Master

SDS TECHNICAL INFORMATION

WRITER'S MASTER COPY

SDS 940 FORTRAN II TECHNICAL NOTES

90 11 42A March 1967

DO NOT REMOVE

SCIENTIFIC DATA SYSTEMS

Price: $ .50

SDS 940 FORTRAN II
TECHNICAL NOTES

90 11 42A March 1967

SDS

SCIENTIFIC DATA SYSTEMS/1649 Seventeenth Street/Santa Monica California

Printed in U.S.A.

PREFACE

This manual contains technical information which a systems programmer may find useful when maintaining or modifying the SDS 940 FORTRAN II Sub-System.

This supplement is intended to be a guide for experienced programmers, and familiarity with the following publications will be helpful to the reader.

SDS 940 Computer Reference Manual	90 06 40
SDS 940 Time Sharing System Reference Manual	90 11 16
SDS 940 FORTRAN II Reference Manual	90 11 10
SDS 940 TAP Reference Manual	90 11 17
SDS 940 DDT Reference Manual	90 11 13

CONTENTS

1. FORTRAN COMPILER

COMPILER STRUCTURE

The 940 FORTRAN II compiler uses a set of data structure conventions with interpretive operations for some of the data manipulations it performs. According to the data structure conventions, there are a fixed set of 25 lists numbered 0-24. These lists can be used as a symbol table for fixed point scaler identifiers, floating, array, and dummy variable identifiers. Other functions of the lists are to hold the stack of exits for recursive calls, hold generated pieces of machine code awaiting rearrangement, and produce a work list that plays a special role as a push-down accumulator.

Each list occupies a contiguous block of storage. (See Figure 1.)

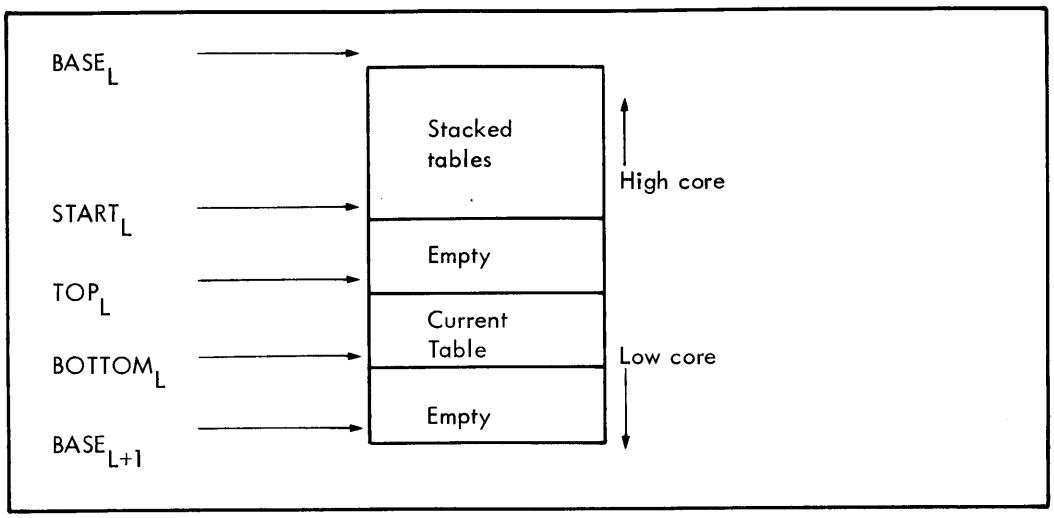


Figure 1.

Each push-down list of tables can be used as a last-in-first-out and as a first-in-first-out stack. For a list labeled L, there are four pointers to its block of storage.

$BASE_L$ points to the word before the list. Thus the i^{th} word of the list is in location $BASE_L + i$.

$START_L$ points to the last word of the pushed-down tables, the word before the region available to the current table.

TOP_L points to the word before the current table.

$BOTTOM_L$ points to the last word of the current table.

$BASE_{L+1}$ serves as a limit on $List_L$. It is the word before the region allocated to the next list. Thus, the following relationship is always true: $BASE_L \leq START_L \leq TOP_L \leq BOTTOM_L \leq BASE_{L+1}$.

The following list describes some of the interpretive operations (POPs, or programmed operators) that use only the current table of a list. In these descriptions, M is used as the effective memory address of an instruction after indirect addressing and indexing, and m or [M] as the contents of that address. The work list is designated by W.

