period saw the introduction of the 231R computer, (See Figure 13) a machine which was to see much service in the '60s.

Applications

The applications of this era (the end of the first generation) perhaps are best described by scanning the list of titles of papers that were presented at the 1958 Fall National Simulation Council Conference (Figure 10). From the list of titles it is clear that the aerospace/defense industry dominated applications, but there were a significant number of papers reporting new mathematical techniques and even applications of digital computers to the field of simulation. New hardware circuits such as the card programmed diode function generator and a quarter square multiplier were first described. Also included were descriptions of a much later transistorized analog, a computer optimization study by analog computers, as well as discrete event simulation by digital computers.

THE SECOND GENERATION

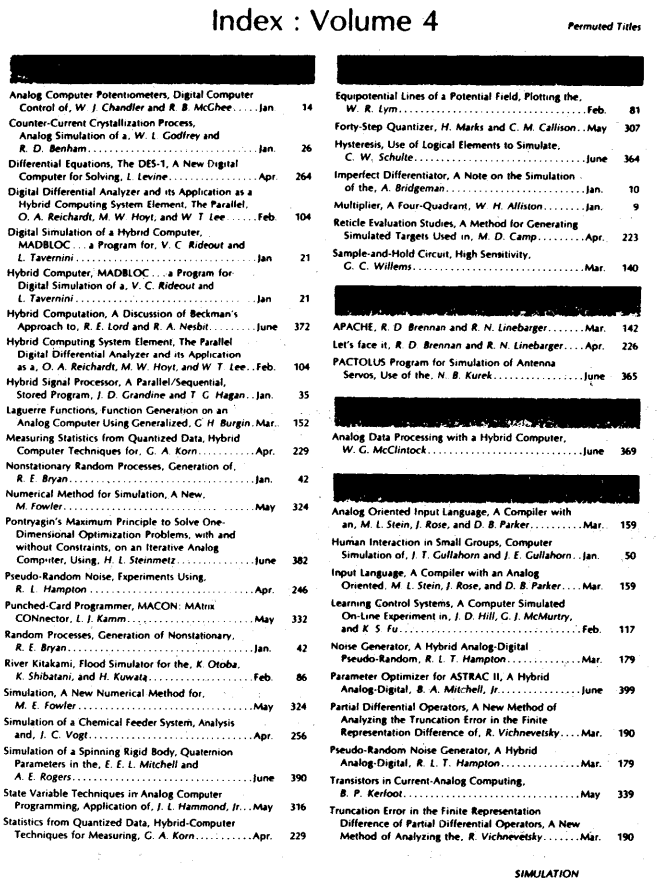
The next generation which I must here call the second, lasted roughly from 1960 to 1965. The size of the analog computer at the upper end was getting physically larger and larger, which by virtue of the vacuum created at the small end led to the design of a small desk-top computer, which was the logical outgrowth of the transitorization of analog components. The first transitorized computers were of the small desk-top type and had a voltage range of plus or minus 10 volts. They

coexisted with their big brothers, the 100 volt vacuum tube computer, during this period. Another hardware innovation saw the combining of heretofore separate *fast* and so called *slow* analog into a single machine. This occurred in both the desk-top machines as well as the large 100 volt machines. This latter turn of events, was brought about by the introduction and the widespread use of quarter square multipliers, replacing the old slow servo and time division multipliers. At the same time, manufacturers introduced the fixed diode type of function generator for such analytical functions as sines and cosines of angles and exponential functions. The above developments, in conjunction with the introduction of solid state highspeed switching made the high speed and the low speed analog in a single computer a practical reality.

Two new companies were formed during this period that proved to be significant factors throughout the '60s. These were Comcor, Inc., which subsequently merged with Astrodata, and Applied Dynamics, Inc., which subsequently merged with Reliance Electric. These companies helped to fill the partial vacuum created by the withdrawal of GEDA from the field during the second generation and the subsequent rise and fall of several other companies. During the same period (1960 to 1965), the Reeves Instruments Company, which had been in the business of manufacturing analog computers from the very beginning, more or less indicated that it was finished with this activity.

The second generation equipment brought to fruition the concept of patchable, parallel, digital logic as an integral part of the analog computer. At the same time, it should be noted that the first conference on the use of combined analog and digital computers in a single problem simultaneously, (the first conference on essentially hybrid applications) was held in 1960, Reference 12, which is why I chose 1960 as a key date for the second generation.

Hybrid computation evolved along several different paths. In one path a "stand-alone digital," was placed next to a "stand-alone analog" with communication between the two allowed via logic lines, control lines, an AD (analog to digital) converter with a multiplexer, and several D/A (digital to analog) converters. Along another path a "logic computer" was developed by Electronic Associates which consisted of a number of parallel, patchable circulating (memory) delay lines, "and" gates, "or" gates, flipflops, four bit registers, shift registers, and one shots (mono-stables) to the point where, if a programmer were clever enough, he could devise and patch together his own special purpose digital computer. The logic computer ultimately got reduced to a reasonably small, manageable complement of logic functions (see Figure 14) on later analog



Index : Volume 4

Permuted Titles

Analog Computer Potentiometers, Digital Computer Control of, W. J. Chandler and R. B. McGhee....Jan. 14
Counter-Current Crystallization Process, Analog Simulation of a, W. L. Godfrey and R. D. Benham....Jan. 26
Differential Equations, The DES-1, A New Digital Computer for Solving, L. Levine....Apr. 264
Digital Differential Analyzer and its Application as a Hybrid Computing System Element, The Parallel, O. A. Reichardt, M. W. Hoyt, and W. T. Lee....Feb. 104
Digital Simulation of a Hybrid Computer, MADBLOC ... a Program for, V. C. Rideout and L. Tavernini....Jan. 21
Hybrid Computer, MADBLOC ... a Program for Digital Simulation of a, V. C. Rideout and L. Tavernini....Jan. 21
Hybrid Computation, A Discussion of Beckman's Approach to, R. E. Lord and R. A. Nesbit....June 372
Hybrid Computing System Element, The Parallel Digital Differential Analyzer and its Application as a, O. A. Reichardt, M. W. Hoyt, and W. T. Lee..Feb. 104
Hybrid Signal Processor, A Parallel/Sequential, Stored Program, J. D. Grandine and T. G. Hagan..Jan. 35
Laguerre Functions, Function Generation on an Analog Computer Using Generalized, G. H. Burgin..Mar. 152
Measuring Statistics from Quantized Data, Hybrid Computer Techniques for, G. A. Korn....Apr. 229
Nonstationary Random Processes, Generation of, R. E. Bryan....Jan. 42
Numerical Method for Simulation, A New, M. Fowler....May 324
Pontryagin's Maximum Principle to Solve One-Dimensional Optimization Problems, with and without Constraints, on an Iterative Analog Computer, Using, H. L. Steinmetz....June 382
Pseudo-Random Noise, Experiments Using, R. L. Hampton....Apr. 246
Punched-Card Programmer, MACON: MAtrix CONnector, L. J. Kamm....May 332
Random Processes, Generation of Nonstationary, R. E. Bryan....Jan. 42
River Kitakami, Flood Simulator for the, K. Otoba, K. Shibatani, and H. Kuwata....Feb. 86
Simulation, A New Numerical Method for, M. E. Fowler....May 324
Simulation of a Chemical Feeder System, Analysis and, J. C. Vogt....Apr. 256
Simulation of a Spinning Rigid Body, Quaternion Parameters in the, E. E. L. Mitchell and A. E. Rogers....June 390
State Variable Techniques in Analog Computer Programming, Application of, J. L. Hammond, Jr....May 316
Statistics from Quantized Data, Hybrid-Computer Techniques for Measuring, G. A. Korn....Apr. 229

Equipotential Lines of a Potential Field, Plotting the, W. R. Lym....Feb. 81
Forty-Step Quantizer, H. Marks and C. M. Callison..May 307
Hysteresis, Use of Logical Elements to Simulate, C. W. Schulte....June 364
Imperfect Differentiator, A Note on the Simulation of the, A. Bridgeman....Jan. 10
Multiplier, A Four-Quadrant, W. H. Allnston....Jan. 9
Reticle Evaluation Studies, A Method for Generating Simulated Targets Used in, M. D. Camp....Apr. 223
Sample-and-Hold Circuit, High Sensitivity, G. C. Willems....Mar. 140

APACHE, R. D. Brennan and R. N. Linebarger....Mar. 142
Let's face it, R. D. Brennan and R. N. Linebarger....Apr. 226
PACTOLUS Program for Simulation of Antenna Servos, Use of the, N. B. Kurek....June 365

Analog Data Processing with a Hybrid Computer, W. G. McClintock....June 369

Analog Oriented Input Language, A Compiler with an, M. L. Stein, J. Rose, and D. B. Parker....Mar. 159
Human Interaction in Small Groups, Computer Simulation of, J. T. Gullahorn and J. E. Gullahorn..Jan. 50
Input Language, A Compiler with an Analog Oriented, M. L. Stein, J. Rose, and D. B. Parker....Mar. 159
Learning Control Systems, A Computer Simulated On-Line Experiment in, J. D. Hill, G. J. McMurtry, and K. S. Fu....Feb. 117
Noise Generator, A Hybrid Analog-Digital Pseudo-Random, R. L. T. Hampton....Mar. 179
Parameter Optimizer for ASTRAC II, A Hybrid Analog-Digital, B. A. Mitchell, Jr....June 399
Partial Differential Operators, A New Method of Analyzing the Truncation Error in the Finite Representation Difference of, R. Vichnevetsky....Mar. 190
Pseudo-Random Noise Generator, A Hybrid Analog-Digital, R. L. T. Hampton....Mar. 179
Transistors in Current-Analog Computing, B. P. Kerfoot....May 339
Truncation Error in the Finite Representation Difference of Partial Differential Operators, A New Method of Analyzing the, R. Vichnevetsky....Mar. 190

SIMULATION

Figure 11—Index to Vol. 4 SIMULATION, June, 1965

computers and became fully integrated with the analog computer instead of being a separate device.

The largest analog computing consoles had upwards of 200 amplifiers in them. One example is the Applied Dynamics 256 (Figure 15) which had 256 amplifiers. There was a movement towards more "committed" amplifiers such as the class 0 type quarter square multiplier with many of the largest, most sophisticated users trending towards the class 0 resolver as well. This meant the computer was easier to use, but it became more expensive.

