DATABUS 2 DB2PGS/DB2SYS User's Guide

Version 5

July, 1975

Model Code No. 50169

DATAPOINT CORPORATION

The Leader in Dispersed Data Processing

DATABUS 2
DB2PGS/DB2SYS

User's Guide

Version 5

July, 1975

Model Code No. 50169

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

DATABUS, the Datapoint Business Lanugage, is a family of high-level programming languages designed especially for the Datapoint 2200, Datapoint 1100, and their peripherals.

Unlike conventional small computers, which are built and shipped with little knowledge aforehand of what data processing devices will be attached to them, each Datapoint 2200/1100 computer leaves the factory with at least video display, keyboard, dual cassette tape drives and a variable quantity of solid-state memory. This concept allowed the Datapoint systems programmers to construct a high-level language that could take full advantage of the built-in peripherals that are part of every Datapoint 2200/1100. The language is especially useful in commercial environments where jobs must be written quickly.

Source code for the DATABUS language is created using the general purpose cassette editor. The source code is then converted into object code by the DATABUS compiler. The resulting object code can then be loaded by the standard cassette loader and interpreted by the DATABUS interpreter.

This manual describes an upgraded version of DATABUS 2 designed especially to run in the 8K Datapoint 1100 with dual cassette decks and a local or Servo printer. It will also run in the Version II Datapoint 2200 with 8K or more of memory. The compiler and interpreter may be configured to the machine memory size and printer type being used. The particular configuration will determine the available memory left for user programs. Except for the control instructions CHAIN, ACALL, and TRAP, the I/O instructions KSENSE and DSENSE, and the cassette input/output instuctions, all DATABUS 2 statements in this version are upward compatible with DATASHARE 3 statements.

Version 5 of DATABUS 2 contains the following new features and revisions to Version 4. Note that some of the changes are not upward compatible with Version 4.

1. The interpreter is now a self contained program. The use of the CTOS symbolic loader has been eliminated; therefore, reference to files by symbolic name is not allowed in the CHAIN instruction. Instead, the octal cassette file number must be supplied. The file can then be loaded by the cassette loader. If the file number specified is not on the interpreter system tape or a loader

failure occured, the program will abort and a return to the MASTER program is performed. The interpreter will be released as a CTOS multi-file tape with the interpreter as file 2 and a MASTER program as file 3. Databus object programs may be added to this CTOS tape and the interpreter run from CTOS. The user may also create a load-and-go interpreter tape with the CTOS command handler. The tape would consist of the interpreter and MASTER program followed by other object file programs. (Note Section 7).

2. The interpreter configurator is no longer a seperate program. It is included in the interpreter itself and works in a similar manner as UNITERM. When the interpreter is loaded, the configuration questions (printer to be used, write verify, etc) will be asked. If requested, the responses will be recorded on the interpreter tape so that the options do not have to be re-entered next time the interpreter is loaded. (Note Section 7).

3. The function of the released Databus MASTER program has been greatly extended. Besides allowing the user to choose what program on the Interpreter system tape he wishes to run under the Databus interpreter, the MASTER program also allows the user to rewind and prepare both the front and rear deck, list the front deck text file on the screen or printer, backspace a record on the front deck, write an end-of-file mark on the front deck, read a record from the front deck and display it on the screen, write an 80-character variable to the front deck, enter an 80 character string from the keyboard, copy the front deck file to the rear deck file, and copy the rear deck file to the front deck. Note Section 7.2 for detailed operating instructions of the released MASTER program.

4. The compiler will be released as a LGO tape and, as in the interpreter, will no longer have an accompanying configuration program. The configuration procedures will be included in the compiler itself and work in a similar manner as in the interpreter. (Note Section 6).

5. All input/output routines (cassette, keyboard, display, and printer) of the interpreter have been made interruptable with hardware tape I/O foreground driven. The exceptions are PREPARE and REWIND on the rear deck. These routines use the cassette loader to search for the scratch file (file 040) on the rear deck so could not be foreground driven. Only one tape operation may be active at once causing following tape operations to wait until the current one is complete before continuing. For example, when a WRITE statement is encountered, the write foreground is initiated and the program proceeds to the next statement. If the next

statement was a READ, the program would wait until the previous WRITE was completed before continuing. If trap and error conditions, such as end-of-file, end-of-tape, parity error, occur in the foreground process, the trap or abort will not be executed until the next I/O statement, CHAIN, KEYIN time out list control encountered (*T), STOP, or other abort condition. The READ and BKSP statements use the foreground hardware read routine when a new physical record is required to complete the execution of the statement. Both types of statements wait until the foreground read is finished before continuing its execution. (Note Section 4.6).

6. The old numeric operations package (STATH) has been replaced by the new short STATH package with the numeric KEYIN and DISPLAY routines re-written. This new STATH package has changed the way the multiply and divide instructions work. (Note Sections 4.4.3 and 4.4.4).

7. The BRANCH, STORE, and LOAD instructions have been modified to be compatible with DATASHARE 3. In the previous versions of DATABUS 2, the index of the instruction was rounded to no decimal places before execution (e.g. 1.7=2). This version truncates to no decimal places before execution (e.g. 1.7=1). (Note Sections 4.2.7, 4.3.13, 4.3.14, 4.4.7, and 4.4.8).

8. Keyin continuous and keyin time out have been added to the KEYIN routine as list controls. See Section 4.5.1 for definitions of those features.

9. In addition to the individual horizontal and vertical positioning KEYIN and DISPLAY list controls (*H and *V respectively), the list control, *Px:y, has been added which allows the setting of the horizontal and vertical positions with one list control. The x in the list control specifies the desired horizontal position and the y specifies the desired vertical position. This list control is compatible with DATASHARE which does not allow the *H or *V list controls. (Note Section 4.5.1).

10. The READ instruction will now read record compressed and space compressed data files. In addition, a continued READ is possible in which part of a logical record may be read in one READ statement and the rest of the record read in the next READ statement. The BKSP instruction will now backspace one logical record instead of one physical record. The new READ feature requires a 256-byte internal buffer and a one byte buffer pointer for each of the cassette tape decks. (Note Section 4.6).

11. The end-of-file trap has been eliminated. Instead of trapping the occurence of a file mark during the execution of a READ or BKSP statement, a test of the OVER flag should be made immediately after either type of statement. (Note Sections 4.6.3 and 4.6.6).

12. The WRITE instruction does not write compressed records. It still writes blocked records - one logical record per physical record with no space compression. However, a write-verify option has been added. If this option is set during interpreter configuration, each record written will be re-read to check for errors.

13. The RFAIL trap now indicates a bad read on the selected deck (such as parity errors). The FORMAT trap has been added to trap format errors when reading string data into numeric variables.

14. Local and Servo printer drivers are available in both the DATABUS 2 compiler and interpreter. The short Servo driver routine (version I) which prints one character at a time was used to obtain additional memory space. Backwards tabbing is no longer possible. The Servo driver routine is not interrupt driven in the compiler but is interruptable in the interpreter.

15. The TRAPCLR instruction has been added to this version of DATABUS 2. This instruction allows the user to clear a previously set trap location for a specified event (read failure, format error, etc.). If the event occurs after the trap location has been cleared, the program aborts with an appropriate message and returns to the MASTER program (see Section 4.2.7). ?

CHAPTER 2. STATEMENTS

There are three basic types of statements in DATABUS: comment, data definition, and program execution. Comment lines begin with a period and may occur anywhere in the program. Comments are most useful in explaining program logic and subroutine function and parameterization to enable someone reading through the program to understand it more easily. Data definition statements must occur before any program execution statements and are used for setting up all the variables in the program. All data definition statements must have unique labels. Program execution statements must appear after any data definition statements and may or may not have labels. The labels on program execution statements may be the same as labels on the data definition statements although it is not recommended for reasons of clarity. Program execution always begins with the first executable statement. The following are examples of DATABUS statements.

```
         NAME    DIM 35
         TITLE   INIT "TIME REPORT"
DEF      HOURS   FORM 5.2
         TOTAL   FORM 10.2
         RATE    FORM "2.50"
         TAX     FORM "10.00"
         .THIS IS A COMMENT
         START   DISPLAY *H1,*V1,*EF,TITLE
INPUT            PREPARE 2
                 KEYIN *H1,*V3,"NAME:",NAME
                 KEYIN *H1,*V4,"HOURS:",HOURS
         CALCR   MULT RATE BY HOURS
CALC             ADD HOURS TO TOTAL
                 SUB TAX FROM TOTAL
         OUTPUT  PRINT "NAME:",NAME,*30,"RATE:",RATE;
output           PRINT *40,"HOURS:",HOURS;
                 PRINT *50,"TOTAL:",TOTAL
                 WRITE 2,NAME,RATE,HOURS,TOTAL
                 GOTO START
```

Labels for variables and executable statements may consist of any combination of up to six letters and numbers, but it must begin with a letter. The following are examples of valid symbols:

```
     A
     ABC
     A1BC
     B1234
     ABCDEF
```

The following are examples of invalid symbols:

```
     ABCDEFG  (too long)
     HI,JK    (contains an invalid character)
     3DAS     (begins with a number)
```

Statements other than comments consist of a label field, an operation field, and a comment field. The label field is considered empty if a space appears in the first column. The operation field denotes the operation to be performed on the following operands. In many operations, two operands may be connected either by an appropriate preposition (BY, TO, OF, FROM, or INTO) or a comma. One or more spaces should follow each element in a statement, except where a comma is used, in which case it must be the terminating character of the previous element and may be followed by any number (including zero) of spaces. The following are all examples of valid statements:

```
     LABEL1   SUB TWO FROM DIFF
     LABEL2   SUB TWO OF DIFF
     LABEL3   SUB TWO, DIFF          THIS IS A COMMENT
     LABEL4   SUB TWO,DIFF
```

Note that any prepositions may be used, even if it does not make sense in English. The following are examples of invalid statements:

```
     LABEL1   SUB TWO DIFF           (missing connective)
     LABEL2   SUB TWO ,DIFF          (space before comma)
```

Certain DATABUS statements allow a list of items to follow the operation field. In many cases, this list can be longer than a single line, in which case the line must be continued. This is accomplished by replacing the comma that would normally appear in the list with a colon and continuing the list on the following line. For example, the two statements:

```
              PRINT A,B,C,D:
                    E,F,G
              PRINT A,B,C,D,E,F,G
```

will perform the same function. Note that the first entry of the continued line should not begin in the first column, the opcode field is the recommended place to begin the continued line.

CHAPTER 3. DATA TYPES

There are two types of data used within the Databus language. They are numeric strings and character strings. The numeric variable arithmetic instructions are performed on numeric strings and the string instructions are performed on character strings. There are also instructions available to allow movement of numeric strings into character strings and character strings into numeric strings.

Numeric strings have the following memory format:

```
0200      1      2      .      3      0203
```

The leading character (0200) is used as an indicator that the string is numeric. The trailing character (0203) is used to indicate the physical location of the end of the string. Note that the format of a numeric string is set at definition time and does not change throughout the execution of the program. When a move into a number occurs from a string or differently formatted number, reformatting will occur to cause the information to assume the format of the destination number (decimal point position and the number of digits before and after the decimal point) with truncation occurring to the left of the decimal point and rounding to the right.

Character strings have the following memory format:

```
9      5      THE QUICK BROWN      0203
                  9   ↑
```

The first character is called the logical length and points to the last character position currently being used as the end of the string (K in the above example). The second character is called the formpointer and points to a character position currently being accessed in the string (Q in the above example). The formpointer may point to any character in the string from position one through the logical length. The 0203 byte indicates the physical end of the string. The use of the logical length and formpointer in character strings will be explained in more detail in the discussions of each character string handling instruction.

Basically however, these pointers are the mechanism via which the programmer deals with individual characters within the string.

3.1 Variable Definition

Whenever a numeric or character string variable is used in a program, it must be "defined" at the beginning of the program using either the FORM, DIM, or INIT instructions. These instructions reserve the memory space described above for the data variable whose name is given in the label field. Note that all variables must be defined before the first executable statement is given in the program and that once an executable statement is given no more variables may be defined. Numeric strings are created with the FORM instruction while character strings are created with the INIT or DIM instruction.

3.2 Numeric String Variables

Numeric string variables are defined with the FORM instruction as shown in the following illustration:

```
EMRATE  FORM 4.2
XAMT    FORM " 382.4 "
```

In this example EMRATE has been defined as a string of decimal digits which can cover the range from 9999.99 to -999.99. The FORM instruction illustrated reserves space in memory for a number with four places to the left of a decimal point and two places to the right of a decimal point and initializes the value to zero. When the number is negative, one of the places to the left of the decimal point is used by the minus sign. XAMT, in the example, is defined with four places to the left of the decimal point and three to the right but with an initial decimal value of 382.400.

Care should always be taken when defining variables not to make them larger than will be needed for the values they will hold in the program. Making them larger than needed will set aside memory space that cannot otherwise be used and will reduce the overall space available to the program.

3.3 Character String Variables

Character strings are defined with either the dimension instruction, DIM, or the initialization instruction, INIT. The DIM reserves a memory space for the given number of characters, sets the length and formpointer to zero, and initializes all the characters to spaces. For example:

```
        ANAME DIM 24
```

A character string can also be defined with some initial value by using the INIT instruction. For example:

```
        TITLE INIT "PAYROLL PROGRAM"
```

initializes the string TITLE to the characters shown and gives it a logical and physical length of 15 and a formpointer of 1. Note that in the case of strings, the actual amount of physical space reserved is three bytes greater than the number specified in the DIM or quoted in the INIT instruction (TITLE occupies 18 bytes in memory, 15 of which hold characters).