FET M Fetch m

$BOTTOM_W \leftarrow BOTTOM_W + 1$, $[BOTTOM_W] \leftarrow m$. This stacks m on the bottom of the work list. This POP may be thought of as an LDA to the work list.

ADR M Address M

Stack M on the bottom of the work list. This POP corresponds to an immediate load to the work list.

SOB M Save on bottom of M

Stack the contents of the hardware accumulator on the bottom of $List_M$.

MON M Move onto M

Unstack the bottom word of the work list and stack it as the bottom word of $List_M$.

MOF M Move off M

Unstack the bottom word of $List_M$, and stack it on the bottom of the work list.

LDA * BOTTOM + L (A hardware instruction)

Bring to the hardware accumulator the word from the bottom of $List_L$. The asterisk signifies indirect addressing.

TOT M Take off top of M

Unstack the top word of the current table of $List_M$, and save it on the bottom of the work list.
$TOP_L \leftarrow TOP_L + 1$; $BOTTOM_W \leftarrow BOTTOM_W + 1$; $[BOTTOM_W] \leftarrow [TOP_L]$.

SKR BOTTOM + L (A hardware instruction)

Unstack the bottom word of $List_L$. $BOTTOM_L \leftarrow BOTTOM_L - 1$.

MIN TOP + L (A hardware instruction)

Unstack the top word of the current table of $List_L$. $TOP_L \leftarrow TOP + 1$.

LCF M Load Central from M

Load words CTL1 and CTL2 with the two words from the top of the current table of $List_M$.

LCO M Load Central off M

Same as LCF, except that the two words are unstacked from $List_M$.

MCO M Move Central onto M

Stack the two words CTL1 and CTL2 on the bottom of $List_M$.

Note that words are added to a list at the bottom, but may be taken off at the bottom (last-in-first-out: a stack) or the top (first-in-first-out: a queue). All operations that add words to a list use the SOB operation. SOB checks whether the allocated space if full ($BOTTOM_L = BASE_{L+1}$), and if so, calls on a storage allocator to move the lists and change the pointers.

It is possible to create a new current table T_1 on a list without harming the previous current table T_0, that will again become the current table when T_1 is released. This is done by means of two operations:

RSV M Reserve M

Stack $START_M - BASE_M$ and $TOP_M - BASE_M$ on the current table of $List_M$, then $START_M \leftarrow BOTTOM_M$, $TOP_M \leftarrow BOTTOM_M$, creating a new (empty) current table on $List_M$.

Thus if the previous appearance of $List_M$ was as shown in Figure 2,

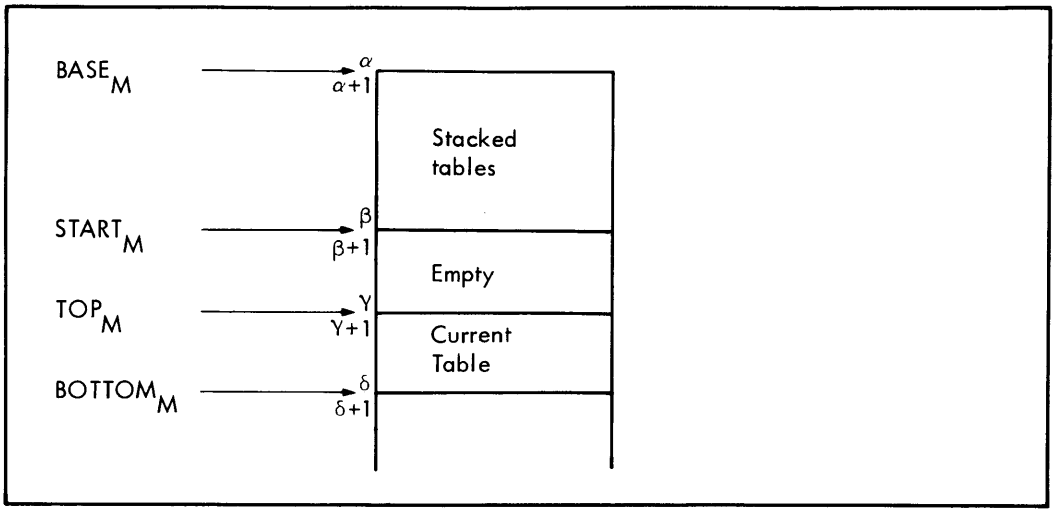


Figure 2.

we would now have the situation shown in Figure 3.