The applications of analog during the period 1960–1965 are more difficult to characterize since by this time the well known industry-wide Spring and Fall Joint Computer Conferences, which are sponsored by AFIPS, had replaced the old National Simulation Conferences. To help plug the information on applications gap, a significant event occurred during the second generation, which was the launching and the publication of the new journal SIMULATION by Simulation Councils, Inc., under the editorship of John McLeod. The index of articles published in Volume 4 (June, 1965) is shown in Figure 11, as an example of the type of applications

that were being done on these bigger, better and more powerful systems. It may be remarked in passing that even during this second generation period, indeed throughout the history of the analog, the analog has been used very much as it was originally used *when there was no patchboard on the analog console.* This method of use consists of committing the analog to a single problem, of very high priority, and tieing it up full time doing the same job over and over and over again, as exemplified by the typical hardware or man-in-the-loop simulator. Very often when the project that required the simulator was completed or nowadays we would say cancelled, there was no further use or need for the analog computer, since no one else had been able to get at the machine during the "fat" days. Those analysts who had short duration, small problems, which can be considered to be ideal candidates for the analog computer, especially during the development or the "model" stage of the problem, were forced to go against their own wishes to the, by then, widely available large, fast, digital computer of the 7090 class. These small, repetitive, studies went to digital *not* because the machine was fast, *not* because the digital was cheaper, *not* because it was better, *not* because it was more accurate, *but* simply because it was available!

Year	Typical Computer	
1951	C100 (Reeves)	20 amplifier computer; servo multipliers introduction of removable patchboard.
1954	31R (EAI)	20 amplifier computer, expandable to 60; more accurate servo multipliers; integrated slaving system; .01% capacitors; .01% resistors, both temperature controlled.
1956	131R (EAI) C400 (Reeves)	Integrated readout; human engineered for faster, easier programmer use; electronic time division multipliers; mechanical digital voltmeter; tube diode function generators.
1959	231R (EAI)	100 amplifier computer; modular concept patchboard; significant improvements in amplifier bandwidths providing faster response and switching times; compressed time capability (some jobs can be run as fast as 10:1 real time instead of all at real time); faster potentiometer readout; electronic digital voltmeter; solid state diodes in function generator; repetitive operation capability.
1962	Improved 231R (EAI)	More accurate 1/4 square multiplier; electronic sinusoidal generator; point storage via transistor circuit; card-set function generators; Mark 200 recorder now compatible with accuracy and repeatability of computer. More useful bandwidth.
1964	231RV (EAI Beckman	Electronic mode control of integrators, time-scale selection (6 decades) via push buttons, more accurate multipliers and sinusoid generators. Digital logic control capability - for the first time analog has a full 10KC bandwidth in all components, Variable breakpoint and polarity, card-programmed function generators. This allows instant set-up of DFGs (takes only one hour to turn around a problem). (Mostly pot settings time.)
1966	Ci-5000 ADI-4 EAI 8800 EAI 680	Fully transistorized, more accurate, more reliable analog computer - all gates (reset, hold, operate, are electronic) bandwidth up to and beyond 100 KC, reliability estimated as 60,000 hours MTBF for amplifiers vs. 5,000 hours measured on 231R-V. Computers can have 300-400 amplifiers in one console. Analog directly controllable by small digital computer. Easy mating with digital for hybrid computation. Self-contained patchboard - digital logic (much, much larger than in 231R-V). Card-set DFGs programmable from a standard IBM card.
1968 to Present	ADI - Various EAI - Various	Digital pots for microsecond (electronic gate) setup - or millisecond (reed relay setup), large scale use of MDACs in hybrid interface, software developed for automatic setup and checkout of hybrid analog computers. Direct digital/analog function generator (more accurate than card set diode function generator) completely controllable from digital computer.

THE THIRD GENERATION

The third generation has shown itself to be in existence from roughly 1965 to the present time, 1970. The major hardware characteristic of this generation is the complete transitorization of the analog computer, for both the large scale 100 volt machine and the small scale 10 volt machine. A new scale machine evolved in between these two extremes, called the medium scale. A major hardware feature is the integral design of digital logic as part and parcel of most analog consoles,

Figure 12—History of analog computer evolution since 1951

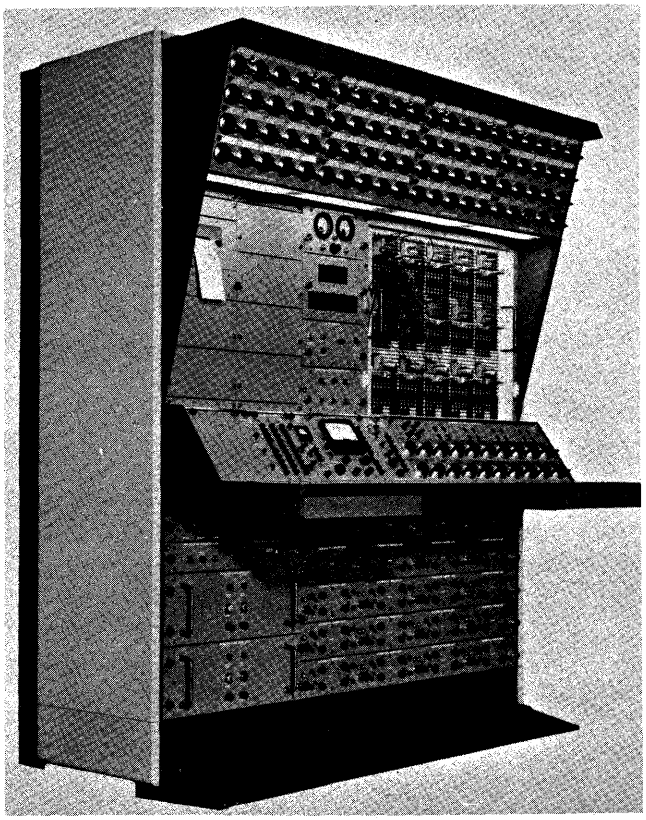


Figure 13—231*R* computer, courtesy Electronic Associates, Inc.

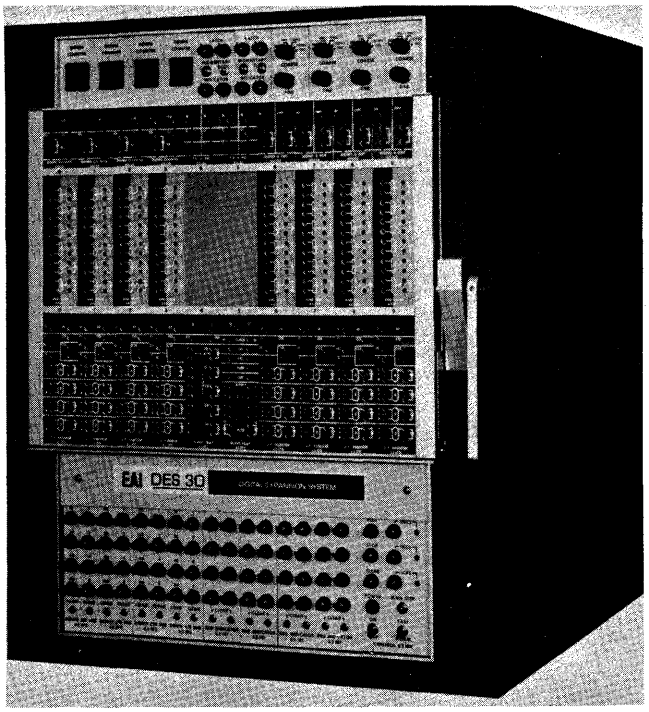


Figure 14—Digital expansion system by EAI (allows parallel patchable digital logic expansion to 10 volt systems, in a self-contained desk top frame

small and large, which has certainly made pure analog computation, if we include this digital logic, more powerful than it has ever been. Another hardware feature is the complete flexibility of the multi-time scale integration capability of the analog, wherein one can have a choice of fast, slow or in-between speeds of solution as well as the flexibility of using any integrating capacitor as an integrator gain. The most versatile machines have a choice of 6 capacitors, giving the programmer a five-decade range of integrator gains or time scales. Examples of this class of computer are the Applied Dynamics AD/4 (Figure 16), the Electronic Associates, Inc. 8800 (Figure 17) and the Comcor Ci-5000 (Figure 18). Note the two patchboards in each, one for digital logic, and one for analog components.

This period also saw a more intimate tie-in of the analog computer with a digital computer due to the development of such true hybrid devices as the MDAC (multiplying D/A) and the "digital attenuator" or "digital potentiometer." So widely accepted has the hybrid aspect of analog computation become that it appears that close to half of the larger consoles that are being sold at the present time are going into hybrid systems. This in turn has led to the need, and the development of software specifically designed to aid the hybrid programmer and operator. The large systems have grown larger and larger and now are truly prodigious, consisting of 300, 400, even 500 amplifiers in a

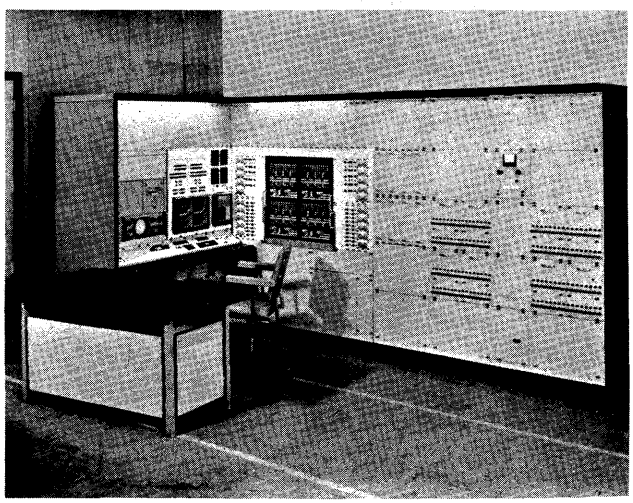


Figure 15—Applied Dynamics large scale 256 amplifier computer

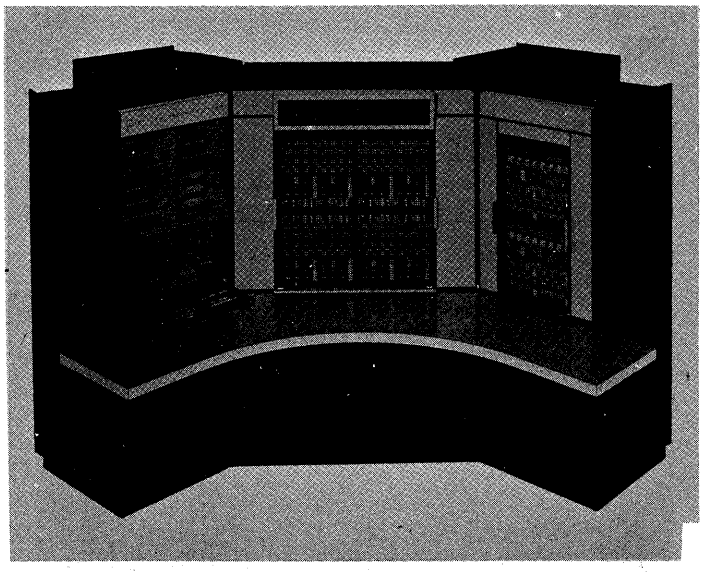


Figure 16—Applied Dynamics AD/4 analog computer

single console. At the low end of the scale, the 10 volt desk-top computers have grown larger and larger until they are no longer desk-top and now are fully grown consoles consisting of several hundred amplifiers, as exemplified by the EAI 680 computer shown in Figure 19.