CHAPTER 4. INSTRUCTIONS

Every statement other than a comment must contain an instruction. There are six classes of instructions to provide the basic types of operations the Datapoint 2200/1100 must perform. They are:

DIRECTIVES - These instructions are basically instructions to the compiler. Directives define variables and establish their initial values and sizes. They may also establish the size of the user program, or tell the compiler to continue an instruction from one line to the next.

CONTROL - These instructions control the order in which a program is executed. They permit transfer of control from one part of the program to another depending on the results of other operations, stopping the program, or loading and running another program stored on the system tape.

STRING - These instructions perform the various string handling operations on character strings. The operations include string move, append, match, character match and move, and manipulation of the formpointer.

NUMERIC VAIABLE ARITHMETIC - These instructions perform the basic arithmetic operations on numeric variables, transfer of a value from one variable to another, and comparison of one variable to another.

KEYBOARD, C.R.T., PRINTER INPUT/OUTPUT - These instructions perform the basic I/O functions to the mentioned devices.

CASSETTE TAPE INPUT/OUTPUT - These instructions perform the basic cassette tape handling functions for reading and writing tapes.

D C S N K C

4.1 Directive Instructions

4.1.1 FORM

The FORM instruction defines the length and initial value of a numeric string variable. The FORM instruction must be used with a label which is used as the variable name throughout the program. The maximum length of a numeric string variable may be 22 including the decimal point and minus sign.

Examples:

```
RATE   FORM "6.5"
AMT    FORM 6.2
ZERO   FORM 1
PI     FORM "3.14159"
```

If the numeric variable is defined with a quoted item, the same number of character positions are reserved in memory as are in the number between the quotation marks and the variable is initialized to the value given. In the above example RATE is dimensioned to a number with one place to the left and one place to the right of the decimal point, and initialized to a value of 6.5.

If the numeric variable is defined without quotes then the numbers that appear to the right and left of the decimal point specify how many positions to the right and left of the decimal point are reserved in memory. In the above example AMT reserves space in memory for a number with six places to the left of the decimal point and two places to the right of the decimal point and initializes the number to zero.

4.1.2 DIM

DIM defines a character string variable, determines its physical length in memory, and initializes its logical length and formpointer to zero. The DIM instruction must be used with a label which is used as the variable name throughout the program. The maximum length of a character string variable is 127.

Examples:

```
REFLBL  DIM 60
XCODE   DIM 6
MAXLEN  DIM 127
```

4.1.3 INIT

The INIT instruction is the same as the DIM instruction except that the initial value of the character string is established. This value may be initialized only by a quoted string. The INIT instruction establishes physical and logical lengths that are equal, and initializes the formpointer to one.

Examples:

```
HDING  INIT "REORDER FORM"
```

4.1.4 Common Data Areas

Since DATABUS has the provision to chain programs so that one program can cause another to be loaded and run, it is desirable to be able to carry common data variables from one program to the next. The procedure for doing this is as follows:

a. Identify those variables to be used in successive programs and in each program define them in _exactly_ the same order and size and at _beginning_ of each program. This is to cause each common variable to occupy the same locations in each program.

b. For the first program to use the variables, define them in the normal way.

c. For all succeeding programs place an asterisk in each FORM, DIM, or INIT statement as illustrated below to prevent those variables from being initialized when the program is loaded into memory.

Great care must be used when incorporating the feature into a program. An error in programming can produce strange results if a common variable is misaligned with respect to the variable in a previous program.

Example:

```
     MIKE FORM *4.2
     JOE  DIM  *20
     BOB  INIT *"THIS STRING WON'T BE LOADED"
```

4.1.5 Line Continuation

The KEYIN, DISPLAY, PRINT, READ, WRITE, LOAD, STORE, and BRANCH instructions allow statements to be continued from one line to the next.

These instruction statements may be continued to the next line if a colon (:) is the terminating character of the instruction. The colon replaces the comma separating the last entry of the first line from the first entry on the second line. The first entry of the second line should begin in the instruction field. Examples of each are given in the instruction section.

4.2 Control Instructions

4.2.1 GOTO

The GOTO instruction transfers control to the program statement indicated by the label following the instruction:

```
GOTO CALC
```

causes control to be transferred to the instruction labeled CALC.

The GOTO instruction may be conditional, however, and the transfer of control occurs only if a specified condition is met. Six possible conditions can be specified and are OVER, LESS, EQUAL, ZERO, and EOS. The conditions result from previously executed instructions and reference should be made to the discussion on the various operations for their meaning (EQUAL and ZERO are two different names for the same flag).

In the example:

```
GOTO CALC IF OVER
```

control is transferred to the instruction labeled CALC if an overflow occurred with the last arithmetic operation, otherwise, the next instruction following the GOTO is executed.

The sense of the condition can be reversed as follows:

```
GOTO CALC IF NOT OVER
```

meaning control is transferred only if overflow did not occur.

4.2.2 CALL

The CALL instruction is very similar to the GOTO instruction except that when a RETURN instruction is encountered after a transfer, control is restored to the next instruction following the CALL instruction. CALL instructions can be nested up to eight deep. That is, up to eight CALL instructions may be executed before a RETURN instruction is executed. If more than eight CALL instructions are executed before a RETURN, the stack will end-around. Being able to call subroutines eliminates the need to repeat frequently used groups of statements, and may be made conditional as discussed in the GOTO instruction.

Examples:

```
CALL FORMAT
CALL XCOMP IF LESS
```

4.2.3 RETURN

The RETURN instruction is used to transfer control to the location indicated by the top address on the subroutine call stack. This instruction has no operand field but may be conditional.

Examples:

```
RETURN
RETURN IF EQUAL
```

4.2.4 STOP

The STOP instruction causes the program to terminate and return to the MASTER Program.

The STOP instruction may be conditional as discussed under the GOTO instruction.

Examples:

```
STOP
STOP IF OVER
```

4.2.5 CHAIN

The CHAIN instruction enables the DATABUS 2 user to fetch and run another program on the interpreter system tape. Since the interpreter may be a LGO system without the CTOS catalog, program chaining must be done by program file number. The character string in the referenced variable provides the octal file number of the desired program. The number will begin at the formpointed character of the string and continue through the logical length or the first three characters if the logical length is longer than three. Leading zeros are allowed but leading blanks are not. The number must be octal and be between 1 and 040 inclusive if the interpreter system is LGO, or between 3 and 040 inclusive if the system is on a CTOS tape. File 040 is normally used as the scratch file on the rear deck but it may be referenced through CHAIN.

Example:

```
    NXTPGM INIT "020"
                 "
                 "
           CHAIN NXTPGM
```

causes file 020 (file 16 decimal) on the interpretive tape to be loaded into memory and run.

All foreground tape operations must be completed without trap or error conditions before the CHAIN is executed.

If the specified file number is not valid, a chain failure trap, CFAIL, will occur. If the specified program is not on the interpretive system tape or if the program did not load successfully, the current program will abort and a return to the MASTER program is made.

4.2.6 TRAP

TRAP is a unique instruction because it does not take action at the time it is executed in the program but specifies that a transfer of control should occur later if a specified event occurs. For example:

```
    TRAP EMSG IF EOT2
```

specifies that control should be transferred to EMSG if the end-of-tape is encountered on cassette deck two (front deck). The transfer that occurs is like the GOTO instruction. Once the trap location is set, transfer will continue to occur to that location until the trap is reset with another TRAP statement or cleared with the TRAPCLR instruction.

The events that may be specified are:

EOT(n)	-	End-of-tape mark on indicated device
RFAIL(n)	-	Read failure on indicated device (Parity error, bad file mark)
FORM(n)	-	String data read into numeric field from indicated device

n = 1 or 2
1 = cassette deck 1
2 = cassette deck 2

CFAIL	-	Specified file number of CHAIN instruction not valid

The program will abort if one of the above conditions occurs and the corresponding trap has not been set (see Section 7). The cassette I/O events which occur in foreground are not trapped until the next tape I/O instruction, CHAIN instrucion, KEYIN instruction with a keyin time out control (*T), STOP instruction, or some other abort condition.

Note that end-of-file may not be trapped. Instead, the OVER flag should be tested after a READ or BKSP instruction. Refer to the Sections 4.6.3 and 4.6.6 for further explanation.

4.2.7 TRAPCLR

The TRAPCLR instruction clears the current trap location for the specified event. For example:

```
TRAPCLR RFAIL1
```

clears the trap location set for the read failure on deck one event. If the event is not re-trapped with a TRAP instruction, the program will abort upon the next occurrence of the event.

4.2.8 BRANCH

The BRANCH instruction transfers control to a statement specified by an index.

For example:

```
            BRANCH N OF START,CALC,POINT
```

causes control to be transferred to the label in the label list pointed to by the numeric variable index N. (i.e. START if N=1, CALC if N=2, and POINT if N=3).

If the index is negative, zero, or larger than the number of variables in the list, control continues with the following statement. Note that the index is <u>truncated</u> to the no decimal places before it is used (e.g. 1.7=1).

The BRANCH instruction statement may be continued to the next line if a colon (:) is the terminating character of the instruction. The colon replaces the comma separating the last entry of the first line from the first entry on the second line. The first entry of the second line should begin in the instruction field.

Example:

```
LABEL  BRANCH N OF LOOP,START,READ,WRITE:
       PRNT,END
```

4.2.9 ACALL

The Assembly Language Call Instruction allows the user to call assembly language subprograms to be executed outside of the interpreter. The assembly language programs should not overlay any of the interpreter or the Databus user area which calls it, unless the program reloads the interpreter or user program before returning, in which case the user program should be restarted.

Example:

```
ACALL 020000
```

calls a subprogram starting at location 20000 octal. The location to be called may be decimal or octal, but must be a numeric literal. The last statement in the subprogram executed should be a RET to return to the interpreter to resume execution of the Databus program. Only one entry in the stack must be preserved by the assembly subprogram, and this should be at the top of the stack upon return, i.e. no calls should be made within the subprogram without corresponding returns. If the stack is destroyed, however, the user may resume by jumping to the Databus Entry Point for the interpreter (02076).

There are three ways to load these subprograms into memory. One is to have all the subprograms on one or more LOAD & GO tapes and load them into memory before loading the LOAD & GO Databus Interpretive Tape.

The second method is to use the Databus CHAIN instruction. With this method, the first instruction of every program chained to must be a jump to the assembly subprogram entry point of the Interpreter (i.e. JMP 02076). Jumping back to the Interpreter will cause execution of the next instruction after the chain. Using this method, all subprograms are filed on the Interpretive System tape and may be loaded in by the Databus user program. For example,

ASSEMBLY PROGRAM FOR DATABUS CALL

```
        SET    020000
SUBR    JMP    02076            RETURN TO DATABUS INTERPRETER
ENTRY   EX     BEEP             ASSEMBLY SUBPROGRAM
        HL     MESG
        LD     40
        LE     11
        CALL   DSP$
        RET
DSP$    EQU    07363
MESG    DC     'ACALL TEST MESSAGE',0203
        END    SUBR
```

The above subprogram ENTRY would be called by ACALL 020003.

The third method is to append the Databus Interpreter to the subprogram. This subprogram will always be resident when the interpreter is loaded. The CTOS command handler file routines will accomplish this.

<u>Caution</u>: Since this version of DATABUS 2 uses the interrupt feature of the DATAPOINT 1100 and the Version II 2200, all programs called with the ACALL instruction must be interruptable. For example,

```
        SET    020000
SUBR    JMP    02076
ENTRY   LA     0341             BEEP IF KEYBOARD & DISPLAY DEPRESSED
        DI
        EX     ADR
        EI
        IN
        ND     014
        XR     014
        RFZ
        EX     BEEP
        RET
```

4.3 Character String Handling Instructions

Each string instruction, except LOAD and STORE, requires either one or two character string variable names following the instruction. (Note that the MOVE instruction is capable of moving strings to numbers, numbers to strings, numbers to numbers and strings to strings. See Sections 4.3.4 and 4.4.5 for all descriptions of the MOVE instruction. In the following sections, the first variable will be referred to as the source string and the second variable will be referred to as the destination string.

4.3.1 CMATCH

CMATCH compares two characters, one taken from each of the source and destination operands. The characters to be compared may be from under the formpointer of a string variable, a quoted alphanumeric character, or a number. This number may be octal or decimal but it must have a value between 0 and 127 decimal.

An EOS condition occurs if the character is taken from a string which has a formpointer of zero, and no other conditions are set. Otherwise, the EQUAL and LESS conditions are set appropriately. The LESS condition is set if the second string character is less than the first string character.

Examples:

```
     CMATCH XDATA TO YDATA
     CMATCH Y,X
     CMATCH "A",DOG
     CMATCH DOG TO "B"
     CMATCH CAT,0101
```

4.3.2 CMOVE

CMOVE moves a character from the source operand to under the formpointer in the destination string. The character from the source operand may be a quoted alphanumeric, a number, or the character from under the formpointer of a string variable. If either operand has a formpointer of zero, an EOS condition and no transferral occurs.