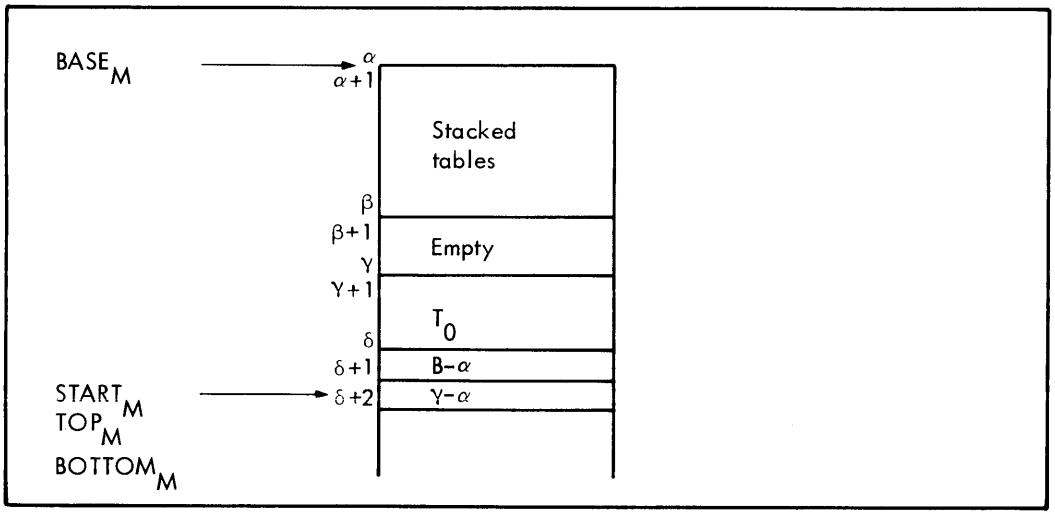


Figure 3.

After several uses of SOB_M and TOT M, and possible memory reallocations which move the entire list without changing its contents, it is possible to have the situation shown in Figure 4 (r is a relocation constant).

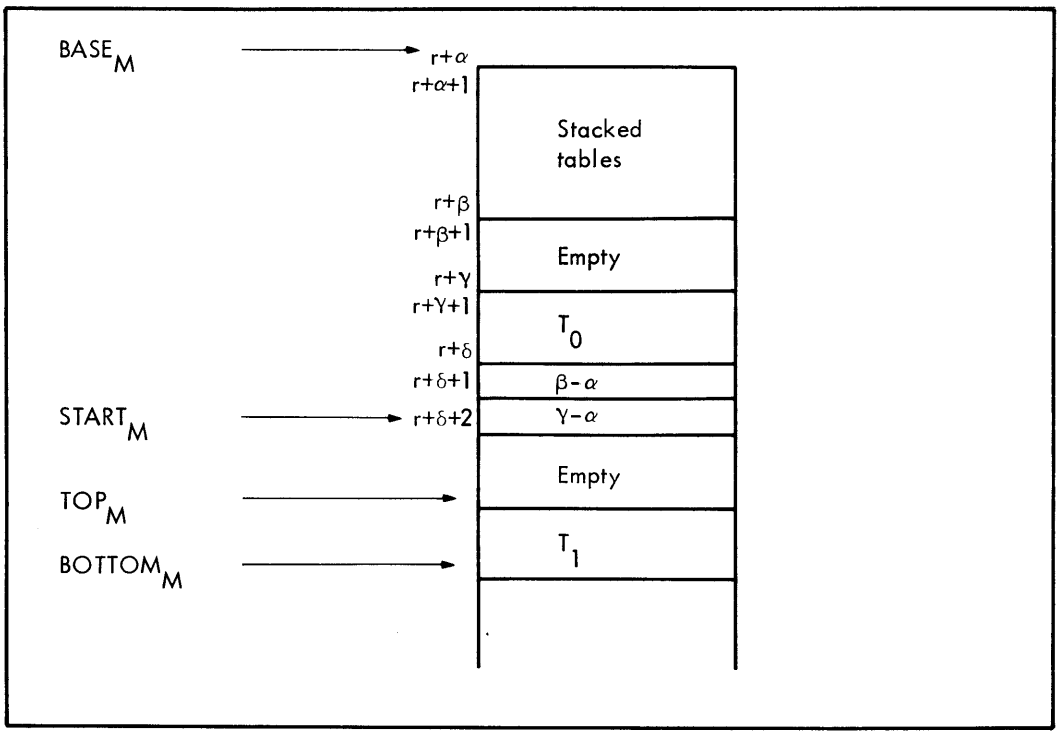


Figure 4.