The solid state revolution, which only overtook analog in the third generation has led to the concept of the class 0 type component or "blackbox" use of the analog components to help minimize patching and to make it easier for the more casual user of the machine to program, patch, and obtain solutions by himself. Another reason for this trend is that the solid state amplifiers are obviously less costly and more reliable than their vacuum tube predecessors. Analog speeds of solution which could be too fast to be absorbed by humans, or recorded by devices, even back in the early 50's, are even faster. Present day bandwidth ranges from a minimum of 100 KHz to over 1 MHz. Some of the other important equipment improvements are quarter square multiplier accuracy of close to 0.01 percent and arbitrary function generation performed by a true hybrid device, the digitally controlled function generator (DCFG), which eliminates spurious drifts, non-repeatability, and difficulty in setup of the old diode function generator. These, together with the new digital potentiometer, a good hybrid interface with good software, and a well integrated system design, make it theoretically possible to setup and checkout an analog computer in a few *seconds*.

Some persons have been known to state the opinion that an analog computer of today is not much different than one of 10 years ago. A reading of this paper should dispel such a notion. To make clear the advances that have been made in the analog field, from post World War II to the present time, I have summarized in Figure 12 the major hardware improvements by year of general availability showing the typical computers incorporating the named improvements. It is obvious that these improvements have come at more frequent intervals than analog computer generations as I have defined them, and shows that major improvements have come along in the analog field at an average spacing of about 2½ years. This interval of time is, interestingly enough, approximately equal to the half-life of a "generation" of analog computers. This fact might lead to the conclusion that one generation of computers cannot survive (or absorb) two sets of major hardware improvements, but that the manufacturers have been reasonably successful in extending the life of a generation of their computers through at least one significant hardware evolution. Perhaps it is the ability to extend the life of a "generation" of analog computers, because of the nature of the organization of analog computers (parallel building blocks) which has led to the inaccurate observation that "analog computers of today are not much different than they were 5 or 10 years ago."

ANALOG/HYBRID TODAY

We have now come to the point in analog/hybrid developments where not only do we have more raw computing speed than it is possible to take full advantage of, for solutions, but we also have more speed in terms of setup and checkout than we have customers

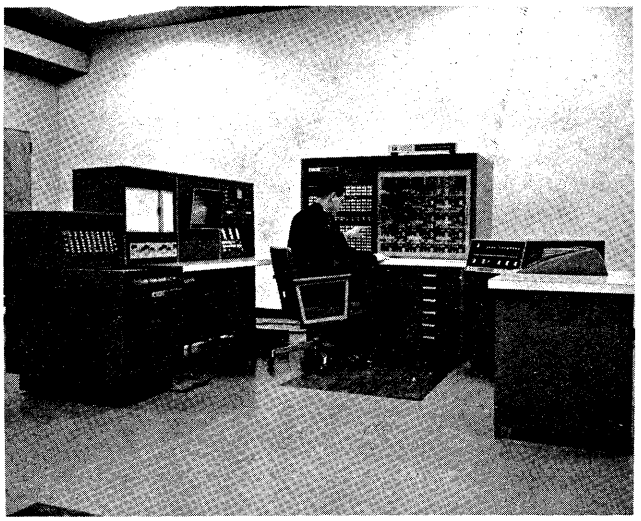


Figure 17—680 10*V* computer with display wing

who understand this type of computation. Or to put it another way, we've reached the stage in evolution where we can get a customer on, get his answers for him, and get him off, far faster than is justifiable based on the fact that we have a highly serial, slow input, mainly the input from a single man, to a very fast parallel console. We have almost reached the stage, as a matter of fact, where the slow recorders on the outputs from the analog are one of the limiting output factors. We've reached the point where we can make many, many solutions in a very short time. In other words, we are production oriented in terms of solution speed. At the same time, we have retained all of our man-machine interactive capabilities which everyone says is desirable in the engineering use of computers, but which obviously work *against* production. In fact, production capabilities are so great that I have estimated that for every hour of production running on our modern hybrid systems, the amount of post run data reduction of the results by a large fast, stand alone digital computer operating in a batch mode would be at least two and possibly as high as five hours depending on how much analysis is desired, or more realistically, how much the project can afford.

The application of hybrid equipment is still heavily oriented toward the aerospace-defense industry where most of the large systems are installed. The chemical process industries have maintained some interest in these systems over the years, but not at an increasing rate. The education field has interest in the small and medium size systems. Nuclear and power technology have shown signs of increasing awareness of the capability of hybrid systems for their highly complex

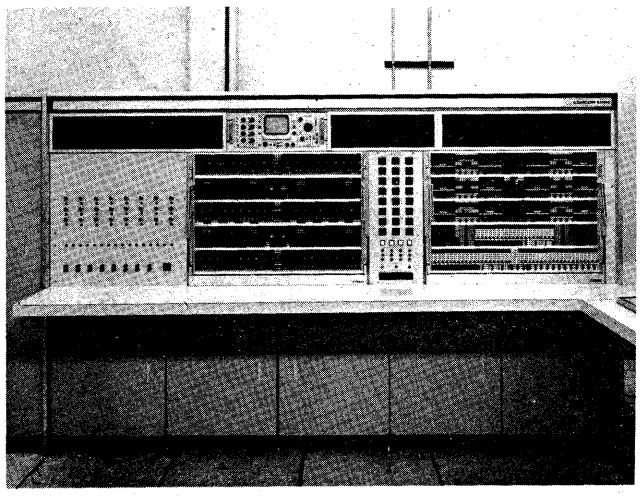


Figure 18—Comcor Ci-5000 analog computer

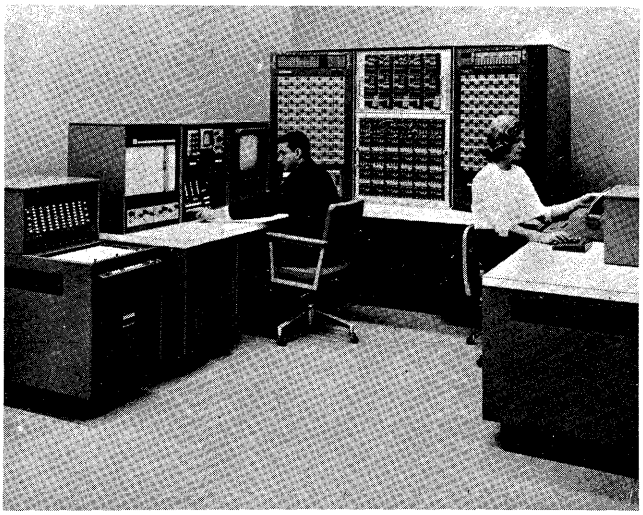


Figure 19—8800 100V transitorized computer with display wing

design, control, and training studies. Other popular applications are as an on-line testing device, such as measuring the amount of pollutants in an automobile engine exhaust (Reference 6); measuring the roundness of tires (Reference 7) in acting as an on-line predictor or adaptor-controller for a wide variety of processes (Reference 8), and for helping to control the quality of steel (Reference 9).

So what is the hybrid/analog system of today? It is a highly efficient fast production device when the user or man is not allowed to intervene and interfere with its operation. This is in direct contradiction to its other main feature, that is, its ease of man-machine communication which almost cries out for man's intervention. I would say that the analog/hybrid computer exhibits schizophrenic characteristics which may explain why not too many people understand it. It is almost impossible for a device to be responsive to man's intervention and at the same time to be highly productive. At least not the way the hybrid systems are configured today. It is this paradox that limits the expansion of the analog/hybrid field.

The analog hardware today is far more reliable than its early beginnings. The MTBF for a transistorized amplifier is somewhere between 30,000 hours and 60,000 hours. The high quality, chopperless amplifier, a recent development, brings us back, almost full circle to the point where we were with the very first analog amplifiers, that is, a chopperless, unstabilized amplifier with a virtually instantaneous overload recovery. This is a feature that all users will appreciate. However, it has taken 25 to 30 years, an electronic revolution, and 3 or 4 generations of computers to eliminate the drift and

unreliability of the first unstabilized amplifiers, while retaining the desirable features of simplicity and quick overload recovery.

The future

The analog/hybrid computer could become more widespread in its use and acceptance by industry if it can eliminate its schizophrenia and solve its paradox. Hardware and software ideas have been mentioned for doing just this, such as an automatically patched analog computer (Reference 10), coupled with a high level language for programming the machine in user oriented language, such as APSE and APACHE, all of which is made highly accessible and productive with many interactive graphics terminals (Reference 11) controlled and hybridized by one of those next generation, fast, cheap, can-do-anything digital computers that I keep hearing about.

At the very least, it will continue to be used in those on-line experiments, those teaching-learning situations, those high frequency problems, that saturate large digitals, and by those specialists who are addicted to analog, as it has been used in the past.