Examples:

```
CMOVE XDATA,YDATA
CMOVE "A" TO CAT
CMOVE X,Y
CMOVE 0101 TO STRING
```

4.3.3 MATCH

MATCH compares two character strings starting at the formpointer of each and stopping when the end of either string is reached. If either formpointer is zero, the MATCH operation will result in only clearing the LESS and EQUAL flags and setting the EOS flag. Otherwise, the "length" of each string is calculated to be LENGTH-FORMPOINTER+1 and the LESS flag is set if the destination string length is less than that of the source string. The two strings are then compared on a character-for-character basis for the number of characters equal to the lesser of the two lengths. If all the characters match, the EQUAL flag is set. If they do not match, the LESS flag's meaning is changed to indicate whether the numeric value of the destination character (in the character pair) is less than the numeric value of the source character (LESS flag set) or vice versa (LESS flag reset). Some examples and their results follow:

Source	Destination	Result
ABCDE	ABCD	EQUAL,LESS
ABC	Z	NOT EQUAL,NOT LESS
ZZZ	AAA	LESS,NOT EQUAL
ABC	ABC	EQUAL,NOT LESS
ABCD	ABCDE	EQUAL,NOT LESS

Examples:

```
MATCH A TO B
MATCH STR1,STR2
```

4.3.4 MOVE

MOVE transfers the contents of the source string, starting from under the formpointer, into the destination string. Transfer into the destination string starts at the first physical character and when transfer is complete, the formpointer is set to one and the logical length points to the last character moved. The EOS flag is set if the ETX in the destination string would have been overstored and transfer stops with the character that would have overstored the ETX.

The MOVE instruction can also move character strings to numeric strings and vice versa. (The movement of numeric strings to numeric strings is discussed in Section 4.4.5.) A character string will be moved to a numeric string only if the character string is of valid numeric format (only digits, spaces, a leading minus sign, and one decimal point allowed). Otherwise, the numeric string is set to zero. Note that only the part of the character string starting with the formpointer is considered in the validity check and transferred if the string is of valid numeric format. The number in the character string will be reformatted to conform to the format of the numeric string. The TYPE instruction (see Section 4.3.10) is available to allow checking the character string for valid numeric format before using the MOVE instruction. When a numeric string is moved to a character string, all characters of the numeric item (unless the ETX would be overstored) are transferred starting with the first physical character in the destination string. The formpointer of the destination string is set to one and the logical length is set to point to the last character transferred.

Examples:

```
MOVE STRING TO STRING
MOVE A,B
MOVE STRING TO NUMBER
MOVE NUMBER,STRING
```

4.3.5 APPEND

APPEND appends the source string to the destination string. The characters appended are those from under the formpointer through under the logical length pointer of the source string. The characters are appended to the destination string starting _after the formpointed character_ in the destination string. The source string pointers remain unchanged, but the destination string pointers both point to the last character transferred. The EOS condition will be set if the new string will not fit physically into the destination string, but all characters that will fit will be transferred.

Examples:

```
APPEND SOURCE TO DEST
APPEND NAME,BUFF
```

4.3.6 RESET

RESET changes the value of the formpointer of the source string to the value indicated by the second operand. If no second operand is given, the formpointer will be reset to one. The second operand may be a quoted character, in which case the ASCII value minus 32 (space gives zero, ! one, " two, etc.) will be used for the value of the formpointer of the source string. The second operand may also be a character string, in which case the ASCII value minus 32 of the character under the formpointer of that string will be used for the value of the formpointer of the source string. The second operand may also be a numeric string or a number, in which case the value of the number will be used for the formpointer of the source string.

RESET also has the capability of extending the logical length of the first operand. If the formpointer value specified is past the logical length of the first operand, the logical length will be extended until it will accommodate the formpointer value. If this would cause the logical length to be past the physical end of the string, the logical length and formpointer will both be left pointing to the last physical character in the string. This feature is useful in extracting and inserting information within a large string. The EOS condition will be set if a change in the logical length of the first operand occurs.

Examples:

```
RESET XDATA TO 5
RESET Y
RESET Z TO NUMBER
RESET Z TO STRING
```

Note that the RESET instruction is very useful in code conversions and hashing of character string values as well as large string manipulation.

4.3.7 BUMP

BUMP increments or decrements the formpointer if the result will be within the string (between 1 and the logical length). If no parameter is supplied, BUMP increments the formpointer by one. However, a positive or negative literal value may be supplied to cause the formpointer to be moved in either direction by any amount. An EOS condition will be set and no change in the formpointer occurs if it would be less than one or greater than the logical length after the movement had occurred.

Examples:

```
BUMP CAT
BUMP CAT BY 2
BUMP CAT,-1
```

4.3.8 ENDSET

ENDSET causes the operand's formpointer to point where its logical length points.

Example:

```
ENDSET PNAME
```

4.3.9 LENSET

The LENSET command is implemented in Version 4 Interpreters only. LENSET causes the operand's logical length to point where its formpointer points.

Example:

```
     LENSET QNAME
```

4.3.10 TYPE

TYPE sets the EQUAL and ZERO condition if the string is of valid numeric format (only leading minus, one decimal point, and digits or spaces).

Example:

```
     TYPE ALPHA
```

4.3.11 EXTEND

EXTEND increments the formpointer, stores a space in the position under the new formpointer, and sets the logical length to point where the new formpointer points if the new logical length would not point to the ETX at the end of the character string. Otherwise, the EOS flag is set and no other action is taken.

Example:

```
     EXTEND BUFF
```

4.3.12 CLEAR

CLEAR causes the operand's logical length and formpointer to be zero.

Example:

```
     CLEAR NBUFF
```

4.3.13 LOAD

LOAD performs a MOVE from the character string pointed to by the index numeric operand, the second operand, to the first character string specified. The index must be a numeric string variable. The instruction has no effect if the index is negative, zero, or greater than the number of items in the list. Note, that the index is <u>truncated</u> to no decimal places before it is used (e.g. 1.7=1).

For example:

```
     LOAD AVAR FROM N OF NAME,TITLE,HEDING
```

causes the contents of one of the variables in the list, based on the value of the numeric variable N, to be moved into the first operand AVAR.

4.3.14 STORE

STORE performs a MOVE from the first character string specified to a character string in a list specified by an index numeric operand given as the second operand. The index must be a numeric variable. The instruction has no effect if the index is negative, zero, or greater than the number of items in the list. Note that the index is <u>truncated</u> to no decimal places before it is used (e.g. 1.7=1).

For example:

```
     STORE Y INTO NUM OF ITEM,ENTRY,ALINK,LIST
```

causes the contents of the first operand Y to be moved into one of the variables in the list, based on the value of the numeric variable NUM.

The LOAD and STORE instruction statements may be continued to the next line if a colon (:) is the terminating character of the instruction. The colon replaces the comma separating the last entry of the first line from the first entry of the second line. The first entry of the second line should begin in the instruction field.

Examples:

```
LABEL LOAD SYMBOL FROM N OF VAR,CONST,DEC:
      CNT,FLAG,LIST

      STORE NAME INTO N OF A,B,C,D,E,F,G:
      H,I,J,K,L,M
```

4.4 Numeric String Variable Arithmetic Instructions

All of the numeric variable arithmetic instructions have certain characteristics in common. Except for LOAD and STORE, each numeric variable arithmetic instruction is always followed by two numeric string variable names. The contents of the first variable is never modified and, except in the COMPARE instruction, the contents of the second variable always contains the result of the operation.

For example in:

```
ADD XAMT TO YAMT
```

the content of XAMT is not changed, but YAMT contains the sum of XAMT and YAMT after the instruction is executed.

Following each numeric string variable arithmetic instruction, the condition flags, OVER, LESS, and ZERO (EQUAL) are set to indicate the results of the operation. OVER indicates that the result of an operation is too large to fit in the space allocated for the variable (a result is still given with truncation to the left and rounding to the right, however). LESS indicates that the content of the second variable is negative following the execution of the instruction (or would have been in the case of COMPARE). ZERO (EQUAL) indicates that the value of the second variable is zero following the execution of the instruction.

Whenever overflow occurs, the higher valued digits that do not fit the variable are lost. For example, a variable is defined:

```
NBR42   FORM 2.2
```

and a result of 4234.67 is generated for that variable, NBR42 will contain only 34.67.

Whenever an operation produces lower order digits than a variable was defined for, the result is rounded up. A variable with the FORM 3.1 would contain:

46.2 for 46.213
812.5 for 812.483
3.7 for 3.666
3.9 for 3.850

Note that if an OVER occurs during an ADD, SUB, or COMPARE of two strings of different physical lengths, the result and the LESS condition flag may not be correct.

4.4.1 ADD

ADD causes the content of variable one to be added to the content of variable two.

Examples:

```
ADD X TO Y
ADD DOG,CAT
```

4.4.2 SUB

SUB causes the content of variable one to be subtracted from the content of variable two.

Examples:

```
SUB RX350 FROM TOTAL
SUB Z,TOTAL
```

4.4.3 MULT

MULT causes the content of variable two to be multiplied by the content of variable one. The restrictions on length with the new arithmetic package incorporated in this DATABUS 2 version requires that the sum of the number of characters in the two operands must be less than 32.

Examples:

```
    MULT DICK BY HARRY
    MULT W,Z
```

4.4.4 DIV

DIV causes the content of variable two to be divided by the content of variable one. The restrictions upon division operands is that the number of characters in the dividend plus the number of characters in the divisor plus two times the number of characters after the decimal point in the divisor must be less than 32. Division by zero results in the OVER condition begin set and the destination variable not being changed.

If the quotient cannot be represented fully in the destination variable format, the quotient will be rounded to the number of places in the destination variable if the divisor has at least one digit place after the decimal point. If there are no digit places after the decimal point in the divisor, the quotient will be truncated (rounded down) to the number of places in the destination variable.

Examples:

```
    DIV SFACT INTO XRSLT
    DIV X3,HOURS
```

4.4.5 MOVE

MOVE causes the content of variable one to replace the content of variable two.

Examples:

```
    MOVE FIRST TO SECOND
    MOVE A,B
```

4.4.6 COMPARE

COMPARE does not change the content of either variable but sets the condition flags exactly as if a SUB instruction had occurred.

Examples:

```
COMPARE XFRM TO YFRM
COMPARE RING,DING
```

Care should be used in defining variables to be compared. Comparison of variables in which the length of the first variable is longer than the length of the second variable results in an overflow condition. The OVER flag is set, and the EQUAL or ZERO flag is set to show the result of the comparison. However, the LESS flag is not set in this case.

4.4.7 LOAD

The LOAD instruction for numeric string variables selects an operand out of the list based on the index operand. It then performs a MOVE operation from the contents of the selected variable into the first operand. The index must be a numeric variable. If the index is negative, zero, or greater than the number of items in the list, then the instruction has no effect. Note that the index is <u>truncated</u> to no decimal places before it is used (e.g. 1.7=1).

For example:

```
LOAD CAT FROM N OF FACT,MULT,SPACE
```

causes the contents of one of the variables in the list, based on the value of the numeric variable N to be moved into the first operand CAT.

4.4.8 STORE

The STORE instruction for numeric variables selects an operand out of the list based on the index operand. It then performs a MOVE operation from the contents of the first operand into the selected variable. The index must be a numeric variable. If the index is negative, zero, or greater than the number of items in the list, the instruction has no effect. Note that the index is <u>truncated</u> to no decimal places before it is used (e.g. 1.7 =1).

For example:

```
STORE X INTO NUM OF VAL,SUB,TOT
```

causes the contents of the first operand X to be moved into one of the variables in the list, based on the value of the numeric variable NUM.

The LOAD and STORE instruction statements may be continued to the next line if a colon (:) is the terminating character of the instruction. The colon replaces the comma separating the last entry of the first line from the first entry on the second line. The first entry of the second line should begin in the instruction field.

Examples:

```
LABEL LOAD NUMBER FROM N OF N1,N2,N3,N4,N5:
          N6,N7,N8,N9

          STORE COUNT INTO N OF TIME,RATE,DIST,SPG:
          COST,TOT,SUM
```

4.5 Keyboard, C.R.T., Printer Input/Output Instructions

These statements move data between the program variables and the keyboard, screen, or printer. They each allow a list of variables to follow the operation mnemonic. This list may be continued on more than one line with the use of colon. The I/O list may contain some special control information besides the names of the variables to be dealt with. DATABUS has no formatting information other than the list controls and that which is implied by the format of the variables. The number of characters transferred is always equal to the number of characters physically allocated for the string, therefore, allowing the programmer to set up his formatting the way he dimensions his data variables.

Note, that all input/output instructions have been made interruptable.

4.5.1 KEYIN

KEYIN causes data to be entered into either character or numeric strings from the keyboard. A single KEYIN instruction may contain many variable names and list control items. When characters are being accepted from the keyboard, the flashing cursor is on. At all other times, the cursor is off.

When a numeric variable is encountered in a KEYIN statement, only an item of a format acceptable to the variable (not too many digits to the left or right of the decimal point and no more than one sign or decimal point) is accepted. If a character is struck that is not acceptable to the format of the numeric variable, the character is ignored and the Datapoint 2200/1100 signals a "beep". Note that if fewer than the allowable number of digits to the left or right of the decimal point are entered, the number entered will be reformatted to match the format of the variable being stored into. When the ENTER key is struck, the next item in the instruction list is processed.

When a character string variable is encountered, the system accepts any set of ASCII characters up to the limit of the physical length of the string. The formpointer of the string variable is set to one, and characters are stored consecutively starting at the physical beginning of the string. When the ENTER key is struck, the logical length is set to the last character entered, and the next item in the keyin list is processed.