The second operation is

RLS M Release M

$BOTTOM_M \leftarrow START_M - 2$,

$TOP_M \leftarrow [START_M] + BASE_M$,

$START_M \leftarrow [START\ M - 1] + BASE_M$.

We now have $BOTTOM_M = r + \delta$,

$TOP_M = r + \gamma$, $START_M = r + \beta$,

$BASE_M = r + \alpha$, so that the original status of the list has been restored with T_o as the current table.

It is not useful to save absolute addresses of words stored on lists that are constantly subject to relocation. Instead, one should store a pointer to a word that contains the number of the list and the location of the word relative to the base of the list.

BOP M Bottom pointer of M

Saves a pointer on the work list that designates a bottom word on list M.

CNT M Count M

Saves the size of the current table on $List_M$ on the work list.

SAL M Save a list M

Saves $START_M - BASE_M$, $TOP_M = BASE_M$, and $BOTTOM_M - BASE_M$ on the save list

REC M Recover M

Is inverse to SAL, and restores $START_M$, TOP_M, and $BOTTOM_M$ to their previous values.

Each list has a standard size of one to five words of item stored on it. For example, a list of floating point constants might have two as its standard item length. In addition, lists such as symbol tables have items which begin with a one- or two-word key, the symbol itself, followed by other information.

SER M Search M

Search the list M for an item that has a key equal to CTL2, or equal to the two-word pair (CTL1, CTL2), depending on what the key-length of M is. The pointer to the matching item is saved on the work list.

CONTROL STRUCTURES

The program is organized as a set of recursive subroutines. Exits are saved on one of the lists, the exit list. The programmed operators are implemented by recursive subroutines so that they can use themselves and each other. Apart from those subroutines accessed by the programmed operators, a recursive subroutine may be reached by

JRS M Jump to recursive subroutine M

At each level of subroutine nesting an answer bit is kept and used to record the results of tests.

JAT M Jump to M if answer is true

JAF M Jump to M if answer is false

Many of the operators described earlier set the answer true or false. For example the programmed operators that remove items from lists set the answer false if the source list is empty. The search instruction sets the answer false if no match is found. We also have

CSA M Character scan or alternative

If the next input character equals m, scan over it and set the answer true; otherwise do not scan, and set the answer false.

SNE M Set nonempty

Set the answer true if the current table of $List_M$ is nonempty, otherwise false.

SOC M Set on character

Set the answer true if the next input character has, in its entry in a certain table, a flag bit set in the same position as the bit set in M. This instruction can be used to ask if the next character is an alphanumeric.

SOF M Set on flag M

This operation is the same as SOC, except that it tests the bottom word on the work list, rather than the next input character.

SOL M Set out of limit

Set the answer true if the absolute value of the double precision hardware accumulator is not greater than the double precision limit M.

The following POP's may be used to do backtracking:

TRY M Enter the recursive subroutine M

If the subroutine is left normally by a transfer to EXIT, control returns to the instruction following the TRY, with the answer set to TRUE. If an exit occurs by a transfer to FAIL, control returns with the answer set to FALSE, and with the list pointers reset to their values at the time TRY was executed.

FEX M Fail exit M

After execution of this instruction, a transfer to FAIL will cause control to go to M, with lists restored to their state when FEX was executed.

CSF M Character scan or fail

If the next input character is M, scan it; otherwise go to FAIL.

QSF M Quote scan or fail

Scan on the input string the string stored at M, or go to FAIL.

FTC SYSTEM MAKE

A save file of the compiler is produced in the following manner.

Assemble the symbolic files 1C, 2C, . . . , 6C producing the binary files 1B, 2B, . . . , 6B. These files contain

temporary variables and compiler lists

FTC command processor: analyzers for 70% of the legal statements

30% of the statement analyzers, statement element analyzers, post-compilation printout routines, and a memory allocator

element analyzers

compiler programmed operators

I-O routines, read only storage

Next use DDT to load the binary files. Temporary storage is kept in block zero below and the read only compiler code in blocks one and two.

```
200;T /1B/.          temporary storage
4000;T /2B/.   }
    :          }     code to be saved
;T /6B/.       }
```

Return to the EXEC and save two blocks.

```
SAVE CORE FROM 4000 to 13777 ON /FTC/,
STARTING LOCATION 4000.
```

2. FORTRAN OPERATING SYSTEM

CHARACTER CODES

Trimmed ASCII character code is used throughout the FORTRAN II system. FTC packs four characters per word internally. However, all output strings from FTC for FOS are composed of 8-bit characters packed three to a word. FOS uses three characters per word with one exception in the loader. Since the assembler packs symbols four characters per word, the loader expects this. When an assembled program is loaded, its name is converted to the FOS standard of three characters per word.