REFERENCES

1 J ROEDEL
An introduction to analog computers
From A Palimpsest on the Electronic Analog Art ed by H M Paynter first printed by George A Philbrick Researchers Inc 1955 pp 27-47

2 M CONNELLY O FEDOROFF
A demonstration hybrid computer for real-time flight simulation
February 1965 Report ESL-FR-218 Contract AF 33(616)-8363 M I T Cambridge 39 Mass

3 Project Cyclone-Symposium I, Reeves Instrument Corp under contract with the Special Devices Center of the Department of the Navy March 1951

4 Project Cyclone—Symposium II Reeves Instrument Corp (Part II) under contract with the Special Devices Center of the Department of the Navy April 1952

5 Project Typhoon—Symposium III on Simulation and Computing Techniques Bureau of Aeronautics and US Naval Air Development Center October 1953

6 J P LANDAUER
Hybrid computer real-time data processing for engine testing
Electronic Associates Inc Market Development Report 17-70

7 J T MAGUIRE A J SCHNABOLK
Sorting out the tires
ELECTRONICS March 18 1968

8 P ADAMS A SCHOOLEY
Adaptive-predictive control of a batch reactor
EAI applications Reference Library #6.2.2/a

9 H J HENN J D SCHIMKETS T G JOHN
Development and operation of a refining control system for stainless steels
ASME Electric Furnace Conference Detroit Mich December 1969

10 G HANNAUER
Automatic patching for analog and hybrid computers
SIMULATION Vol 12 #5 May 1969 pp 219-232

11 R M HOWE R A MORAN
Time sharing of hybrid computers using electronic patching
Proc of 1970 Summer Computer Sim Conf Vol 1 pp 124-133

12 Proc of Combined Analog/Digital Computer Systems Symposium sponsored by SCi and General Electric Company December 1960 Available from SARE

The hologram tablet—A new graphic input device

by MITSUHITO SAKAGUCHI and NOBUO NISHIDA

Nippon Electric Company, Ltd.
Kawasaki, Japan

INTRODUCTION

Graphic data tablets are input devices which digitize coordinate positions of topological patterns.

The graphic data tablet is a powerful tool with respect to applications of man-machine communications:

1. Terminal for Computer Aided Instruction. The graphic data tablet plays a role as a selector of an item from an array of items which are displayed on a screen or printed on a paper, or as a highly versatile programmable keyboard.
2. Terminal for Data Communications requesting information guidance or information retrieval by hand-written characters.
3. Input terminal of pictorial informations for a graphic manipulation such as Computer Aided Design of integrated circuits.

For the sake of applications, the graphic data tablet should satisfy the following conditions.

1. Simple and easy to handle.
2. Compact size and light weight.
3. Easy to interface or connect to the associated computers.
4. Low price.

Conventional devices such as the RAND tablet[1] and the sylvania data tablet[2] do not seem to satisfy conditions (2) and (4), because they employ different hardwares for quantizing the coordinates and encoding the positions.

Hologram tablet, shown in Figure 1, provides a new graphic input device in which the quantizing function and the encoding function are carried out in a single hologram plate which contains a two-dimensional array of small holograms recording the encoded position signals.

Immediate generation of encoded binary signal without complicated electronic means, which is achieved by use of holography, is the most important merit of the hologram tablet. This leads to the realization of high speed, high resolution, compact size and low price.

In the prototype device, the tracing speed 10^4 positions per second and the resolution 2 lines/mm were obtained.

Principle of the hologram tablet

Holography[3] is a two-step imaging process. One step is the recording of wavefronts of coherent light beam spatially modulated by an object in the form of interference pattern with a reference light beam, and the other is the reconstructing of the object image by illuminating the interference pattern (hologram) with a coherent beam. Hologram tablet was achieved by introducing a new function to hologram

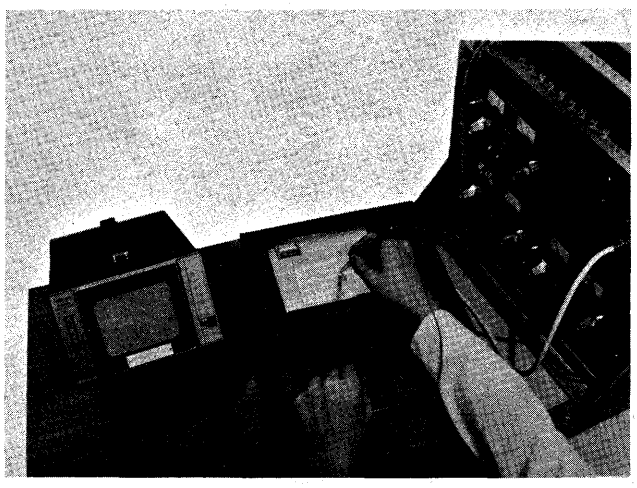


Figure 1—Prototype device of the hologram tablet in operation

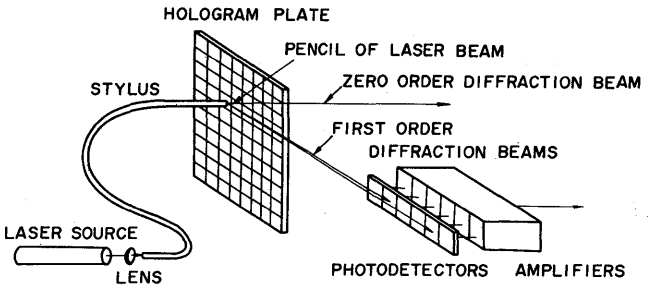


Figure 2—Schematic diagram of the hologram tablet

memory[4,5] in which small holograms of memory plane contain arbitrary informations. The location of a small hologram in the hologram memory is an area of "temporary residence" recording information.

In a hologram tablet the location of a small hologram expresses the decoded signal of the position information recorded in the small hologram. The location of a small hologram and the information in a small hologram are connected each other by logical functions. Thus the area of a small hologram in the hologram tablet is considered to carry out the quantization or the smoothing of input patterns drawn by stylus.

Figure 2 shows the principle of hologram tablet. The hologram tablet consists of a hologram plate containing small holograms recording the binary-coded position informations, a stylus emitting a pencil of laser beam and photodetectors to receive the laser beams diffracted from each of the small holograms. When the stylus moves on the hologram plate, the pencil of laser beam illuminates the small hologram just below the stylus and generates the first order diffraction beams expressing the binary-coded signals of coordinates X and Y indicated by the stylus, which are detected by photodetectors. The outputs from the photodetectors are fed to computer memories.

Since the zero order diffraction beam traces the movements of the stylus, it is possible to take a hard copy of the trace of the input pattern on the photosensitive paper behind the hologram plate, or access simultaneously several hologram plates aligned in cascade.

If there are $2^n \times 2^m$ small holograms, then the position-code consists of $(n+m)$ bits. Thus $(n+m)$ photodetectors should be prepared.

Size of hologram plate and that of small hologram are determined by the dimensions of input patterns and the quantization size, respectively. The diameter of the pencil of laser beam should be small enough to smooth the meaningless movement of the stylus such as tremors in the small hologram. Each photodetector should have an area wide enough always to receive the reconstructed image beam whose position might undergo random fluctuation due to that of the incident beam direction and aberrations of hologram plate. In ideal conditions the photodetector diameter D_p is given by

$$D_p \geqq 4S_H S_D F\lambda/\pi D_H,$$

where

$D_H = 2S_H W_H$,

D_H = small hologram diameter,

F = distance from the hologram plate to the reconstructed image position,

λ = wavelength,

S_H = smoothing factor of the small hologram,

S_D = safety factor for the photodetector,

W_H = beam waist of the pencil of laser beam.

D_p should be greater than 3.2 mmϕ for the values utilized for the prototype device described below, and

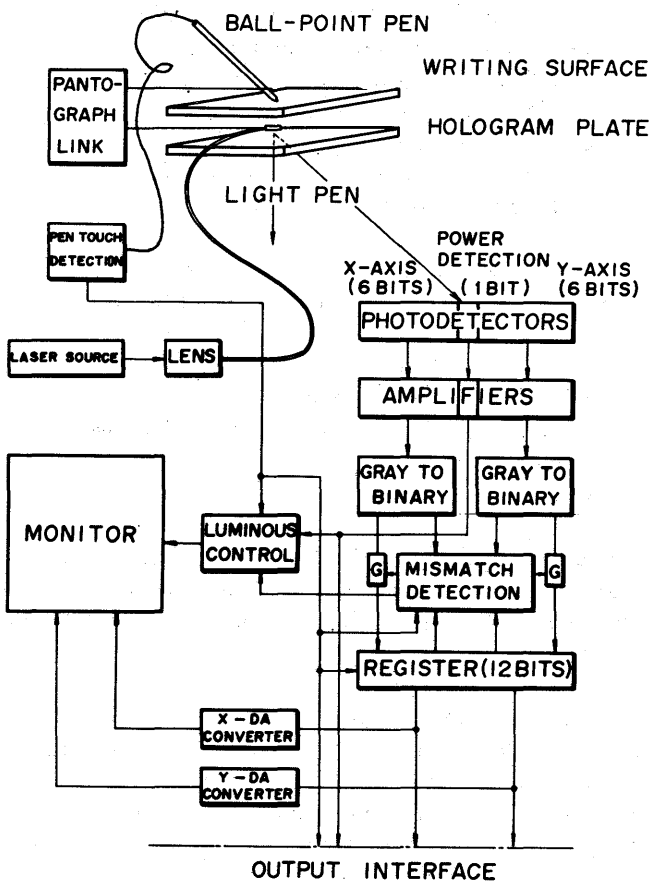


Figure 3—System block diagram of the prototype device, "G" indicates a gate circuit

D_H = 0.5 mm, F = 200 mm, λ = 0.6328 μm, S_H = 2, and S_D = 5.

Data input rate is restricted only by photodetector and amplifier response.

System description

A prototype hologram tablet was constructed in order to confirm the expected performance, particularly the accuracy of the code-generation and easy handling.

The accuracy of the code-generation is determined by

1. Overlap of position-coding masks at the time of constructing the hologram plate.
2. Uniformity of the reconstructed image intensity over all of the small holograms in hologram plate.
3. Fluctuation of the reconstructed image beams on photodetector surfaces.