Other than variable names, the KEYIN instruction may contain quoted items and list controls. Quoted items are simply displayed as they are shown in the statement. The list controls begin with an asterisk and allow such functions as cursor positioning and screen erasure. The *H<n> control causes the cursor to be positioned horizontally to the position specified by n. The *V<n> control causes the cursor to be positioned vertically to the position specified by n. The cursor may also be positioned to a specified horizontal and vertical position with the single list control, *Px:y, where 'x' gives the horizontal position and 'y' the vertical position (Note that the *Px:y list control is compatible with DATASHARE and should be used if the program is to be run under the DATASHARE Interpreter also). Position specifications in any of the cursor positioning list controls may be literals or numeric variables. The horizontal position is restricted by the interpreter to be from 1 to 80 and the vertical position is restricted to be from 1 to 12. Numbers outside of the horizontal or vertical position ranges have the effective value of 1. The *EF control erases the screen from the current cursor position, the *EL control erases the rest of the line from the current cursor position, and the *R control causes the screen to be rolled up one line. The *C control causes the cursor to be set to the beginning of the current line, and the *L control causes the cursor to be set to the following line in the current horizontal position.

Normally, the cursor is positioned to the start of the next line at the termination of a KEYIN statement. If the current line is at the bottom of the screen (line 12), the entire screen will roll-up one line. However, placement of a semicolon after the last item in the list will cause this positioning to be suppressed, allowing the line to be continued with the KEYIN or DISPLAY statement. This feature is also true of the PRINT command.

Examples:

```
KEYIN    *P1:1,*EF,"NAME: ",NAME,*H35,"ACNT NR: ":
         ACTNR, " ADDRESS: ",STREET,*P10:Y,CITY:
         *HX,*V4,"ZIP: ",ZIP;
```

While keying in a given variable, the operator may strike the BACKSPACE key and cause the last character entered to be deleted. He may also strike the CANCEL key and cause all of the characters entered for that variable to be deleted. Whenever an input from the keyboard is expected, the cursor flashes on and off. It remains off at all other times.

Two additional KEYIN list controls are provided with this version of DATABUS 2. A mode called keyin continuous is available (turned on with list control *+ and off with list control *- or the end of the statement) which causes the system to react as if an ENTER key had been struck when the operator enters the last character that will fit into a variable. This mode allows the system to react in much the same way as a keypunch machine with a control card.

The list control, *T, may also be included in the KEYIN statement causing a time out if more than two seconds elapse between the entry of two characters. The current variable is terminated as if the ENTER key was struck. Any other list controls or quoted items will be processed as usual, but all subsequent variables in the statement will be set to zero or their formpointers and logical lengths set to zero depending on whether they are numeric or string variables. Control will fall through to the next DATABUS statement. The timing routine for KEYIN time out is foreground driven and cannot be initiated until the current tape operation is complete. Therefore, a pause in execution of the KEYIN statement may occur if KEYIN time out is requested when some tape operation is still in process. When the tape I/O finishes, the keyin continuous foreground routine will be initiated and the rest of the statement executed.

4.5.2 DISPLAY

DISPLAY follows the same rules as the KEYIN except that when a variable name is encountered in the list following the instruction, the variable's contents are displayed instead of keyed in.

Character strings are displayed starting with the first physical character and continuing through the logical length. Spaces will be displayed for any character positions that exist between the logical length and physical end of the string. Numeric strings are always displayed in their entirety in both interpreters.

Examples:

```
DISPLAY  *P5:1,"RATE: ",RATE:
         *H5,*V2,"AMOUNT: ",AMNT
```

4.5.3 PRINT

The PRINT instruction causes the contents of variables in the list to be printed in a fashion similar to the way DISPLAY causes the contents of variables to be displayed. The list controls are much the same as DISPLAY except that cursor positioning cannot be used, column tabulation is provided: *<n> causes tabulation to column <n> unless that column has been passed, *F causes an advance to the top of the next form, *L causes a line feed to be printed, and *C causes a carriage return to be printed. The PRINT statement may be continued on more than one line by use of the colon.

PRINT begins printing at the first character of the string and continues through the physical end of the string. Blanks are printed for all characters after the logical end of the string. Numeric variables are printed in their entirety in both interpreters.

Examples:

```
PRINT    *20,"TRANSACTION SUMMARY",*C,*L:
         PNAME,*C,*L,*10,RATE,*20,HOURS,*30:
         AMNT,*L
```

Note, with moving head matrix printers, the continuous nature of the line printing activity precludes vertical format commands within a printed line. With these printers, all vertical format information between end-of-line commands (carriage returns) will be acted upon before the first character.

4.5.4 BEEP

The BEEP instruction causes the machine to produce an audible tone.

Example:

```
BEEP
```

4.5.5 CLICK

The CLICK instruction causes the machine to produce an audible click.

Example:

```
CLICK
```

4.5.6 DSENSE

The DSENSE instruction tests the DISPLAY key sense switch. If the DISPLAY key has been depressed, then the EQUAL condition flag is set. If the DISPLAY key is not depressed then the EQUAL condition flag is reset.

Example:

```
DSENSE
```

4.5.7 KSENSE

The KSENSE instruction acts like DSENSE except that it tests the KEYBOARD key sense switch.

Example:

```
KSENSE
```

4.6 Cassette Tape Input/Output Instructions

4.6.1 Cassette Data File Structure

The Databus cassette data file follows the standard CTOS record format (referred to as GEDIT format by the cassette editor). The most basic structure within a data file is a physical record. A physical record can contain at most 250 characters (with 6 additional bytes reserved for CTOS system usage). The next level of structuring is a logical record. Depending upon the way the user creates his cassette data file, there may or may not be an integral number of logical records within a physical record. A logical record consists of a string of data characters terminated by a 015 character after which another logical record may begin. Note that logical records can extend across physical record boundaries (<u>record compressed format</u>). A physical record is terminated by a 003 character and an inter-record gap (IRG). For example, a file with logical records may appear in the first two physical records as follows (the items in parentheses are the logical and physical record termination characters):

```
01128558382 AASDFQWERKFKDSKA (015) 1234848 (003) (IRG)
8483 LAKSJDFLKASDFKKJ (015) 48828388483 KI (003) (IRG)
```

Note that the first logical record extends about two thirds of the way through the first physical record and is then terminated by the 015 character. The first seven characters of the second logical record are also contained in the first physical record at which point the first physical record is terminated. The rest of the second logical record extends about half way through the second physical record and is then terminated by the 015 character. At this point the third logical record starts and so on.

A logical record is restricted to less than 250 characters inferring that a single logical record may not extend across more than one physical record. If one had wanted to keep only one logical record per physical record (refered to as <u>blocked record</u> or <u>write-edit</u> format), he would have made the file appear as follows:

```
     01128558382 AASDFQWERKFKDSKA (015) (003) (IRG)
     12348488483 LAKSJDFLKASDFKKJ (015) (003) (IRG)
     48828388483 KILKJLKJLKSJDFKD (015) (003) (IRG)
```

Note that it took more tape space to store the same amount of information in this case than in the previous case.

In some data files large numbers of contiguous spaces appear. These files can be compressed even further than simple use of record compression by the use of <u>space compression</u> (the general purpose CASSETTE editor, the DOS SORT program, a number of the terminal emulator programs, and other Datapoint software can generate space compressed records). A spaced compressed structure appears much like a record compressed structure except for the addition of the 011 control character. This control indicates that the next byte is a positive 8-bit binary word which tells how many spaces were replaced by the compression code character pair. This number will never be less than 2 (since it is wasteful to expand one or zero spaces into two characters) and may be as large as 255. In addition, the 011 will never appear as the last character in a physical record since the character indicating the number of spaces will always appear after the 011 (otherwise the 003 indicating the end of the physical record and three spaces compressed could not be differentiated). In the following example, a logical record is shown first without space compression and then with space compression:

```
     NOW IS THE  TIME       FOR (015)
     NOW IS THE(011)(002)TIME(011)(007)FOR (015)
```

The second record is physically shorter than the first by six characters. It may seem silly to compress two spaces into a two character compression code but most programs do this because it is logically simpler to program. If more than 255 contiguous spaces appear in the data record, multiple space compression codes will appear. Space compressed records are most useful where large numbers of spaces appear in the file and where the records are not to be modified in place. If the record is to be modified in place, record compression and space compression will be distroyed by the DATABUS 2 write instruction since the WRITE instruction maintains blocked records (one logical record per physical record and no space compression). In most cases, DATABUS 2 will be used to process blocked data files. However, this version also allows reading and processing of GEDIT files. If the user wanted to update a GEDIT file also, he could READ the file onto the scratch area on the back deck and re-write it in blocked format on the front deck. It would then be in proper format for updating.

Note, that it would be possible to overrun the tape output space if the GEDIT file was of considerable length.

DATABUS records written by DATABUS 2 do not contain anything but data characters (no 0200, ETX, or logical length and formpointer characters). Therefore, there is no indication where a variable begins or ends or of what type it was when it was written. When the records are read, the variable list of the READ statement will determine the format of the data in the record.

Reading and writing may be executed on both decks. All I/O on the rear deck (deck 1) is done to the scratch file of that tape (file 040). I/O on the front deck (deck 2) is executed on file 0 of that tape.

4.6.2 Cassette I/O Buffers

Two 256-byte internal buffers are kept by the interpreter to transfer physical records to and from each tape deck. A character pointer is kept for each buffer for scanning purposes. The buffer character pointer can have a value from 1 to 251. When a physical record is read from the indicated deck, the corresponding character pointer is set to one. The pointer is incremented as the variables in the READ statement list are filled. If the end of the buffer is reached during a READ (003 byte encountered), a new physical record will be read, and the character pointer reset to one. All cassette I/O instructions have specific effects on the character pointer which are described in the following sections.

All hardware tape operations, except for PREPARE and REWIND on deck 1, are interrupt driven. Therefore, they occur along with the background (all other processing). Programs, after issuing a tape command, proceed immediately to the next instruction until it encounters another tape command, in which case it will wait for the prior command to complete. <u>Exceptions</u>: 1) The actual reading of a physical record from a specified deck occurs in foreground. However, since READ fills variables which may appear in following instructions, control does not proceed to the next instruction until the READ statement is completed. 2) The BKSP instruction is also executed in the background. The backspace routine uses the foreground driven hardware read routine when it is necessary to read the previous physical tape record in order to complete the backspace. The BKSP routine waits until the hardware read is finished before execution of the backspace is continued. 3)

REWIND and PREPARE on deck 1 use the cassette loader to search for the scratch file (040) on that deck so could not be done in the foreground.

The WRITE operation will be write-verify (read after write) if the user has configured the interpreter to do so. Refer to Section 7 for an explanation of interpreter configuration.

4.6.3 READ

The READ command causes the next logical record to be read from the data file on the indicated tape deck (file 040 on rear deck, file 0 on the front deck). The data is entered into the variables appearing in the list following the READ instruction. A new physical record is read from the tape deck only when the end of the current buffer is reached (003 encountered). If a logical record crosses a physical record boundary, part of the variables will be filled from the current buffer and the rest from the new record read in. At the end of the READ (unless a continued READ), the character pointer will be left pointing after the logical record read (the first character following the 015). If this happens to be at the end of a physical record, the pointer will be pointing at the 003 byte; otherwise, it will be pointing at the first character of the next logical record.

A continued READ is possible in which part of a logical record may be read with one READ statement and the rest read with the next READ statement. This type of READ is indicated by a semi-colon terminating the variable list of the READ statement. After a continued READ, the character pointer will be left pointing at the character after the last one transferred.

Since there are no delimiters in a data file to distinguish the beginning and ending of a variable, the entire physical length of strings starting at the first character is written to tape. Blanks are written for all characters after the logical end of the string. When the record is read, the data is entered into the variables starting at the first position in the string and continuing to the physical end. Space compression codes are expanded. The formpointer is set to one and the logical length is set to the length of the string at the last non-blank character. If the record contains more items or characters than were in the record, the extra strings are blank filled, and the numbers are zeroed. If the variables in the READ instruction are not the same size as the variables in the WRITE instruction for that record, some of the characters may be stored into the wrong variables. However, this may be used intentionally, for example, to reformat variables when they are read.

An RFAIL trap condition will occur if there is a read failure (parity error, bad file mark, etc). The bad record is read 4 times before the condition is set. A FORMAT condition will occur if non-numeric data is read into a numeric variable.

If an end-of-file mark is read, the OVER flag is set and the tape is left positioned before the EOF mark. The remaining variables in the READ statement will be set to zero or blanks depending on whether they are numerics or strings. Note that end-of-file <u>may</u> <u>not</u> be trapped. The user must test the OVER flag after each READ statement in order to catch an end of file.

The number 1 or 2 must appear as the first item in the READ instruction list to indicate which deck is to be read (rear or front respectively).

Examples:

```
READ 1,A,B,TOTAL
GOTO EOF IF OVER

READ 2,A,B,C;
READ 2,D,E,F
```

NOTE: It is not necessary to always read every variable from a record. For example, records of six variables each were written to tape using the following write instruction.

```
WRITE 1,CODE,NAME,COMPANY,ADDRESS,SSN,POSITION
```

Another program might use the same tape, but only need the company name from each record. So this program could use the following instruction.

```
READ 1,CODE,NAME,COMPANY
```

The program may also want to read part of the record (e.g. code, name and company), do some test and either by pass the record or read the rest of that same record. In this case he would use the continued read. For example,

```
RDLP       READ 2,CODE,NAME,COMPANY;
           CMATCH "1",CODE
           GOTO CODE1 IF EQUAL
           READ 2
           GOTO RDLP
CODE1      READ 2,ADDRESS,SSN,POSITION
           .....
```

The above example would use only those records whose code equaled one. Note, how the record was by passed if CODE did not equal one.