LIBRARY FILE MAINTENANCE ROUTINE

The FORTRAN library file consists of a sequence of TAP and FTC binary programs. The library file maintenance routine, LIBFLE, copies from this file to a second file. During this process, it deletes a specified program or inserts the contents of a third file before the specified program in the original file.

The commands are:

Delete <library file>.
<new library file>.

D <program name>.

This operation produces a new copy of the library file with the specified program deleted.

Insert <library file>.
<new library file>.

I <program name>.

<file name of new programs>.

This operation produces a new copy of the library file with the contents of the third file inserted before the specified program in the library file.

List (and copy)

<library file>.
<2nd file>.

L (CR)

This operation lists the programs on the first file in order while copying the file. If a copy is unwanted, the NOTHING file should be specified.

FLOW OUTLINE

1. Accept input file name.
2. Accept output file name.
3. Accept command letter (D, I, or L).
4. Accept 1 to 6 alphanumeric characters
5. If this is an insert operation, accept the insert file name.
6. Open the files.
7. Read the first five words of the next program from the old library file.

8. If this is a named program, insert the insert file or delete this program.

9. Copy or skip the rest of the program from the old library file.

10. Repeat the three previous steps until the end of the old library file is reached.

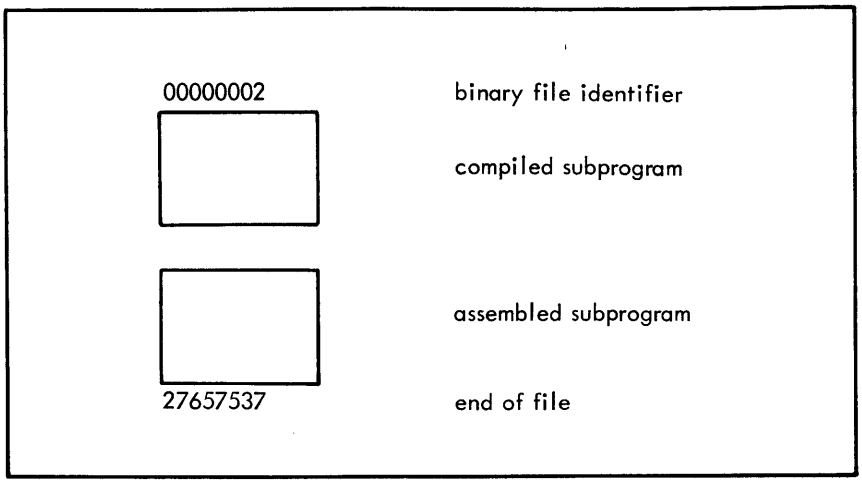


Figure 5. Library File Format

LOADER

The FOS loader is two separate loaders in one since the compiler and assembler output is very different. FTC makes a single pass over the source statements. The second pass is performed in the loader. The compiled binary file is read into core followed by a number of tables generated during compilation. At this point the address fields of instructions contain keys that reference entries in the tables. A pass over the code in core sets up addresses. The tables are no longer needed and this space is released for use by the next program.

00000000 04000000 00000000 01002004 01002004 00600000	Block count (Block number) 02000000, if subprogram Entry point Subprogram name, if not main program BLK LOP (Special Loader OP – "mark end of block")
TEXT	
011NNNNN	ABS LOP NNNNN is number of words of fixed and floating constants that follow in this block
Fixed Constants	
Floating Constants	
00600000	BLK LOP
Array table	
00600000	BLK LOP
Fixed special table	
00600000	BLK LOP
Floating special table	
00600000	BLK LOP
10 special words	
Names of required subprograms	
006NNNNN	BLK LOP NNNNN is the number of words of debugging symbol table which follows. NNNNN may equal zero.
Symbols	
32465152	Three line-feed characters. Compiled subprogram end.