Handling of hologram tablet is affected by the smoothness of the movement of stylus link and the flexibility of the optical guide guiding the pencil of laser beam.

Figure 3 shows the system block diagram of a hologram tablet. The pencil of laser beam emitted from the light pen should be held normal to the hologram plate in order to let the reconstructed image beams fall correctly on the appropriate photodetectors. On the other hand the stylus combined with a ball-point pen to take hard copies of input patterns on the writing surface is required to be movable freely and easily. The ball-point pen and the light pen were connected tandem by a pantograph link in order to carry out the same movement with respect to the coordinates X and Y. The touch of the ball-point pen onto the writing surface was detected by a microswitch installed at the top of the pen point, and the signal from pen-touch detection circuit was used for the on-off control of the brightness of the monitor CRT. The laser beam with a spot size of 0.25 mmϕ was guided from a compact He-Ne gas laser (size 300 mm x 50 mmϕ, output 2 mW) to the light pen containing an optical guide.

The bleached hologram plate contained 64 x 64 small holograms (each small hologram 0.5 x 0.5 mm^2), in each two pairs of the Gray code of six bits and one sign bit were recorded. Each of the Gray codes of six bits indicated the coordinates X and Y of the small hologram in hologram plate. Thirteen solar cells (size 5 x 5 mm^2) were placed at each of the reconstructed image positions (intervals 6.2 mm). Twelve

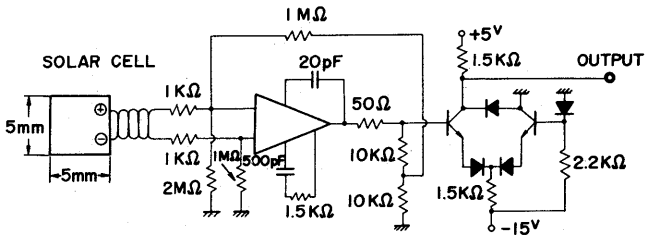


Figure 4—Photodetector and Amplifier circuit

of the solar cells were used to receive the Gray coded image beams corresponding to the coordinates X and Y. Inspection of encoding operation was carried out by detecting the diffraction intensity of the sign bit by the thirteenth solar cell. Output in the sign bit controlled also the on-off of the monitor brightness. After the photocurrents from the solar cells were amplified up to the logic level of DTL by the operational amplifier circuits shown in Figure 4, the Gray to binary code-conversion was made in parallel. Each bit of the binary coded signal was compared with that of the previous signal stored in the register. When the mismatch of a single bit was found by mismatch detection circuits, the present binary coded signal was stored into the register and a pulse of 10 μs width was generated to brighten the monitor. The operation was carried out in order to check the exact change of the Gray code and protect the screen of monitor CRT. The binary code stored in the register was changed into analog voltages by D-A converters and fed to Tektronix type 601 storage CRT in order to monitor the pen movement. Hologram plate and the light pen are the essential components of the hologram tablet.

Hologram plate

We have two methods to form the hologram plate, viz., serial-recording of each small hologram, and common-bits-recording in which each of the bits common to all of the small holograms is recorded individually. We employed the latter to form hologram plate, because the method is similar to that making integrated circuits, and considerably economizes the labor to make hologram plate.

A schematic diagram showing how to form hologram plate is given in Figure 5. We have employed the method of the image hologram.[6] The thirteen masks for the Gray code were prepared. Each of them consisted of transparent or opaque patterns expressing "1" or "0", corresponding respectively to each of the bits common to all of the small holograms.

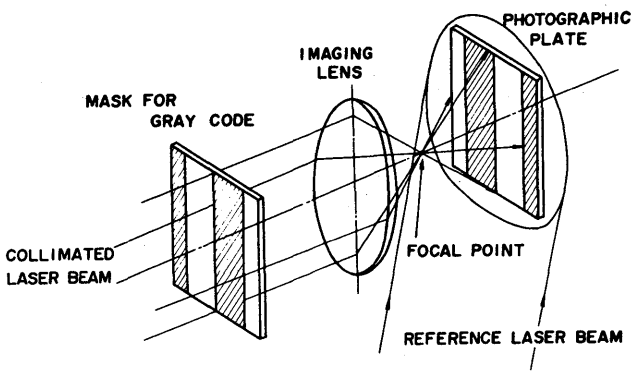


Figure 5—Schematic diagram forming the hologram plate

An imaging lens was set to form a magnified image of a mask for the Gray code on a photographic plate. When a collimated laser beam was used to illuminate the mask for the Gray code, the spatially modulated beams emerging from the mask (object beam) were converged at the rear focal point of the imaging lens and then imaged on the photographic plate. A plane wave laser beam (reference beam) was made to illuminate all over the photographic plate and then interfere with the object beam carrying the mask pattern for the Gray code. Recordings for different masks were made in a similar way, provided that the incident directions of the reference beam were shifted each time.

The reconstructed images are the replicas of the object beams and converge on the array of the solar cells. The distance from hologram plate to the array of the solar cells is the same as that from the rear focal point of the imaging lens to photographic plate.

The angular separation between the adjacent reconstructed image beams falling on the solar cells is equal to the angular shift imposed on the reference beam at the time of recording.

Figure 6 shows the hologram plate recorded on a Kodak 649-F plate and the Gray-coded images reconstructed from a small hologram of the hologram plate. The overlaps of the masks for the Gray code were achieved with an error less than ± 0.05 mm. The hologram plate of higher resolution will be formed by utilizing the ability of holography to achieve high bit densities with a great spatial redundancy and lens-like nature. The transmittivity of the hologram plate was about 80 percent. The diffraction efficiency better than 0.1 percent was obtained for each reconstructed image.

Light pen

Figure 7 shows the construction of the light pen. It consists of a light focusing glass fiber named SELFOC,[7]

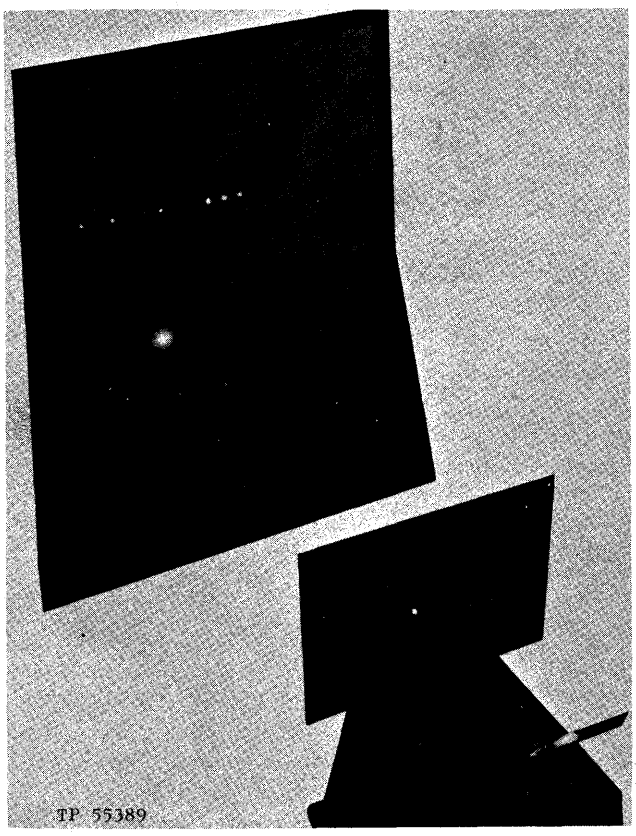


Figure 6—Hologram plate and reconstructed images

lens and a prism. We call it SELFOC light pen. SELFOC is a lens-like optical guide of glass fiber with a parabolic distribution of refractive indices. SELFOC has the following advantages as compared with a clad type optical fiber.

1. Laser beam transmission without band-limitation and waveform distortion.
2. Low-loss transmission and conservation of polarization plane of incident beam.

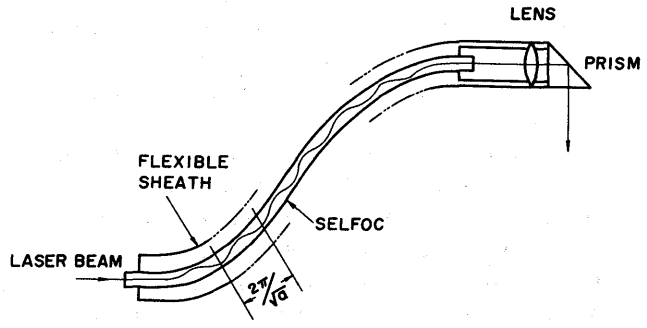


Figure 7—Construction of the SELFOC light pen

3. Realization of a lens with tiny aperture and that with an ultrashort focal length.
4. Increased flexibility due to a small diameter.

The refractive index n of the fiber in the radial direction is given by $n = n_0(1-ar^2/2)$, where n_0, r, and a are refractive index on the optic axis, distance from optic axis, and a constant, respectively.

At present SELFOC optical fibers with diameters from 50 μm to a few mm are available. Although a varies with fiber diameter, the difference between the refractive index on the optic axis and that at the periphery can be made 0.1.

The SELFOC optical fiber used in the SELFOC light pen was 50 cm long with a diameter of 0.2 mm. The coefficient a was 0.5 mm^{-2}, and the transmission loss was less than 0.3 dB/m. The mode pattern of a laser beam after passing through the SELFOC light pen was scarcely deformed as shown in Figure 8. The SELFOC could be bent with a radius of curvature less than 10 cm without a noticeable deformation in pattern. The lens on an end of the SELFOC light pen was used to collimate the emerging laser beam to a spot size of 0.25 mmϕ.