Every variable up to and including the variables desired must be in the read statement in the order the variables appear in the records on tape. To advance the tape past a logical record, only the instruction

```
           READ 1
     or
           READ 2
```

is needed. This is particularly useful for positioning a tape to the end of file.

4.6.4 WRITE

The WRITE instruction causes a physical record to be written to the indicated deck. The record will contain the variables indicated in the list following the WRITE instruction. WRITE always writes a new physical record on the indicated deck. No record compression or space compression is used. The buffer for the indicated deck will be filled with the variables in the list beginning with buffer position one. When the WRITE is completed, the buffer pointer will be left pointing at the 003 byte of the record so that if a READ is executed next, the next physical record from the indicated tape deck will be read. If the write-verify option was chosen during interpreter configuration, the record written will be re-read to check for parity errors. Records in error will be re-read 4 times before setting the RFAIL trap condition. An EOT trap condition may also occur during execution of a WRITE statement.

If an error condition occurs which is not trapped, the program will abort (see Section 7). Aborts _do_ _not_ write end-of-file markers automatically on decks written to. Thus, if the user has issued WRITE's to a deck without an WEOF and the program aborts for some reason, the tape file structure for that deck may be destroyed. It is up to the user to TRAP these events so that condition does not occur.

The record may be any length up to 249 data characters. Since only the actual data characters are written to tape, each numeric and character string variable will have a length equal to its defined physical length. WRITE begins writing at the first character of string variables and continues to the physical end of the string. Blanks are written for all characters after the logical end of the string. Using this technique, a WRITE statement will always write the same number of characters for a variable, no matter what the logical length of the string variable.

The number 1 or 2 must appear as the first item in the WRITE instruction list to indicate which deck is to be written to (1 indicates the rear deck, 2 the front deck).

Examples:

```
     WRITE 2,TIME,TOTAL,NAME
     WRITE 1,FORM1,FORM2,FORM3
```

When using the WRITE instruction to update records of an exisiting data file, the user should be sure to rewrite records of the same length to avoid destroying the record structure of the data file.

The READ and WRITE instruction statements may be continued to the next line if a colon (:) is the terminating character of the instruction. The colon replaces the comma separating the last entry of the first line from the first entry on the second line. The first entry of the second line should begin in the instruction field.

Examples:

```
     START READ 1,NAME,POSN,ADDR,SSN,INS:
           CODE,ITEM,QUANT

     WR    WRITE 2,NAME,POSN,ADDR,SSN,INS:
           CODE,ITEM,QUANT
```

Caution: The space inefficiencies of more frequent inter-record gaps on the output write-edit format tapes which DATABUS 2 produces (when compared with blocked-compressed, GEDIT format) means a relatively full GEDIT tape may not be copied without the risk of overrunning the output tape space. That is, a file of 80 character strings typically requires only half the space in GEDIT format.

4.6.5 REWIND

The REWIND instruction list contains only a 1 or 2 to indicate the rear or front deck respectively. If the rear deck is indicated, the tape will slew to the beginning of the file area following the program library on the rear cassette. If the front deck is indicated, the cassette will be high-speed rewound to the beginning of the tape and the head positioned to the beginning of the first data record. The buffer of the indicated tape deck will be cleared (003 placed in buffer position one) with the character pointer reset to one. Thus, a new physical record from the tape deck will be read on the next READ for that deck.

NOTE

A PREPARE or REWIND instruction must be issued to deck 1 before any other tape instruction can be issued to that deck. A MODE abort will occur if another tape operation is issued first. A REWIND instruction is not necessary for deck 2, but is usually desirable. However, if two or more programs are being chained, the user may wish to have each new program continue writing to deck 2 where the previous program left off. In this case a REWIND instruction would not be desired for deck 2.

Example:

```
REWIND 1
```

4.6.6 BKSP

The BKSP instruction causes the data file on the indicated tape deck to be backspaced one logical record. The buffer character pointer is scanned backwards in the tape buffer until the end of the previous logical record is found (015 encountered). The backwards movement is continued until the beginning of that logical record is found (another 015 or file marker encountered). The buffer character pointer will be left pointing at the first character of that record. If the beginning of the physical buffer is reached during the backwards movement of the character pointer, the cassette tape deck will be backspaced and the previous physical record will be read in and the scanning continued. If the tape is at the beginning of file, the OVER flag is set and the tape is left positioned before the first data record on file. If end-of-tape is encountered during a BKSP, a TAPE error will occur and the program aborted.

A 1 or 2 must follow the BKSP instruction to indicate the rear or front deck respectively.

Example:

```
BKSP 2
GOTO BOT IF OVER
```

4.6.7 PREPARE

The PREPARE instruction list contains only a 1 or 2 to indicate the rear or front deck respectively. If the rear deck is indicated, the instruction performs the same function as REWIND. If the front deck is indicated, the cassette is rewound and a new beginning-of-file marker is written. If the interpreter was configured for write verify, the file marker will be re-read to check for errors. The tape is left positioned after the beginning-of-file marker. The buffer for the indicated tape deck is cleared (003 placed in the first buffer position) and the character pointer set to one. This ensures that a new physical record will be read if a READ instruction is the next cassette operation performed on that deck.

Example:

```
PREPARE 2
```

4.6.8 WEOF

The WEOF instruction causes an end-of-file mark to be written on the indicated deck. If the write verify option was set during interpreter configuration, the file marker will be re-read to check for errors. The tape is left positioned before the file marker. The buffer for the indicated tape deck will be cleared (003 placed in the first buffer location) and the character pointer set to one. If the next cassette operation performed on that tape deck is a READ, a new physical record will be read from the deck.

Note: End-of-file marks are <u>not</u> written automatically after a STOP instruction or some other abort of the program. The user is responsible for writing his own end-of-file marks with this statement. If he writes to a file and terminates the program without writing an end-of-file mark, the tape file structure may be destroyed.

A 1 or 2 must follow the WEOF instruction to indicate the rear or front deck respectively.

Example:

```
WEOF 1
```

CHAPTER 5. DATABUS PROGRAM AND INTERPRETER SYSTEM GENERATION

Databus source code may be generated and edited with one of the released general purpose text editors (cassette GEDIT or DOSGEDIT) using the Databus mode. In addition, text mode of the cassette editor contains an option which allows the user to generate Databus Write-Edit records which can be read by the Databus Interpreter. The source code can then be compiled into object code and executed by the interpreter from one of the system tapes.

Databus program generation and interpretive system generation may be accomplished in a variety of ways which are described below. A working knowledge of CTOS file structure and its command handler, the DOS command handler and the MINMOUT utility is required for complete understanding of the discussions following.

5.1 Cassette Program Generation

A cassette tape with the source code on it should be generated with the cassette editor, GEDIT. This tape should then be placed in the front deck and the Databus Compiler LGO tape loaded in the rear deck. When the compiler is run, the source code on the front deck will be converted into object code which is placed in file 1 after the source file on the tape in the front deck (refer to Section 6 on compiler operation).

5.2 Disk Program Generation

Databus source programs may also be created on disk with the DOS editor, DOSGEDIT. Once the file is created, it should be placed on a cassette tape. The MINMOUT disk utility will accomplish this. For example, the following would place the Databus source program named DB2TEST/TXT on the cassette in the front deck as file 0 followed by a null file 1, null file 040, and an end of file mark:

```
MOUT DB2TEST/TXT $;
```

The program may now be compiled as described in Section 5.1; or since a disk is available, the compiler can be placed on disk and run directly from there. Note that interrupts must be disabled when the compiler is run from disk. Pressing the RESTART key and reloading DOS will accomplish this. When the compiler is run from disk, it places the object code generated in file 040 after the disk boot block on the tape in the rear deck; that code is then copied to the front deck after the program source file (refer to Section 6 for compiler operating instructions).

5.3 Interpretive System Tape Generation

The Databus object code may now be placed on an interpretive system tape. This tape may be in one of two formats: CTOS multiple file format or load-and-go format.

The interpreter is released in CTOS multi-file format which consists of the following:

1. cassette loader block
2. file 0: CTOS
3. file 1: CTOS catalog
4. file 2: Databus 2 Interpreter (cataloged as DB2INT)
5. file 3: Databus 2 Master Program (cataloged as DB2MAS)

The user may add his object code program to this tape with the CTOS command handler. When the CTOS interpretive system is loaded, CTOS will come up running. The source/object tape of the Databus program should be placed in the front deck and the following command issued:

```
IN <name> or IN <file number>
```

The interpreter may then be run with the following command:

```
RUN DB2INT
```

and the program executed (refer to Section 7 for further details on interpreter operation).

A LGO interpreter system would consist of the following:

1. cassette loader block
2. file 0: Databus 2 interpreter
3. file 1: Databus Master Program
4. file 2 thru 037: object programs

A LGO interpreter system may be created from the CTOS system which has the desired program object codes cataloged on it. With CTOS running and a blank tape in the front deck, the user should issue the following command:

```
     OUT *
```

A copy of the CTOS system will be placed on the tape in the front deck leaving off files 0 and 1 (CTOS and catalog), decrementing the file numbers by two of all following named files and preserving the file numbers of files 020 through 037. This tape may then be loaded in the rear deck with the Databus 2 interpreter comming up running.

A LGO interpreter system may also be created from disk. With the MINMOUT utility, the interpreter and MASTER program may be placed on disk from an existing interpretive system tape. In addition, all desired object file programs should exist on disk. The MINMOUT utility will also accomplish this. For example, after a Databus program has been compiled, the object file can be placed on disk from the source/object tape with MIN:

```
     MIN DB2TEST/ABS
```

Then with a blank tape in the front deck, the following should be executed from disk to create the LGO interpretive system:

```
     MOUT DB2INT/ABS DB2MAS/ABS DB2TEST/ABS;L
```

Of course, more than one object program could be MOUTed at once. The interpreter is now ready to be run.

5.4 Databus Check List

The following check list may be used before compiling a program to prevent compile time errors.

Make sure:

1. Labels and variables have only six characters or less and are valid symbols.
2. There are not too many labels or variables in the program.
3. All labels and variables are defined, but not doubly defined. (Two labels or two variables must not have the

same name).

4. All common variables are defined in exactly the same order and length as the variables in the other programs.
5. All instructions are spelled correctly.
6. There are no unmatched quote signs and no cursor positions off the screen.
7. The program does not exceed the allotted user space.

CHAPTER 6. DATABUS COMPILER OPERATION

The Databus Compiler generates Databus object code from Databus source programs which can then be executed by the Databus Interpreter. Each object program should be filed on the interpreter system tape so it can be run any number of times without being recompiled. See Section 7 for instructions on creating interpreter system tapes (LGO or CTOS).

The compiler makes one pass over the symbolic source code. All statements are checked for syntax and form. If any errors are found, flags are given. As the program is compiled, a program listing and an object program on tape are generated. If any errors occur, a flag will be set in the resulting object code so that the interpreter will not attempt to run the program.

The compiler assigns numeric values to the various instructions and operands. Each instruction mnemonic has an octal value assigned to it as do the various conditions, events, units, variables and labels.

Two symbol tables are generated by the compiler, one for variables and one for labels. An 8 bit object code value is assigned to each variable and label encountered. The high order bit determines in which table the entry can be found - 1 indicating the variable table and 0 the label table. The low order seven bits determine the position of the symbol in the table. Each table entry consists of 8 bytes, 6 for the symbol and 2 for the address. The last two bytes of each entry are output by the compiler as part of the object code, forming lookup tables used by the interpreter to locate the labels and variables mentioned above. Each table is limited to 112 entries each.

All variables are defined by directives, that is they must appear in the label field of directive instructions. Any symbols which appear in the label field of executable instructions are placed in the label table. All directives must appear before the first executable instruction in the program. Any directives which appear after the first executable instruction are given I-flags and their labels are placed in the label table instead of the variable table. Therefore, any references to these symbols will be flagged undefined.

In short, variables cannot be forward referenced, but labels can. Since the compiler makes only one pass over the source code, all labels are entered into the label table when found in the label or operand field of an instruction. No U-flags are given for undefined labels until the end of compilation when the symbol tables are output as part of the object code.

All undefined variables are entered into the variable table and flagged at the end so that the symbol tables output at the end of the listing will show all undefined symbols.

The following errors can occur during compilation:

1. D The D flag means DOUBLE DEFINITION. It is flagged if a label or variable has been defined to more than one value during compilation. In that case, it has the first value.

2. I The I flag means INSTRUCTION MNEMONIC UNKNOWN. The instruction was not an acceptable instruction code. In this case a 345 is inserted for the instruction.

3. E The E flag means that an error has occurred in the operand field of a statement or some unrecognizable character appeared in the wrong place. In this case a zero is substituted for the operand or whatever was unrecognizable.

4. U The U flag means UNDEFINED SYMBOL. It is used whenever a symbol is referenced and is not defined.

OVERFLOW - This message is given if the user program exceeds its allotted space.

DICTIONARY FULL - This message is given if the user program has too many labels or variables.

Operating the Compiler:

The compiler may be run from a cassette LGO tape or directly from disk. Interrupts must be disabled before the program will run from disk; if they are not, it will hang or halt. The compiler determines whether it is being run from cassette or disk. If it is being run from cassette, it uses the cassette loader in low memory to position to file 040 on the rear deck (this is where the object code is placed as it is generated). If it is being run from disk, it assumes the tape in the rear deck is a DOS boot tape and is positioned after the boot block. The compiler merely writes a file 040 and begins compilation. Note, of course, if run from disk, the user is not allowed to configure the program tape, see question 1) below.