TEXT

The text is composed of absolute instructions, relocatable instructions, absolute data, and special loader OPS (called LOPS). The different types of loader OPS are:

BLK	006xxxxx	Block end marker
LBL	003xxxxx	Label LOP
ABS	011xxxxx	Absolute LOP indicating x number of data words follow
SYS	005xxxxx	System LOP that is converted to BRM* instruction at load time to branch to a routine
DEL	004xxxxx	The address, xxxxx, is added to the following instruction's address at load time

The text also contains some programmed operators that are converted into machine code by the loader at load time. They are:

POP	Machine Code
124	LDA
130	ADD
134	SUB
106	STA
146	CNA

If the instruction is relocatable, the sign bit is a 1 and the 14-bit address field refers to one of nine different tables.

34340 - 37777	dummy
30704 - 34337	temp
25250 - 30703	link
21614 - 25247	array
16160 - 21613	fixed constant
12524 - 16157	floating constant
7070 - 12523	label key
3434 - 7067	fixed scalar
0 - 3433	floating scalar

ARRAY TABLE

The array table contains one entry for each array referred to by the program. This word gives the location of the array and, if the array is in COMMON, the word is negative.

FIXED SPECIAL TABLE

Each fixed scalar that appears in an EQUIVALENCE or COMMON statement produces a two-word entry in this table. The first word is the scalar's identification number and the second word its address, similar to the addresses that appear in the array table.

FLOATING SPECIAL TABLE

Each floating scalar that appears in an EQUIVALENCE or COMMON statement produces a two-word entry in this table. The first word is the scalar's identification number and the second word is its address, similar to the addresses that appear in the array table.

TEN SPECIAL WORDS

Number of fixed constants
Number of words of floating constant
Beginning of link table
Beginning of dummy storage
Beginning of temporary storage
Beginning of array storage
Beginning of fixed scalar storage
Beginning of floating scalar storage
End of floating scalar storage + 1
Size of COMMON

NAMES OF REQUIRED SUBPROGRAMS

Each subprogram required by this program causes a two-word BCD entry in this table.

ASSEMBLED SUBPROGRAM

FOS loads standard 940 assembler output. Binary output is divided into logical, variable length records. Each record begins with a control word that defines its type. Bits 0, 1, and 2 normally signify the type. The first word of the binary output is a 3-bit register (cf. "0 -Binary Program Follows" below) whose single entry is an octal 4.

Bits 0, 1, and 2 (octal)		Meaning
0		Binary program follows
1		Programmed operator follows
200	Bits 0-8 are used	End of program
201		Origin of literal table is in address field
3		OPD follows
4		External symbol definition (s) follows
5		Identification record follows
6		External symbol usage table follows
7		Symbol table follows

The remaining bits in the control word and the format of the record which follows are different for each type.

0 - Binary Program Follows

Bits 10-23 of the control word are added to the current value of the location counter. A binary program consists of groups of eight machine commands preceded by eight groups of three bits packed into a single word (the 3-bit register). Each group of three bits is associated with a corresponding instruction or control word that follows and serves as a loading indicator for that word. The following indicators are used:

3-Bits (octal)	Meaning
0	Absolute address
1	Evaluate address from external symbol table mod 2^{14}
2	Relocate address mod 2^{14}
3	Relocate word mod 2^{24}
4	Abandon binary program format – next word is a control word.
5	Evaluate word from external symbol table mod 2^{24}
6	unused
7	literal reference – relocate address mod 2^{14}

1 - Programmed Operator Follows

Bits 2-8 of the control word determine the position of a transfer command that is placed by the loader in the programmed operator transfer vector ($100-177_8$). Bits 10-23 determine the address of the transfer command. The information following is a binary program that follows the previous program; i.e., the location counter is unaffected by POPD.

200 - End of Program

No other bits in the control word are significant. The 200 record is a one-word record.

201 - Origin of Literal Table

The origin of the literal table is found in the address field.

3 - Defined Operation Follows

All OPDs are punched in a form of a standard symbol (cf. "4-External Symbol Definition(s) Follows" below).

4 - External Symbol Definition(s) Follows

Each definition consists of a block of three words. The first two words contain the six characters of the symbol in ASCII code, left-justified with trailing blanks. The third word contains the symbol value. Bit 12 of the second word signifies relocation of the external symbol value. Relocation of external symbols is performed modulo 2^{24}. Each block of such definitions is terminated by a single word of all 1's.