Performance

A paper placed on the writing surface provided hard copies of input patterns written or drawn by a ball-point pen. Movements of ball-point pen were followed by the SELFOC light pen and were instantaneously encoded as reconstructed images from the hologram plate. The solar cells adequately covered the fluctuations of the reconstructed image beams caused by jolting of the SELFOC light pen and aberrations of the hologram plate. The signal-to-noise ratio of the Gray-coded images on the solar cells was greater than

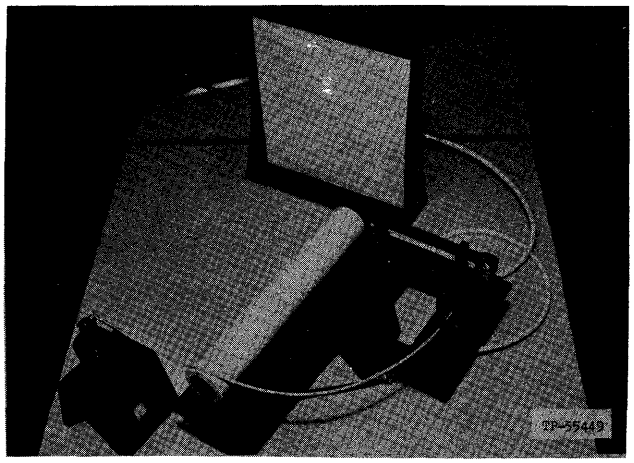


Figure 8—SELFOC light pen guiding a pencil of laser beam

Figure 9—Monitored characters drawn by the stylus on the hologram plate

7 dB, even when the hologram plate was disturbed by dust and scratches. Since the output voltage of the amplifiers was higher than 3 volts, it was found possible to use a laser source with an output less than 2 mW. Data rate was limited by the frequency response of the solar cell and the amplifier, and was obtained up to 10^4 positions per second.

Figure 9 gives an example of the monitored patterns displayed on the storage CRT. Deviations between the input patterns on paper and the displayed patterns on monitor CRT were within ± 0.5 mm. This is considered to be caused only by the essential disadvantage of the Gray code.

The pantograph link and the SELFOC light pen could be moved lightly and freely making the handling of hologram tablet easy. Excellent stability was confirmed for a long period of operation.

Most part of the cost of a hologram tablet consists of that of laser source and SELFOC light pen. In the prototype device, a laser source of three hundred dollars and a SELFOC light pen constructed at the expense of one hundred dollars were used. However, the price would be reduced to half by mass production. If we consider the possibility that a hologram tablet with a larger capacity is constructed by using only hologram plate (the number of record of the masks for the Gray code is made to be equal to the encoded bit number), the encoded bit number of photodetectors and amplifiers, the cost will not increase to a great extent.

CONCLUSION

It has been confirmed that the hologram tablet has many advantages such as high data rate, high resolution, compact size and low price.

These advantages have been achieved by utilizing holographic techniques, and SELFOC. The essential components of hologram tablet are the hologram plate containing small holograms, each records the Gray-coded coordinate X and Y and the SELFOC light pen. The techniques in the hologram tablet can be utilized several ways such as the position control of NC and the feature extraction for pattern recognitions (using the smoothing function of small hologram in the characterized hologram plates).

Although experiments were carried out with a prototype hologram tablet having 64 x 64 small holograms of 0.5 mm pitch, hologram tablets with much larger capacity and higher density could be constructed without great difficulty. The possibility that the increase in capacity and density will not bring a considerable rise in cost is to be found.

Experiment of graphic manipulations with the hologram tablet connected to a computer system is in progress.

ACKNOWLEDGMENTS

The authors wish to thank Dr. T. Uchida and Dr. F. Saito for their helpful suggestions in this work.

REFERENCES

1 M R DAVIS T O ELLIS
The rand tablet: A man-machine graphical communication device
AFIPS Conference Proceedings Fall Joint Computer Conference Volume 26 pp 325-331 1964

2 J F TEIXEIRA R P SALLEN
The sylvania data tablet: A new approach to graphic data input
AFIPS Conference Proceedings Spring Joint Computer Conference Volume 30 pp 315-321 1968

3 E G RAMBERG
The hologram-properties and applications
RCA rev Volume 27 pp 467-499 December 1966

4 L K ANDERSON
Holographic optical memory for bulk data storage
Bell Lab Record Volume 46 pp 318-325 November 1968

5 L F SHEW J G BLANCHARD
A binary hologram digital memory
IEEE J of QE Volume QE-5 pp 333-334 June 1969

6 G B BRANDT
Image plane holography
Applied Optics Volume 8 pp 1421-1429 July 1969

7 T UCHIDA M FURUKAWA I KITANO K KOIZUMI H MATSUMURA
Optical characteristics of a light-focusing fiber guide
To be published in IEEE J of QE

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

OFFICERS AND BOARD OF DIRECTORS OF AFIPS

President

Dr. Richard I. Tanaka
California Computer Products, Inc.
2411 W. LaPalma Avenue
Anaheim, California 92803

Vice President

Mr. Keith W. Uncapher
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Secretary

Mr. R. G. Canning
Canning Publications, Inc.
134 Escondido Avenue
Vista, California 92083

Treasurer

Dr. Robert W. Rector
Cognitive Systems, Inc.
319 S. Robertson Boulevard
Beverly Hills, California 90211

Executive Director

Dr. Bruce Gilchrist
AFIPS Headquarters
210 Summit Avenue
Montvale, New Jersey 07645

Executive Secretary

Mr. H. G. Asmus
AFIPS Headquarters
210 Summit Avenue
Montvale, New Jersey 07645

ACM Directors

Mr. Walter Carlson
IBM Corporation
Armonk, New York

Dr. Ward Sangren
521 University Hall
2200 University Avenue
Berkeley, California

Mr. Donn B. Parker
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

IEEE Directors

Mr. L. C. Hobbs
Hobbs Associates, Inc.
P.O. Box 686
Corona del Mar, California 92625

Dr. Robert A. Kudlich
Wayland Laboratory
Raytheon Company
Boston Post Road
Wayland, Massachusetts 01778

Dr. Edward J. McCluskey
Department of Electrical Engineering
Stanford University
Palo Alto, California 94305

Simulation Councils Director

Mr. James E. Wolle
General Electric Company
Missile & Space Division
P.O. Box 8555
Philadelphia, Pennsylvania 19101

American Society for Information Director

Mr. Herbert Koller
ASIS
2011 Eye Street, N.W.
Washington, D.C. 20006

Association for Computation Linguistics Director

Dr. Donald E. Walker
Head, Language and Text Processing
The Mitre Corporation
Bedford, Massachusetts 01730

Special Libraries Association Director

Mr. Burton E. Lamkin
Office of Education
7th and D Streets, S.W.
Washington, D.C. 20202

Society for Information Display Director

Mr. William Bethke
RADC—(EME, W. Bethke)
Griffis Air Force Base
New York, New York 13440

Society for Industrial and Applied Mathematics Director

Dr. D. L. Thomsen, Jr.
IBM Corporation
Armonk, New York 10504

American Institute of CPA's Director

Mr. Noel Zakin
Manager, Computer Technical Services
AICPA
666 Fifth Avenue
New York, New York 10019

American Statistical Association Director

Dr. Martin Schatzoff
Manager, Operations Research
IBM Cambridge Scientific Center
545 Technology Square
Cambridge, Massachusetts 02139

American Institute of Aeronautics and Astronautics Director

Dr. Eugene Levin
Culler-Harrison Company
745 Ward Drive
Santa Barbara, California 93105

Instrument Society of America Director

Mr. Theodore J. Williams
Purdue Laboratory for Applied Industrial Control
Purdue University
Lafayette, Indiana 47907

JOINT COMPUTER CONFERENCE BOARD

Dr. Richard I. Tanaka—President
California Computer Products, Inc.
2411 W. LaPalma Avenue
Anaheim, California 92803

Mr. Richard B. Blue Sr.—ACM
1320 Victoria Avenue
Los Angeles, California

Mr. Keith W. Uncapher—Vice President
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Mr. John E. Sherman—SCI
Lockheed Missiles and Space Company
Org. 19–30, Building 102
P.O. Box 504
Sunnyvale, California

Dr. Robert W. Rector—Treasurer
Cognitive Systems Inc.
319 S. Robertson Blvd.
Beverly Hills, California 90211

Dr. Robert A. Kudlich—IEEE
Raytheon Co. Equipment Division
Wayland Laboratory
Boston Post Road
Wayland, Massachusetts 01778

JOINT COMPUTER CONFERENCE COMMITTEE

Dr. A. S. Hoagland, Chairman
IBM Research Center
P.O. Box 218
Yorktown Heights, New York 10598

JOINT COMPUTER CONFERENCE TECHNICAL PROGRAM COMMITTEE

Mr. David Brown
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

FUTURE JCC GENERAL CHAIRMEN

1971 SJCC

Mr. Jack Moshman
RAMSCO
6400 Goldboro Road
Bethesda, Maryland 20034

1971 FJCC

Mr. Ralph R. Wheeler
Lockheed Missiles and Space Co.
Dept. 19–31, Bldg. 151
P.O. Box 504
Sunnyvale, California 94088

1970 FJCC STEERING COMMITTEE

General Chairman

Robert A. Sibley, Jr.
University of Houston

Vice Chairman

Eugene H. Brock
NASA—MSC

Technical Program

Larry E. Axsom—Chairman
IBM Scientific Center
Eugene Davis—Vice Chairman
NASA—MSC

Treasurer

Geary Eppley
Agency Records Control, Inc.
Don L. Carmichael—Assistant
Peat, Marwick, Mitchell & Co.

Secretary

Irma J. Morgan
Philco Ford Corporation

Local Arrangements

James N. Gay—Chairman
Hybrid Systems, Inc.
E. H. Hartung—Vice Chairman
Gulf Oil Corporation

Local Arrangements Coordinator

Howard E. Reddy
Pace Management Corporation

Public Relations

John Wilson—Chairman
The Phillips Agency
Larry Goldman—Vice Chairman
Thomas J. Tierney and Associates

Special Activities

Joe B. Wyatt—Chairman
University of Houston
R. A. Westerhouse—Vice Chairman
Computer Complex

Registration

Ed Mulvaney—Chairman
Control Data Corporation
Larry Byrne—Vice Chairman
Milchem, Inc.