Place a DATABUS symbolic source tape generated by the Editor in the front deck.

Run the Compiler. When it is loaded, the Compiler version will be displayed and the following configuration options will be requested:

1) CONFIGURE PROGRAM TAPE?

This option is given only if the compiler is run from a LGO cassette tape. Answer 'Y' if the responses to the remaining questions (except HEADING) are to be recorded on the compiler tape. The last block of code of the compiler is used to record those responses. If the tape is configured, the compiler will us those answers for configuration instead of asking the questions again when the compiler is reloaded. Any option may be left open so that it will always be requested during compiler initialization; this is accomplished by answering '#' when that option is given. The '#' reply is only acceptable when configuration of the program tape has been requested. The recorded configuration may be overridden if the KEYBOARD key is depressed when the compiler is loaded. In that case, all configuration questions will be asked. If 'N' is answered to 1) above, the configuration responses will not be recorded on the compiler tape.

2) OBJECT MACHINE SIZE (8,12,16)?

Type in the size of the machine in which the interpreter and user program will be run. This will define the user area.

3) PRINT?

Answer Y if a hard copy listing is desired; otherwise, type N.

4) LOCAL OR SERVO PRINTER?

Answer L for Local Printer or S for Servo Printer. Answer either if no printer is available. This option is only given if YES is answered after 3) or the tape is to be configured and '#' was replied after 3).

5) DISPLAY?

Answer Y if a CRT display of the compilation results is desired; otherwise answer N.

6) CODE?

Answer Y if the object code is desired in the listing or display; otherwise answer N. This option is asked only if YES was replied after 3) or 5) or the tape is to be configured and '#' was answered after 3) or 5). Code adds 18 columns to the listing.

7) HEADING:

Type in the heading. This option is given only when a listing is desired (answer to 3) was YES). The heading is not configurable.

When all questions have been asked, those which were answered with a '#' will be re-asked and the configuration block written on the program tape if requested. The interpreter object machine size chosen will then be displayed and the source tape in the front deck will be rewound and compiled. At the end of compilation the object code generated on the rear tape is copied to the front deck after the source code (file 1). DONE will appear when all copying is complete and the tape in the rear deck will be rewound and loaded. Therefore, if the compiler was run from a LGO cassette, it will be re-loaded. If it was run from disk, the user will be returned to the Disk Operating System.

CHAPTER 7. RUNTIME OPERATION

The Databus 2 Interpreter may be run from a LGO system tape or a CTOS system tape. If the program tape is LGO, the first block on tape should be the cassette loader. The Interpreter should be file 0 and the MASTER Databus program code file 1 followed by any other program object files (DATABUS or other) to be executed by the interpreter (files 2 through 037). When the LGO tape is loaded, the interpreter will come up running. If the program tape is CTOS, the cassette loader should be followed by CTOS as file 0, the CTOS catalog as file 1, the Databus interpreter as file 2, the MASTER Databus object file as file 3, and any other program object files (through file 037). CTOS will come up running when the sytem tape is loaded; the interpreter may then be run with the following command:

```
RUN DB2INT
```

Note that the interpreter must know what type of system tape it is being run from in order to know what file number the MASTER program is (file 1 or file 3). This information is given to the interpreter during configuration.

7.1 Interpreter Configuration

When the interpreter is run, the following configuration questions will be asked:

1) CONFIGURE PROGRAM TAPE?

Answer 'Y' if the responses to the remaining questions are to be recorded on the interpreter tape. The last block of code of the interpreter is used to record those responses. If the tape is configured, the interpreter will use those answers for configuration instead of asking the questions again when the interpreter is reloaded. Any option may be left open so that it will always be requested during interpreter initialization; this is accomplished by answering '#' when that option is given. The '#' reply is only acceptable when configuration of the program tape has been requested. The recorded configuration may be overridden if the KEYBOARD key is depressed when the interpreter is loaded. In that case, all configuration questions will be asked. If 'N' is answered to 1) above, the configuration responses will not be recorded on the interpreter tape.

2) INTERPRETER TAPE LGO OR CTOS?

Answer L if system tape is LGO or C if it is CTOS. This tells the interpreter what file to run as the MASTER program (file 1 or 3 respectively).

3) LOCAL OR SERVO PRINTER?

Answer L for local printer or S for Servo printer. Answer either if no printer is available.

4) WRITE VERIFY?

Answer 'Y' if read after write is desired on all WRITE operations; otherwise, answer 'N'.

When all questions have been answered, those answered with '#' will be re-asked and the configuration block on the program tape will be written if requested.

7.2 MASTER Program Operation

After initialization, the interpreter will run the MASTER program. The MASTER program is file 1 on LGO interpretive system tapes and file 3 on CTOS sytem tapes. This may be the released MASTER program or any other Databus program the user wishes to put in its place. The MASTER program released with this version of DATABUS allows the user to specify a program on the system tape to be run or execute a variety of tape handling functions. When MASTER is run, the message,

```
UTILITY MASTER - DATABUS 2 RELEASE 5.1
READY
```

will appear with the cursor flashing under the READY message. The program expects either the file number of a program to be run or a special instruction character. If the input is not a valid entry,

```
WHAT?
```

will be displayed and a beep sounded to indicate the error. New input may then be entered.

The user may chain to program file numbers 1 (or 3 if CTOS system) through 040. The file number entered must be octal. Typing a file number not on the system tape will cause the CFAIL abort message to be displayed and the MASTER program will be reloaded. Typing the file number of a non-Databus program will cause it to be loaded and executed if it does not overlay the first 82 bytes of the main execution loop of the interpreter (located at label START in the interpreter code). Overlay of these locations will usually cause complete confusion. Typing the file number of a Databus program will cause it to be loaded and executed unless the compiler generated some error messages, in which case an error abort will be made.

The tape handling functions may be requested with the following special characters. All codes operate on the front deck unless otherwise indicated. An 80-character variable is maintained by the program to allow the user to keyin a line and write it to the front deck. This same variable is used to read records from the front deck. Records greater than 80 characters are truncated; records less than 80 are blank filled to the right. When an end-of-file mark is encountered during some tape operation, that operation will be stopped and the message,

```
* EOF *
```

followed by the READY message will be displayed. If the beginning of file marker is encountered during a backspace,

```
* BOF *
```

will be displayed followed by the READY message.

>	Rewind the front tape.
<	Prepare the front tape.
L	List the file on the front deck to the screen. The tape is not rewound before the read, making it possible to begin listing anywhere on the tape. If an end of file is encountered, the list stops and the message READY appears. The KEYBOARD key may be depressed to stop the listing before the end of file is reached; the DISPLAY key may be depressed to cause a pause in the listing until the key is released.
P	List the file on the printer. Listing begins from the current position of the tape in the front deck. Lines are numbered beginning at 1. Fifty lines are printed per page. The listing will terminate if the end of file is encountered or the KEYBOARD key is depressed. The DISPLAY key may be depressed to cause a pause in the

	listing until the key is released.
B	Backspace one record on the front deck.
E	Write an end-of-file mark on the front deck.
R	Read a record from the front deck and display it on the screen.
W	Write the current 80 character record to the front deck.
K	Keyin an 80 character record. When 'K' is entered the screen will roll up and await input of the 80 character record.
/	Copy the file from the front deck to the back deck. Both files are rewound and the rear deck is prepared. Copying takes place until an end-of-file mark is reached on the front deck. An end-of-file mark is written to the rear deck and both decks are left positioned before their respective end-of-file marks. The message READY will appear when copying is completed.
\	Copy the file from the rear deck to the front deck. Both files are rewound and the front deck is prepared. This copy works similarly to the '/' copy command with the decks reversed.
M	Enter and write multiple records. Records are entered and written to the front deck. No rewind takes place before the entering, permitting additions to be made at the end of the tape, or group modifications to be made in the middle of the file. To return to READY, press the KEYBOARD key along with the ENTER key after typing the record. No end-of-file mark is written on termination, and no rewind takes place.

7.3 Program Termination

Once a program is running, execution may be terminated for a number of reasons. Execution of a STOP statement is equivalent to a CHAIN to the MASTER program. Pressing of the KEYBOARD & DISPLAY keys simultaneously will also cause the program to abort and chain to MASTER after completion of the current statement. All other terminations will first print an error message of the format:

```
(error message)  AT nnnnn
```

nnnnn will be the statement number (number that appears to the left of the statement on the compiler listing) on the statement _after_ the one which is at fault. After this message is displayed, a CHAIN to the MASTER program will be performed. Aborts from tape I/O trap conditions may not occur until the next tape I/O operation, CHAIN statement, KEYIN statement with the *T list control, or STOP instruction. In those cases, the nnnnn statement number will indicate the following instruction which caught the error instead of the one that caused the error.

A list of the error messages and their meanings follows. Those which are followed by an asterisk can be trapped by the Databus program (refer to Section 4.2.6).

CODE	An attempt was made to run an object file which was generated from a source file that the compiler found at fault.
ABORT	Both the KEYBOARD and DISPLAY keys were depressed. The statement before nnnnn was the last one executed.
BOP	An undefined operation code was found at location nnnnn. This can happen only if there is a software error in the DATABUS compiler or interpreter system, if there is a hardware error, or if the interpreter has been destroyed by a non-DATABUS program.
MODE	A tape I/O operation other than REWIND or PREPARE before statement nnnn was issued to deck 1 before an initial REWIND or PREPARE.
TAPE	A PREPARE or REWIND on deck 1 failed, or EOT occurred during a BKSP command. The I/O statement was before statment nnnnn.
BUFUL	During the tape write I/O operation before statement nnnnn, more than 249 bytes were written to cassette tape. The tape write will not occur.
EOT*	An end-of-tape condition arose during the tape I/O operation before statement nnnnn and the trap was not set.

FORMAT*	During the tape read before statement nnnnn, an item of string type was read into a numeric variable.
RFAIL*	During a tape read or write-verify before statement, nnnnn, a read failure occurred (parity error, file marker error, etc). Note that if the RFAIL is not trapped and WRITE's have been issued without a WEOF, tape file structure may be destroyed.
CFAIL*	An invalid file number was requested in the CHAIN instruction before statement nnnn (this condition is the only one that can be trapped); or execution of the CHAIN statement failed to find the requested file number on the interpreter tape; or a loader failure occurred.

CHAPTER 8. CHAINING TO NON-DATABUS PROGRAMS

DATABUS 2 uses the cassette loader to perform the actual loading function. After loading, Databus checks the user program starting location in RUN$ of the Loader. The compiler always generates an object file with the same starting location. The Interpreter then assumes that if the object file just loaded has this starting location, then the object file must be Databus object code. If the starting location was something different, Databus simply jumps to RUN$. Therefore, to CHAIN to a non-Databus program, make a CHAIN to that program, providing that its starting location is not the Databus user program starting location, and that it does not overlay the routine residing in the first 82 bytes (START thru START+82) of the Interpreter. This is the section which checks the user program starting location.

Example:

```
NONDAT INIT "NONDAT"
       CHAIN NONDAT
```

If the non-Databus program resides within the Databus user area without overlaying any part of the interpreter, then it can chain back to a Databus program by supplying the file number to the interpreter so the program will be loaded. This may be accomplished by loading the B-register with the program file number and jumping to CHAINX (02131).

Example:

```
              SET    017000
CHAINX        EQU    02131
.
. program follows
.
END           LB     3
              JMP    CHAINX
```

If the non-Databus program does not reside in the Databus user area but has overlayed part of the interpreter, then it must reload the Databus Interpreter and jump to START (which will cause the MASTER program to be executed).

If the non-Databus program is outside of the interpreter and user area, the program may merely return to START or supply a new program file number to be loaded.

The DATABUS 2 user program starting location is 017000. The START label in the interpreter code is located at 02101 and the CHAINX label is located at 02131.

Caution: Since this version of DATABUS 2 uses the interrupt feature of the DATAPOINT 1100 and the Version II 2200, any non-Databus programs chained to must be interruptable.

CHAPTER 9. INTERPRETER INTERNAL OPERATION

The interpreter fetches and executes instructions (statements) much like a computer. It contains within its workinç storage area the equivalent of the program counter, condition register, instruction register, and other miscellaneous items. The basic instruction format is one byte broken into two fields:

```
N   N   0   0   0   0   0   0
```

The NN bits indicate the number of bytes in the instruction. For I/O operations, this number is either one or two and the rest of the instruction is read by scanning for the list terminator. This number is never zero.

The 000000 bits indicate which operation is to be performed. This number provides an index into an address table which causes the interpreter to execute the proper subroutine to perform that instruction.

Operands and labels are addressed by single bytes. Labels have their sign bit clear and operands have them set. The remaining seven bit numbers index into address tables (one for labels and a different one for operands) which are generated by the compiler at the end of compilation. Because of this, the compiler only needs to be a one pass process. Since these tables are placed after the user's code, they may be located anywhere, s the compiler cranks out two other addresses in the interpreter working storage area which point to the beginning of each table. Thus, a typical instruction execution sequence would be as follows:

a) Get the byte pointed to by the PC and increment the PC.

b) Get the operand pointed to by the PC and increment the PC.

c) Branch to the correct routine based on the value of the right six bits of the opcode. The correct address is obtained by multiplying the right six bits by two and adding the result to the execution routine address table Load the address of the routine from the table and jump to it.

d) The instruction would take the operand number, isolate the right seven bits, multiply it by 2, add it to the base adddress of the operand table, load the address of the variable or label from the table and perform some operation upon the variable or label pointed to.