Publications

R. S. Woodruff—Chairman
Lockheed Electronics Co.
Jury Lewisky—Vice Chairman
Lockheed Electronics Co.

Exhibits

F. J. Kirkpatrick—Chairman
Infotronics, Inc.
Robert J. Mobilia—Vice Chairman
Honeywell Computer Control

SCi Representative

James Van Artsdalen
NASA—MSC

ACM Representative

M. Stuart Lynn
IBM Scientific Center

IEEE Representative

Curt F. Fey
Xerox Data Systems

SESSION CHAIRMAN, PANELISTS AND REVIEWERS

SESSION CHAIRMEN

Ed Battiste
J. D. Baum
Willard Bouricius
Marc Connelly
James R. Deline
Dan Drew
E. A. Feustel
Karl Heinrichs
R. A. Kaenel
A. I. Katz
Billy V. Koen
Robert Korfhage
Lenord Litman
Michael A. Melkanoff
Rudolph Motard
R. R. Muntz
Dick Nance
C. V. Ramamoorthy
James L. Raney
S. Rosen
Art I. Rubin
Sally Sedelow
C. Ray Wallace
Richard Watson

REVIEWERS

R. P. Abbott
C. T. Abraham
R. M. Aiken
G. Albers
R. M. Alden
R. P. Allen
P. Altherton
E. B. Altman
L. K. Anderson
T. C. Anderson
F. Anzelmo
A. Arakawa
M. Arbab
P. Armer
G. N. Arnovick
W. L. Ash
M. M. Astrahan
D. C. Augustin
J. D. Aron
H. L. Babin
G. F. Badger, Jr.
J. A. Baker
D. L. Ball
N. A. Ball
M. Ballot
A. E. Barlow
B. B. Barnes
B. H. Barnes
R. M. Barnett
M. N. Bartakke
J. Bartlett
F. Bates
R. V. Bayles
J. A. Bayless
G. A. Bekey
M. J. Beniston
R. Bennett
P. T. Berning
M. I. Bernstein
W. P. Bethke
D. Bjorner
D. V. Black
J. A. Bloomfield
D. Bobrow
G. Boer
M. J. Bodoia
G. R. Bolton
H. Borko
G. H. Born
E. Bosch
H. Bratman
B. Brawn
R. L. Brening
R. D. Brennan
N. D. Brewer
J. D. Brooks
B. W. Brown
D. C. Brown
J. R. Brown, Jr.
K. Brown
G. E. Bryan
C. A. Caceres
M. A. Calhoun
E. D. Callender
T. W. Calvert
A. V. Campi
R. H. Canaday
D. G. Cantor
D. W. Cardwell
R. B. Carlson
R. L. Carmichael
C. C. Carroll
W. C. Carter
L. J. Chaitin
J. M. Chambers
R. C. Cheek
J. Chernak
B. F. Cheyoleur
C. K. Chow
W. F. Chow
W. Chu
E. H. Clamons
D. Climenson
L. J. Clingman
A. Clymer
E. G. Coffman
D. Cohen
W. L. Colby
L. S. Coles
A. J. Collmeyer
S. Condon
M. M. Conners
R. L. Constable
R. Constant
A. E. Corduan
W. A. Cornell
I. W. Cotton
R. Crandall
D. E. Crawford
B. Creasy
A. J. Critchlow
H. A. Crosby
J. D. Crunkleton
N. Cserhalmi
C. Csuri
A. G. Dale
J. A. Daly
D. A. Darms
C. M. Davis
R. J. P. DeFigueredo
P. De Jong
P. B. Denes
P. J. Denning
J. E. Dennis, Jr.
H. Denslow
E. Desautels
J. Dierman

H. Dinter
D. L. Dittberner
G. G. Dodd
T. L. Drake
R. C. Dubes
J. C. Duffendack
M. A. Duggan
A. I. Dumey
T. J. Dylewski
L. D. Earnest
L. B. Edwin
H. S. Ed Tsou
R. F. Elfant
W. J. Erikson
E. R. Estes
S. E. Estes
C. C. Farrington
G. A. Fedde
E. A. Feustel
F. Field
M. S, Field
R. T. Filep
O. Firschein
R. V. Fitzgerald
J. L. Flanagan
J. E. Foster
F. H. Fowler, Jr.
M. R. Fox
C. V. Freiman
P. J. Friedl
J. Friedman
L. M. Fulton
A. Futterweit
R. L. Gamblin
R. M. Gardner
G. E. Gareis
M. M. Gold
D. G. Gordon
D. F. Gorman
J. A. Gosden
M. H. Gotterer
E. M. Greenawalt
H. D. Greif
D. Gries
D. W. Grissinger
A. Guzman
T. G. Hagan
M. J. Haims
J. E. S. Hale
W. J. Hale
M. Halpern
M. H. Halstead
C. Hammer
M. Hanan
F. M. Haney
A. G. Hanlon
D. R. Haring
J. O. Harrison, Jr.
R. D. Hartwick
A. Hassitt
K. E. Haughton
J. F. Heafner
M. F. Heilweil
W. A. Helbig
P. J. Hermann
B. Herzog
G. E. Heyliger
J. H. Hiestand
A. N. Higgins
R. H. Hill
J. H. Hinrichs
A. D. C. Holden
G. L. Hollander
D. W. Holmes
R. L. Hooper
J. A. Howard
D. K. Hsiao
B. Huberman
T. A. Humphrey
E. Hunt
P. J. Hurley
M. R. Irwin
R. A. Ito
L. Jacobs
E. A. Jacoby
L. F. Jarzomb
R. E. Jeffries
B. Johnson
W. L. Johnson
E. R. Jones
N. D. Jones
E. C. Joseph
J. R. Jump
P. Kadakia
R. Y. Kain
V. A. Kaiser
M. J. Kaitz
J. F. Kalbach
E. Katell
S. M. Keathley
C. H. Kellogg
D. S. Kerr
R. E. King
E. S. Kinney
L. Kleinrock
A. Klinger
K. E. Knight
P. Knowlton
M. Kochen
H. R. Koen
E. C. Koenig
J. S. Koford
A. Kolk, Jr.
H. G. Kolsky
J. Kopf
P. R. Kosinski
L. D. Kovach
J. H. Kuney
J. Kurtzberg
A. Kusahara
K. C. Kwan
D. Laiti
B. W. Lampson
R. C. Larkin
D. J. Lasser
E. G. Lean
R. C. T. Lee
W. T. Lee
M. Lehman
J. Lennie
A. S. Lett
J. M. Lewallen
W. E. Lewis
W. W. Lichtenberger
H. P. Lie
C. R. Lindholm
R. Linebarger
T. P. Linville
H. Lipton
H. Liu
K. M. Lochner, Jr.
E. S. Loebenstein
R. D. Lohman
H. A. Long
R. G. Loomis
D. L. Magill
W. Main
C. M. Malone
R. L. Mandell
M. Marcotty
I. Marshall
W. L. Martin
R. L. Mattison
R. Mattson
H. E. Maurer
L. H. Maxson
C. H. Mays
M. E. McCoy
R. McDowell
F. W. McFarlan
J. L. McKenney
P. T. McKiernan
R. S. McKnight
J. McLeod
M. W. McMurran
L. P. McNamee
M. Meicler
M. A. Melkanoff

H. W. Mergler
J. C. Michener
B. J. Michielsen
W. F. Miller
H. D. Mills
J. Minker
B. A. Mitchell
E. E. L. Mitchell
B. Mittman
J. O. Mohn
M. Montalbano
M. F. Moon
C. G. Moore
D. W. Moore
R. K. Moore
R. A. Moran
H. L. Morgan
L. W. Morrison
M. S. S. Morton
G. J. Moshos
J. H. Munson
J. K. Munson
A. W. Muoio
J. J. Murphy
D. M. Murray
F. W. Murray
R. P. Myers
J. A. Narud
N. W. Naugle
G. W. Nelson
R. A. Nesbit
P. G. Neumann
F. Newman
W. M. Newman
C. B. Newport
R. V. Niedrauer
R. N. Nilsen
N. J. Nilsson
N. Nisenoff
J. D. Noe
D. L. Noles
W. A. Notz
J. A. O'Brien
P. L. Odell
K. O'Flaherty
W. J. B. Oldham, Jr.
M. J. O'Malley
J. T. O'Neil, Jr.
L. S. Onyshkevych
C. Opaskar
G. Oppenheimer
R. H. Orenstein
D. J. Orser
E. E. Osborne
J. T. Owens
D. R. Paden
C. V. Page
J. J. Pariser
J. Pearl
T. F. Penderghast
L. H. Peterson
J. K. Picciano
M. W. Pirtle
W. J. Plath
A. V. Pohm
J. H. Pomevene
J. A. Postley
A. W. Potts
R. C. Prather
R. J. Preiss
J. P. Pritchard, Jr.
J. S. Raby
M. S. Radwin
G. A. Rahe
C. V. Ramamoorthy
L. C. Ray
I. Remson
W. T. Rhodes
P. A. Richmond
F. C. Rieman
E. J. Roberts
R. M. Rojko
J. Roseman
C. A. Rosen
J. L. Rosenfeld
R. R. Rosin
D. L. Ross
P. M. Rubin
M. Rubinoff
F. Ruffino
R. L. Russo
J. D. Sable
J. M. Salzer
P. I. Sampath
J. L. Sanborn
W. B. Sander
L. Sashkin
P. Savage
D. Savitt
D. B. Saylors
M. W. Schellhase
W. E. Schiesser
A. J. Schneider
V. B. Schneider
J. E. Schwenker
S. Y. Sedelon
W. A. Sedelon
T. K. Seehuus
W. D. Seider
A. B. Shafritz
E. B. Shapiro
J. E. Shemer
P. C. Sherertz
J. Shih
J. S. Shipman
D. L. Shirley
S. Shohara
G. E. Short
R. L. Shuey
G. T. Shuster, Jr.
I. Shy
E. H. Sibley
L. C. Silvern
R. F. Simmons
R. M. Simons
Q. W. Sinkins
P. G. Skelly
D. R. Slutz
T. A. Smay
B. L. Smith
L. M. Spandorfer
C. F. Spitzer
F. W. Springe
T. B. Steel, Jr.
H. H. Steenbergen
J. K. Stephens
D. H. Stewart
W. A. Sturm
R. K. Summit
A. Svoboda
P. A. Szego
R. S. Taylor
R. W. Taylor
A. Tephtz
L. G. Tesler
R. E. Thoman
E. M. Thomas
M. D. Thompson
E. N. Timmreck
A. A. Toda
F. M. Tonge
G. R. Trimble, Jr.
G. H. Turner, Jr.
G. T. Uber
L. Uhr
W. R. Uttal
W. Utz
R. L. Van Tilburg
V. Vemuri
S. J. Viglione
R. Von Buelow
A. H. Vorhaus
S. Waaben
R. A. Wagner
S. E. Wahlstrom
J. V. Wait
P. D. Walker
C. J. Walter