In DISPLAY, KEYIN, and PRINT, immediate characters (quoted items) are denoted by not having their sign bit set. These characters are simply printed unless they have a special control function for the instruction in which they appear. The controls fall in the group between 0 and 37 octal.

CHAPTER 10. DATABUS 2 SUMMARY

10.1 Databus Definitions

address	Refers to the location in memory of assembly language subprogram to be called. May be octal or decimal.
blocked record	A type of record format on cassette tape. One logical record is placed per physical record and no space compression is used. Also refered to as write-edit format.
buffer character pointer	The pointer maintained for each buffer used to transfer data to and from the two cassette decks. The pointer indicates the current character position of the physical record read from the indicated deck.
compressed record	A type of record format on cassette tape. Logical records are compressed so that each physical record is completely filled (all 249 data characters used). This results in logical records crossing physical record boundaries. Space compression may also appear in record compressed files.
character string	Any string of alphanumeric characters.
condition	The result of operations used in conditional transfer of control operations. LESS,EQUAL,ZERO,OVER: arithmetic operation result LESS,EQUAL,ZERO,EOS: string operation result OVER: READ or BKSP operation result

event	The occurrence of end of tape, tape read error, data type error, or program chain failure.
	
	EOT(unit)
	RFAIL(unit)
	FORM(unit)
	CFAIL

label	A name assigned to a statement.
list	A list of variables, quoted character strings, or controls appearing in an input/output type of instruction.
literal	A quoted alphanumeric character or a number. The number may be octal or decimal as long as it is between 0 and 127 decimal.
n	Refers to an integer between 0 and 127 decimal.
n.m	Refers to any number octal or decimal. Octal if it is preceded by a zero, up to 22 total digits including the decimal point.
nvar	A label assigned to a directive defining a numeric string variable.
size	A number defining the memory size of the Datapoint 2200 in which the user program and interpreter will be run. It may be 8, 12, or 16.
space compression	A compression technique used to make cassette records shorter. Two or more contiguous spaces are reduced to two bytes. The first byte is always a 011 to indicate a space compression byte follows. The second byte is a positive 8-bit number indicating the number of spaces compressed; this byte may be from 1 to 255.
sval	A label assigned to a directive defining a character string variable, or a quoted alphanumeric character, or a number. This number may be octal or decimal as long as it is between 0 and 127 decimal.
svar	A label assigned to a directive defining a character string variable.

unit	A number defining a tape deck. 1 = Deck 1 (rear) 2 = Deck 2 (front)
456.23	Refers to any octal or decimal number, up to 22 total digits.

10.2 Databus Input/Output Controls

CONTROL	APPLICABLE INSTRUCTION	FUNCTION
*Hn	KEYIN DISPLAY	Causes cursor to be positioned horizontally to the column indicated by the literal or numeric variable n, 1<n<80.
*Vn	KEYIN DISPLAY	Causes the cursor to be positioned vertically to the row indicated by the literal or numeric variable n, 1<n<12.
*Px:y	KEYIN DISPLAY	Causes the cursor to be positioned to the horizontal and vertical position indicated by 'x' and 'y' respectively. Both 'x' and 'y' may be either a literal or numeric variable with 1<x <80 and 1<y <12.
*EL	KEYIN DISPLAY	Causes the c.r.t. screen to be erased from the current cursor position to the end of the line.
*EF	KEYIN DISPLAY	Causes the c.r.t. screen to be erased from the cursor position to the end of the screen.
*R	KEYIN DISPLAY	Causes the c.r.t. screen to roll up one line, losing the top line and setting the bottom line to blanks. (The cursor position does not move.)
*+	KEYIN	Initiates keyin continuous mode.
*-	KEYIN	Turns off keyin continuous mode.
*T	KEYIN	Time out after 2 seconds for KEYIN statement.
*n	PRINT	Causes horizontal tab to the column indicated by the number n. (No action occurs on the local printer if the carriage is past the column indicated by n.)

;	KEYIN DISPLAY PRINT READ	In KEYIN, DISPLAY, and PRINT, suppresses a a new line function when occuring at the end of a list, i.e., the cursor or print carrriage remains in the position indicated by the completion of the last list element. In READ, causes the buffer character pointer to remain pointing at the character in the buffer after the last transfered during the READ.
"	KEYIN DISPLAY PRINT	Any characters appearing between quotes are displayed or printed when encountered.
*F	PRINT	Causes the printer to be positioned to the top of form.
*L	KEYIN DISPLAY PRINT	Causes a linefeed to be printed.
*C	KEYIN DISPLAY PRINT	Causes a carriage return to be printed.

10.3 Program Length

a) Numeric String Variables use two words plus one word for each string character (including decimal point and sign if negative).

b) Character String Variables use three words plus one word for each string character.

c) String Instructions except LOAD and STORE use two or three words depending on whether one or two variable names are required for the instruction.

d) Arithmetic Instructions except LOAD and STORE use three words. LOAD and STORE fall into the Control category for space allocation.

e) Control and Input/Output Instructions require one word for the command plus one word for each label, condition, event, variable, or unit used. Strings found in I/O instructions add one word per character. I/O controls which begin with an asterisk add one, two, or four words for each occurrence (*C, *L, *F, *EL, *EF, *R, *+, *-, *T use one word, *Px:y uses four words, and all others use two). Every instruction which contains a list uses one additional word for the list terminator.

f) Two additional words are used for each label or variable.

10.4 Language Summary

10.4.1 Instructions

```
Directives
   FORM n.m
   FORM "456.23"
   DIM n
   INIT "character string"
   FORM *n.m
   FORM "456.23"
   DIM *n
   INIT *"character string"
Control
  TRAP (label) IF (event)
  TRAPCLR (event)
  GOTO (label)
  GOTO (label) IF (condition)
  GOTO (label) IF NOT (condition)
  CALL (label)
  CALL (label) IF (condition)
  CALL (label) IF NOT (condition)
  RETURN
  RETURN IF (condition)
  RETURN IF NOT (condition)
  STOP
  STOP IF (condition)
  STOP IF NOT (condition)
  CHAIN (svar)
  BRANCH (nvar) OF (label list)

String
  CMATCH (sval) TO (sval)
  CMOVE (sval) TO (svar)
  MATCH (svar) TO (svar)
  MOVE (svar) TO (svar)
  MOVE (svar) TO (nvar)
  MOVE (nvar) TO (svar)
  APPEND (svar) TO (svar)
  RESET (svar) TO (sval)
  RESET (svar) to (nvar)
  RESET (svar)
  BUMP (svar) by (literal)
  BUMP (svar)
  ENDSET (svar)
  LENSET (svar)
  TYPE (svar)
  EXTEND (svar)
  CLEAR (svar)
  LOAD (svar) FROM (nvar) OF (svar list)
  STORE (svar) INTO (nvar) OF (svar list)

Numeric Variable Arithmetic
  ADD (nvar) TO (nvar)
  SUB (nvar) FROM (nvar)
  MULT (nvar) BY (nvar)
  DIV (nvar) INTO (nvar)
  MOVE (nvar) TO (nvar)
  COMPARE (nvar) TO (nvar)
  LOAD (nvar) FROM (nvar) OF (nvar list)
  STORE (nvar) INTO (nvar) OF (nvar list)

Keyboard, C.R.T., Printer I/O
  KEYIN (list)
  KEYIN (list);
  DISPLAY (list)
  DISPLAY (list);
  PRINT (list)
  PRINT (list);
  BEEP
  CLICK
  DSENSE
  KSENSE

Cassette Tape I/O
  READ (unit),(list)
  READ (unit),(list);
  READ (unit)
  WRITE (unit),(list)
  REWIND (unit)
  BKSP (unit)
  PREPARE (unit)
  WEOF (unit)
```

10.4.2 Conditions

OVER
LESS
EQUAL
ZERO
EOS

10.4.3 Events

EOT1
EOT2
RFAIL1
RFAIL2
FORM1
FORM2
CFAIL

10.5 User Area

<u>Interpreter Machine</u>

```
 8K -  01000  bytes (017000 - 017777)
12K - 011000  bytes (017000 - 027777)
16K - 021000  bytes (017000 - 037777)
```

10.6 Dictionaries

<u>Compiler Machine</u>
112 labels
112 variables

10.7 Interpreter Internal Structure

The Databus 2 Interpreter is layed out in memory as follows. Note, that at load time, the local printer driver is located in the cassette buffer for deck two. If the interpreter is configured to use the local printer, this code will be moved into the actual printer driver area. The configuration routine is located part in the user program area, part in I/O buffer 1 and 2 and part in the working storage page since it is only needed during interpreter initialization.

```
                                      37777
USER AREA
16K
                                      27777
USER AREA
12K & 16K
                                      17777
USER AREA
8K
                                      17000
WORKING STORAGE
                                      16400
NUMERIC OPERATIONS
                                      13557
PRINTER DRIVER
                                      13027
MAIN INTERPRETER BLOCK
                                      2000
CASSETTE I/O BUFFER
DECK TWO
                                      1400
CASSETTE I/O BUFFER
DECK ONE
                                      1000
LOADER
                                      0
```

10.8 Compiler Internal Structure

The Databus 2 Compiler is layed out in memory as follows. The compiler configuration routine is located part in the operand dictionary and part in the I/O buffers since it is only needed during initialization. The routines to process PSEDUO op codes (DIM, INIT, and FORM) is located in the label dictionary since it is not needed after the first executable statement is reached.

```
                              17777
LABEL DICTIONARY
                              16160
OPERAND DICTIONARY
                              14350
MAIN COMPILER
                              2000
OUTPUT BUFFER
                              1400
INPUT BUFFER
                              1000
CASSETTE LOADER
                              0
```

10.9 Sample Programs

The sample Databus 2 programs included make up a simple file handling system. It is by no means complete, but serves to give an idea of what can be done with Databus 2. Note that a general beginning address was given for the examples (01000). For this version of Databus 2, the user program area begins at 017000.

A brief summary of each program will be given to aid in tracing through the programs. These programs include an update file entry program, a two tape merge of the update file into the master file, as well as programs to display and copy the two files.

UPDATE PROGRAM

1. Positions rear deck to Update File.
2. Allows user to type in the 5 fields of information for the update records.
3. As each field is input, it is appended to a buffer. Slashes are used as field delimiters.
4. Writes out the packed record to the update file.
5. If more update files need to be input goes to 2.
6. Otherwise writes a dummy record to indicate the end of the update file, and a physical end of file.
7. Chains back to the MASTER program.

UPDATE FILE DISPLAY PROGRAM

1. Reads update file records from the rear deck.
2. Displays each record exactly as it was written to tape, rolling up the screen as each entry is displayed.
3. When all records have been displayed, execution returns to the MASTER program.

TWO TAPE MERGE PROGRAM

1. Asks if front tape is a new master tape.
2. If the master is new, the front deck is prepped. The rest of the program treats the front deck the same whether it is an old or new master.
3. The rear deck is positioned to the update file.
4. Five records are read from the update file. The records were written with the name field first, and the merge is done in alphabetical name order.

5. The rear deck is then positioned to the end of the update file.
6. The smallest of the five update records is found.
7. The smallest record is then merged into the master tape. This is done in the following manner:

    a. The master tape records are read in one at a time.
    b. The master record is compared to the update record. If the master record is smaller it is written to the update tape (now positioned after the end of the update file), and a new master record is read in and compared.
    c. If the update record is smaller, it is written to the update tape.
    d. Execution then returns to 6 where the next smallest update record is found until all 5 update records have been merged.

8. Once all 5 are merged, the rest of the master tape is copied to the rear deck.
9. The rear new master is copied back to the front tape.
10. The update file is then positioned after the last five update records are read in, and then five more records are read (or as many as are left).
11. Execution then returns to 5.
12. Once all update records are merged and the final master copied back to the front tape, the tapes are rewound, and execution returns to the MASTER program.

MASTER FILE DISPLAY PROGRAM

1. Rewinds the front master tape.
2. Reads in a record.
3. Unpacks the record into five fields. The unpacking is done by a character match, searching for a slash, the field delimiter.
4. When all fields are unpacked, the information is displayed on the screen.
5. Execution then returns to 2 until all records have been read and displayed.
6. Execution returns to the MASTER program.

COPY PROGRAM

1. Copies records (maximum of 127 chars) from front deck to rear, and rear deck to front. The records are written to file 32 on the rear deck, and file 0 on the front deck.
2. When all records have been copied, execution returns to the MASTER program.

MASTER FILE DISPLAY PROGRAM

```
         . PROGRAM LIST
         .
         . DISPLAYS MASTER FILE
         . READS IN RECORD FROM TAPE, UNPACKS THE
         . DATA INTO FIVE FIELDS, AND DISPLAYS THE
         . FIELDS ON THE SCREEN.
01000    NAME       DIM 20
01027    ADDR       DIM 30
01070    SSN        DIM 11
01106    BUSNES     DIM 20
01135    CCODE      DIM 4
01145    SLASH      INIT "/"
01150    BUFF       DIM 89
01304    TEMP       FORM 30
01345    COUNT      FORM "01"
01351    ONE        FORM "1"
01354    SIX        FORM "6"
01357    EXIT       INIT "001    "
01370    START      DISPLAY *P1:1,*EF,*H15,"MASTER FILE DISPLAY"
01424               DISPLAY *P1:2,"FRONT TAPE MASTER?";
01454               KEYIN TEMP
01457               REWIND 2
01461    RD         READ 2,BUFF
01465               GOTO END IF OVER
01470               MOVE ONE,COUNT
01473               CLEAR TEMP
01475    LOOP       CMATCH BUFF,SLASH
01500               GOTO NEXT IF EQUAL
01503               EXTEND TEMP
01505               GOTO NEXT IF EOS
01510               CMOVE BUFF,TEMP
01513               BUMP BUFF
01515               GOTO NEXT IF EOS
01520               GOTO LOOP
01522    NEXT       RESET TEMP
01525               STORE TEMP INTO COUNT OF NAME,ADDR,SSN:
01533               BUSNES,CODE
01536               ADD ONE,COUNT
01541               COMPARE COUNT,SIX
01544               GOTO DISPLY IF EQUAL
01547               CLEAR TEMP
01551               BUMP BUFF
01553               GOTO LOOP IF NOT EOS
         .
         . DISPLAY ALL FIVE FIELDS ON SCREEN
         .
01556    DISPLY     DISPLAY *H1,*V5,*EF,"NAME:",NAME
```

```
01574               DISPLAY *H1,*V6,"ADDRESS:",ADDR
01614               DISPLAY *H1,*V7,"SOCIAL SECURITY #:",SSN
01646               DISPLAY *H1,*V8,"COMPANY:",BUSNES
01666               DISPLAY *H1,*V9,"CUSTOMER CODE:",CCODE
01714               CLEAR NAME
01716               CLEAR ADDR
01720               CLEAR SSN
01722               CLEAR BUSNES
01724               CLEAR CCODE
01726               GOTO RD
01730   END         CHAIN EXIT
01732               STOP

01733   START
01735   END
01737   RD
01741   LOOP
01743   NEXT
01745   DISPLY

01747   NAME
01751   ADDR
01753   SSN
01755   BUSNES
01757   CCODE
01761   SLASH
01763   BUFF
01765   TEMP
01767   COUNT
01771   ONE
01773   SIX
01775   EXIT
```

DATABUS TWO TAPE MERGE PROGRAM

```
              . PROGRAM DATABUS SORT PROGRAM
              .
              . MERGE PROGRAM
              .
              . READS IN UPDATE TAPE ON REAR DECK 5 RECORDS AT
              . A TIME AND MERGES THEM INTO MASTER ON FRONT DECK.
              . IF THE MASTER TAPE IS NEW, THE UPDATE TAPE IS
              . SORTED AND WRITTEN TO THE MASTER.
              .
      01000   N1          DIM 89
      01134   N2          DIM 89
      01270   N3          DIM 89
      01424   N4          DIM 89
      01560   N5          DIM 89
      01714   N6          DIM 89
      02050   MASTER      DIM 89
      02204   TST         DIM 89
      02240   NL          INIT "^^^^^^^^^^^^^^^^^^^^"
      02367   DUMMY       INIT "**WEOF**"
      02402   EXIT        INIT "001    "
      02413   FLAG        FORM "0"
      02416   COUNT       FORM "01"
      02422   SMALL       FORM "0"
      02425   CNTSV       FORM "00"
      02431   ONE         FORM "1"
      02434   ZERO        FORM "0"
      02437   TEMP        DIM 2
      02444   RECORD      FORM "0000"
      02452   CNT         FORM "00"
      02456   SIX         FORM "6"
      02461   TEN         FORM "10"
              .
      02465   START       DISPLAY *P1:1,*EF,*H15,"TWO TAPE MERGE PROGRAM"
      02524               DISPLAY *H1,*V3,"READS IN UPDATE TAPE ";
      02557               DISPLAY "AND MERGES IT INTO MASTER TAPE ";
      02617               DISPLAY "IN ALPHABETICAL ORDER"
      02646   ASK         DISPLAY *H1,*V4,*EL,"NEW MASTER?";
      02670               KEYIN TEMP
      02673               REWIND 1
      02675               CMATCH TEMP,"Y"
      02700               GOTO REWD IF NOT EQUAL
      02703               PREPARE 2
      02705               WEOF 2
      02707   REWD        REWIND 2
              .
      02711               SORT MOVE NL,N5
      02714   RD          READ 1,TST
```

```
02720            MATCH DUMMY,TST
02723            GOTO SETFLG IF EQUAL
02726            STORE TST INTO COUNT OF N1,N2,N3,N4,N5
02737            ADD ONE,COUNT
02742            COMPARE SIX,COUNT
02745            GOTO RD IF LESS
      .
02750  CNT       COMPARE COUNT,ONE
02753            GOTO END IF EQUAL
02756            MOVE COUNT,CNTSV
02761            SUB ONE,CNTSV
      .
02764  FEOF      READ 1,MASTER
02770            MATCH MASTER,DUMMY
02773            GOTO FEOF IF NOT EQUAL
02776            CLEAR MASTER
      .
03000  M1        MOVE ONE,COUNT
03003  FIND      LOAD TST FROM COUNT OF N1,N2,N3,N4,N5
03014            MATCH N5,TST
03017            CALL MOVE
03021            ADD ONE,COUNT
03024            COMPARE CNTSV,COUNT
03027            GOTO FIND IF LESS
03032            GOTO FIND IF EQUAL
      .
03035            RESET MASTER
03040            GOTO MRG IF NOT EOS
03043  MERGE     READ 2,MASTER
03047            GOTO MOVSTR IF OVER
03052  MRG       MATCH N5,MASTER
03055            GOTO MOVSTR IF NOT LESS
03060            WRITE 1,MASTER
03064            GOTO MERGE
      .
03066  MOVSTR    WRITE 1,N5
03072            MOVE NL,N5
03075            STORE NL INTO SMALL
03106            ADD ONE,CNT
03111            COMPARE CNTSV,CNT
03117            MOVE ZERO,CNT
03122            RESET MASTER
03125            GOTO TRNSFR IF EOS
03130            WRITE 1,MASTER
03134  TRNSFR    READ 2,MASTER
03140            GOTO COPY IF OVER
03143            WRITE 1,MASTER
```

```
03147                GOTO TRNSFR
         .
03151    COPY        WEOF 1
03153                REWIND 1
03155                PREPARE 2
         .
03157    SRCH        READ 1,MASTER
03163                MATCH MASTER,DUMMY
03166                GOTO SRCH IF NOT EQUAL
         .
03171    RDWR        READ 1,MASTER
03175                GOTO SETUP IF OVER
03200                WRITE 2,MASTER
03204                GOTO RDWR
         .
03206    SETUP       WEOF 2
03201                REWIND 2
03212                REWIND 1
03214                COMPARE FLAG,ONE
03217                GOTO END IF EQUAL
03222                ADD CNTSV,RECORD
03225                MOVE ZERO,COUNT
03230    RECRD       READ 1,MASTER
03234                ADD ONE,COUNT
03237                COMPARE COUNT,RECORD
03242                GOTO RECRD IF NOT EQUAL
03245                MOVE ONE,COUNT
03250                CLEAR MASTER
03252                GOTO SORT
         .
03254    END         CHAIN EXIT
         .
03256    MOVE        RETURN IF NOT LESS
03260                RETURN IF EQUAL
03262                LOAD N5 FROM COUNT OF N1,N2,N3,N4,N5
03273                MOVE COUNT TO SMALL
03276                RETURN
         .
03277    SETFLG      MOVE ONE,FLAG
03302                BKSP 1
03304                GOTO CNT
03306                STOP

03307    START
03311    ASK
03313    REWD
03315    SORT
```

```
03317   RD
03321   SETFLG
03323   CNT
03325   END
03327   FEOF
03331   M1
03333   FIND
03335   MOVE
03337   MOVSTR
03341   MRG
03343   MERGE
03345   COPY
03347   TRNSFR
03351   SRCH
03353   SETUP
03355   RDWR
03357   RECRD

03361   N1
03363   N2
03365   N3
03367   N4
03371   N5
03373   N6
03375   MASTER
03377   TST
03401   NL
03403   DUMMY
03405   EXIT
03407   FLAG
03411   COUNT
03413   SMALL
03415   CNTSV
03417   ONE
03421   ZERO
03423   TEMP
03425   RECORD
03427   CNT
03431   SIX
03433   TEN
```

UPDATE FILE DISPLAY PROGRAM

```
              . PROGRAM LIST
              .
              . LIST UPDATE FILE
              . READS IN RECORDS FROM SCRATCH FILE ON REAR DECK
              . AND DISPLAYS THEM ON THE SCREEN
              .
     01000    BUFF        DIM 89
     01134    EXIT        INIT "001    "
              .
     01145    START       REWIND 1
     01147    RD          READ 1,BUFF
     01153                GOTO END IF OVER
     01156                DISPLAY *P1:12,*EL,BUFF
     01166                GOTO RD
              .
     01170    END         CHAIN EXIT
     01172                STOP

     01173    START
     01175    END
     01177    RD

     01201    BUFF
     01203    EXIT
```

```
         . UPDATE PROGRAM
         .
         . ALLOWS USER TO TYPE IN DESIRED INFORMATION.
         . THE DATA IS THEN PACKED AND WRITTEN OUT TO TAPE.
         . THE SCRATCH FILE ON THE REAR DECK IS USED FOR
         . THE UPDATE FILE.
         .
         .
01000    NAME       DIM 20
01027    ADDR       DIM 30
01070    SSN        DIM 11
01106    BUSNES     DIM 20
01135    CCODE      DIM 4
01144    TEMP       DIM 10
01161    UPDTE      INIT "UPDATE"
01172    END        INIT "END"
01200    EXIT       INIT "001    "
01211    SLASH      INIT "/"
01215    BUFF       DIM 89
01351    DUMMY      INIT "**WEOF**"
01364    START      DISPLAY *H1,*V1,*EF,*H15,"UPDATE PROGRAM"
01413               DISPLAY *H1,*V2,"TYPE IN THE REQUESTED INFO."
01464               PREPARE 1
         .
         . KEYIN INFORMATION FOR UPDATE RECORDS
         .
01466    UPDAT      KEYIN *P1:5,*EF,"NAME (LAST,FIRST):",NAME
01521               APPEND NAME,BUFF
01524               APPEND SLASH,BUFF
01527               KEYIN *H1,*V6,"ADDRESS:",ADDR
01547               APPEND ADDR,BUFF
01552               APPEND SLASH,BUFF
01555               KEYIN *H1,*V7,"SOCIAL SECURITY NUMBER:",SSN
01614               APPEND SSN,BUFF
01617               APPEND SLASH,BUFF
01622               KEYIN *H1,*V8,"COMPANY:",BUSNES
01642               APPEND BUSNES,BUFF
01645               APPEND SLASH,BUFF
01650               KEYIN *H1,*V9,"CUSTOMER CODE:",CCODE
01676               APPEND CCODE,BUFF
01701               RESET BUFF
         .
         . WRITE BUFFER TO UPDATE FILE
         .
01704               WRITE 1,BUFF
01710               CLEAR BUFF
         .
```

```
               . SEE IF END OF UPDATE OR MORE INFO
               .
      01712    ASK          KEYIN *H1,*V11,"UPDATE OR END?",TEMP
      01740                 MATCH TEMP,UPDATE
      01743                 GOTO UPDAT IF EQUAL
      01746                 MATCH TEMP,END
      01751                 GOTO ASK IF NOT EQUAL
               .
               . IF END THEN WRITE DUMMY END OF FILE AND EOF
               . TO UPDATE FILE
      01754                 WRITE 1,DUMMY
      01760                 WEOF 1
      01762                 CHAIN EXIT
      01764                 STOP

      01765    START
      01767    UPDAT
      01771    ASK

      01773    NAME
      01775    ADDR
      01777    SSN
      02001    BUSNES
      02003    CCODE
      02005    TEMP
      02007    UPDDTE
      02011    END
      02013    EXIT
      02015    SLASH
      02017    BUFF
      02021    DUMMY
```

```
        . PROGRAM COPY
        .
        . COPY FILE FROM FRONT DECK TO REAR OR REAR
        . DECK TO FRONT
        .
01000   EXIT        INIT "001   "
01011   BUFF        DIM 127
01213   TEMP        DIM 10
01230   FRONT       INIT "FRONT"
01240   BACK        INIT "BACK"
        .
01247   START       DISPLAY *P1:1,*EF,"COPY FRONT OR BACK TAPE?":
01307               KEYIN TEMP
01312               MATCH TEMP,FRONT
01315               GOTO COPYF IF EQUAL
01320               MATCH TEMP,BACK
01323               GOTO START IF NOT EQUAL
        .
        . COPY BACK DECK TO FRONT DECK
        .
01326   COPYB       REWIND 1
01330               PREPARE 2
01332   LOOPB       READ 1,BUFF
01336               GOTO ENDB IF OVER
01341               WRITE 2,BUFF
01345               GOTO LOOPB
01347   ENDB        WEOF 2
01351               CHAIN EXIT
        .
        . COPY FRONT DECK TO BACK DECK
        .
01353   COPYF       REWIND 2
01355               PREPARE 1
01357   LOOPF       READ 2,BUFF
01363               GOTO ENDF IF OVER
01366               WRITE 1,BUFF
01372               GOTO LOOPF
01374   ENDF        WEOF 1
01376               CHAIN EXIT
01400               STOP

01401   START
01403   COPYF
01405   COPYB
01407   ENDB
01411   LOOPB
01413   ENDF
```

```
01415   LOOPF

01417   EXIT
01421   BUFF
01423   TEMP
01425   FRONT
01427   BACK
```