C. Walton
G. Y. Wang
H. R. Warner
K. Wasserman
M. C. Watson
C. W. Watt
A. L. Weihrer
B. Weinberg
M. N. Weindling
L. H. Weiner
C. Weissman
R. R. Wheeler
G. Wiederhold
R. L. Wigington
R. C. Wilborn
L. C. Wilcox
M. Wildmann
D. A. Willard
T. G. Williams
T. J. Williams
A. N. Wilson
C. A. Wilson
D. E. Winer
H. Wishner
R. P. Wishner
E. W. Wolf
J. E. Wolle
R. C. Wood
F. Worth
J. H. Worthington
J. H. Wright
K. R. Wright
S. L. Wright
R. E. Wyllys
J. C. Wyman
J. W. Young
L. S. Young
D. C. Zatyko
N. S. Zinbel
A. S. Zukin

PANELISTS

D. Beach
U. N. Bhat
C. R. Blair
Jack Brooks
T. E. Cheatham, Jr.
S. Crocker
P. J. Denning
D. S. Diamond
A. Frederickson
B. A. Galler
N. Gorchow
H. R. J. Grosch
F. E. Heart
R. Howe
H. R. Koller
G. Korn
R. Lawrence
W. A. Leby
A. E. Lewis
S. Levine
J. Mauceri
J. Minker
H. S. McDonald
C. B. Newport
J. W. O'Byrne
J. F. Ossanna
T. C. O'Sullivan
G. Salton
J. H. Saltzer
J. E. Sammet
L. L. Selwyn
J. E. Shemer
T. B. Steel, Jr.
F. N. Trapnell
D. H. Vanderbilt
V. N. Vaughan
P. Wegner

FJCC 1970 PRELIMINARY LIST OF EXHIBITORS

Addison-Wesley Publishing Company
Addmaster Corporation
Addressograph Multigraph Corporation
Advance Research, Inc.
Advanced Information Systems, Inc.
Advanced Memory Systems, Inc.
Advanced Space Age Products, Inc.
Advanced Terminals, Inc.
AFIPS Press
Airoyal Mfg. Co.
Allen-Babcock Computing, Inc.
Allied Computer Technology, Inc.
American Data Systems
American Elsevier Publishing Co., Inc.
American Regitel
American Telephone and Telegraph Co.
AMP Incorporated
Ampex Corp.
Anderson Jacobson, Inc.
Applied Computer Systems, Inc.
Applied Digital Data Systems, Inc.
Applied Magnetics Corporation
Association for Computing Machinery
Atlantic Technology Corp.
Atron Corp.
Audio Devices, Inc.
Auerbach Info., Inc.
Auricord Div.—Scoville Mfg., Co.
Automata Corp.
Auto-Trol Corp.
Beehive Medical Electronics Inc.
The Bendix Corporation
BIT, Inc.
Boeing Computer Services
Boole & Babbage, Inc.
Bridge Data Products, Inc.
Brogan Associates, Inc.
Bryant Computer Products
Bucode, Inc.
The Bunker-Ramo Corporation
Business Press International, Inc. (Information Week)
California Computer Products, Inc.
Call-A-Computer, Inc.
Cambridge Memories, Inc.
Canadian Government Exhibition Commission
Centronics Data Computer Corp.
Century Data Systems, Inc.
Cincinnati Milacron, Inc.
Clare-Pendar Company
Codex Corp.
Cogar Corp.
Collins Radio Company
Colorado Instruments Inc.
ComData Corporation
Communitype Corporation
Compat Corp.
CompuCord, Inc.
Computek, Inc.
Computer Automation, Inc.
Computer Communications, Inc.
Computer Complex, Inc.
Computer Design Publishing Corp.
Computer Devices, Inc.
Computer Micro-Image Systems, Inc.
Computer Sciences Corporation
Computer Synetics, Inc.
Computer Terminal Corporation
Computer Terminals of Minnesota
Computer Transmission Corporation
Computerworld
Comress
Consultants Associated, Inc.
Control Data Corporation
Control Devices, Inc.
Courier Terminal Systems, Inc.
CSPI (Computer Signal Processors, Inc.)
Daedalus Computer Products, Inc.
Data 100 Corporation
Data Card Corporation
Data Computer Systems, Inc.
Data General Corp.
Dataline Inc.
Datamate Computer Systems, Inc.
Datamation
Datapac Incorporated
Data Printer Corp.
Data Processing Magazine (North American Pub. Co.)
Data Product News
Data Products Corporation
Dataram Corporation
Data Systems News/Newstape
Datatype Corp.
Datawest Corporation
Datotek, Inc.
Delta Data Systems Corporation
Diablo Systems, Incorporated
A. B. Dick Company
DID, Data Input Devices, Inc.
Digi-Data Corporation
Digital Equipment Corporation
Digital Information Devices, Inc.
Digital Information Systems Corp.
Digital Resources Corp. Hybrid Systems Div.
Digital Scientific Corporation
Digitronics Corporation
Dresser Systems, Inc.

Dylaflo Business Machines Corp.
Eastman Kodak
EDP News Service
Edutronics Systems, International Inc.
Eldorado Electrodata Corp.
Electronic Arrays, Inc. (Systems Div)
Electronic Arrays, Inc. (Components Div)
Electronic Laboratries, Inc.
Electronic Memories & Magnetics
Electronic News—Fairchild Pubs.
Engineered Data Peripherals Corp.
Fabri-Tek, Inc. (Memory Products Div)
Facit-Odhner, Inc.
Ford Industries, Inc.
Four-Phase Systems, Inc.
General Electric Company (Bull Corp)
General Electric Company (Scotia)
General Instrument Corporation
General Kinetics Incorporated
Genisco Technology Corporation
The Gerber Scientific Instrument Co.
Gould Inc., Graphics Div.
GRI Computer Corp.
Hayden Publishing Company, Inc.
Hazeltine Corp.
Hetra, Inc.
Hewlett-Packard
Hitachi America, Ltd.
Hi-Tek Corp. (Electronics Div)
Honeywell Computer Control Div.
Honeywell—EDP
Houston Instrument
Howard Industries
IBM Corporation
IDAK Corporation
IEEE Computer Group
IER Corp.
Image Systems, Inc.
Incoterm Corporation
Inforex, Inc.
Information Data Systems, Inc.
Information Displays, Inc.
Information International, Inc.
Information Storage Systems, Inc.
Infotronics Corp.
Interactive Info Systems, Inc.
Interdata
International Computer Products, Inc.
International Computers Ltd.
International Data Corp.
Kennedy Company
Keymatic Data Systems Corp.
Kongsberg Systems, Inc.
Kybe Corporation
Lenkurt Electric
Licon Div. Illinois Tool Works, Inc.
Lipps., Inc.
OEM Products, Automated Business Systems Div.
Litton Industries
Litton DATALOG Div.
Lockheed Electronics, Data Products Div.
Logicon, Inc.
Lundy Electronics & Systems, Inc.
M&M Computer Industries, Inc.
The Macmillan Company
Magnusonic Devices, Inc.
MAI Equipment Corp.
Marshall Data Systems, Div. of Marshall Ind.
MCI
Memorex
Memory Systems, Inc.
Memory Technology, Inc.
Microform Data Systems, Inc.
Micro Systems, Inc., A Microdata Subsidiary
Milgo Electronic Corp., Int'l. Communications Corp.
Mobark Instruments Corp.
Modern Data
Mohawk Data Sciences Corp.
R. A. Morgan
MSI Data
NCR
Nemonic Data Systems, Inc.
Noller Control Systems
Nortec Computer Devices, Inc.
Nortronics Company, Inc.
Novar
Nuclear Data, Inc.
Numeridex Tape Systems, Inc.
Odec Computer Systems, Inc.
Omega-t Systems, Inc.
Omnitec Corp.
On Line Computer Corp.
Optical Memory Systems
Optel Corporation
Path Computer Equipment, Inc.
Penril Data Communications, Inc.
Peripheral Equipment Corp.
Peripheral Technology, Inc.
Periphonics Corp.
Photophysics, Inc.
Plessey Electronics Corp.
Prentice Electronics Corp.
Prentice Hall, Inc.
Princeton Electronic Products, Inc.
Quantum Science Corp.
Raytheon Computer
RCA Memory Products Div.
Realist Microform Products Div.
Recortec, Inc.
Redcor Corp.
Remex Electronics, A Div. of Ex-Cell-O Corp.
Research/Development Magazine

RFL Industries, Inc.
Royco Instruments, Inc.
Sagetic Corporation
Sangamo Electric Company
Science Accessories Corporation
Scientific Control Corporation
Singer Company, Friden Div.
Singer-Librascope
Singer Micrographic Systems
Singer Telesignal
S.I.N.T.R.A.
Sonex, Inc.—I/Onex Division
Spartan Books
Standard Memories, Inc.
Storage Technology Corporation
Sykes Datatronics
Sylvania
Syner-Data Inc.
SYS Computer Corporation
Stromberg Datagraphix, Inc.
Tally Corporation
TDK Electronics Corp.
Technical Concepts, Inc.
Tektronix, Inc.
Teletype Corporation
Telex Computer Products
Tel-Tech Corp.
Tennecomp Systems, Inc.
Texas Instruments
Timeplex, Inc.
Time Share Peripherals Corp.
Time-Zero Corporation
Tops On-Line Services, Inc.
Tracor Data Systems
Treck PhotoGraphic Inc.
Trio Laboratories, Inc.
Typagraph Corporation
Ultronic Systems Corp.
United Business Communications, Inc.
United Telecontrol
Univac, Div. of Sperry Rand Corp.
Universal Data Acquisition Co.
Universal Graphics, Inc.
Vanguard Data Systems, Inc.
Varian Data Machines
Varisystems Corporation
Vermont Research Corporation
Versatec
Viatron
Video Systems Corporation
Wabash Computer Corp., PI Div.
Wang Computer Products, Inc.
Wang Laboratories, Inc.
Warner Electric
Western Union
Westinghouse Electric Corp.
John Wiley & Sons, Inc.
Xerox Corporation
Xerox Data Systems
Zeta Research, Inc.

AUTHOR INDEX