5 - Identification Record Follows

The identification record consists of one block of three words. The format of the block is identical to that for each entry of type 201 records (see above), although only the six characters of the identification symbol are meaningful.

6 - External Symbol Usage Table Follows

Each entry of the usage table is a three-word block of the same format as type 4 records (see above).

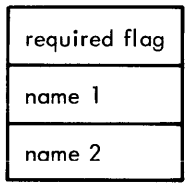
7 - Symbol Table Follows

The format of the local symbol table is the same as for type 4 records (see above). The order of records is as follows:

[ident record]
[external symbol usage table] (if any)
[literal table origin]
binary program
programmed operators } in any arrangement
external symbol definitions
end of program

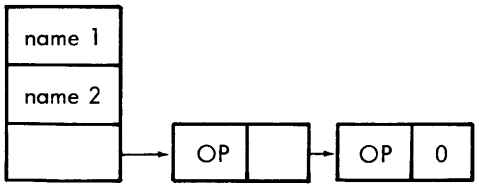
LINKING SYMBOLS

Undefined symbol names are entered in the LINKS vector, which works down from core during loading. A typical entry is:



When the symbol is defined, the program required flag is replaced with a branch to the entry point.

A table of three-word entries works up in core from the end of temporary storage (block zero). Undefined symbol chains begin at this table.



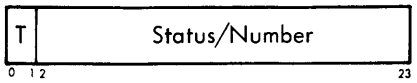
Undefined symbols are linked in this manner until loading is completed.

PROGRAMMED OPERATOR LINKAGES

To load properly, compiled POP's must be between 100_8 and 167_8. In assembled programs, POP's between 160_8 and 177_8 as well as all 940 system POP's may be used.

FILES

The file numbers table, FLNMT, consists of ten words laid out as follows:



where

T is a file type indicator specifying whether one file is used for input or output (0 means input file, 1 means output file).

Status/Number is a value indicating the status and identifying number of the file (a value of -1 means the file is not opened; any other value means the file is opened and its identifying number is the value given).

File unit numbers 2 through 9 may be assigned at will. Units zero and one are permanently assigned to teletype input and output. Input or output to an unopened unit selects the controlling teletype by default. A bell sounds whenever teletype is expected.

OPEN STATEMENT

The statement "OPEN (<expression>, {INPUT / OUTPUT}, /file/)"

generates

```
                                    normal fixed expression evaluation
                                    normal fixed expression evaluation
                                    normal fixed expression evaluation
SBRM*     DECFL                     declare file
NOP       0                         or 1 if output
BRU       D1                        branch to next statement
/FI                                 ASCII character string
LE/
DI
```

CLOSE STATEMENT

"CLOSE (<expression>)" generates the code

```
                                    normal fixed expression evaluation
                                    normal fixed expression evaluation
                                    normal fixed expression evaluation
                                    normal fixed expression evaluation
SBRM*     CLFLE                     close file
```

The expression may be a list of expressions separated by commas, in which case the above code is repeated for each file to be closed.

FORK STATEMENT

A subroutine that is normally entered with

CAL SUB (arg1,...,argN)

may be started up as a fork instead, with

FORK SUB (arg1,...,argN)

The code generated for this is

```
        LDA   D4
        STA   D1
        LDA   D5
        BRS   9
        BRU   D3      continue normal flow
D5      ZRO   D1      address of fork table
D4      ZRO   D2      address of subroutine call
D1      BSS   7       fork table

D2      (standard code for ...
            CALL SUB ( )  )
        BRS           10    fork stop
D3      ...           next statement in the main branch
```

Note that the subroutine is standard in all respects.

When the RETURN statement is executed, a normal return is made from the subroutine. A BRS 10 is then executed in the main code. Therefore, a single subroutine may be activated either as a fork or as a normal subroutine from within the same program. The user must avoid doing both at once. To assist him, the WAIT statement generates a BRS 31.

DEBUGGER

The debugger's context is always the main program or a particular subprogram. The main program is assumed initially after the completion of loading. A line beginning with a legal FORTRAN name followed by a comma switches the debugger to a new program. The name string is not saved for any length of time but is used in the search that follows immediately. A successful search results in the updating of three cells:

UPROGS	user program start
UDATAS	user data start
UPROGN	user program end

These three variables define the current environment for the debugger.

If debugging information is requested at compile time, the statement label POP (STL-156_8) is inserted just before each statement. This POP is produced for all statements including non-executable ones to insure accurate relative addressing by the debugger. (100-3 = statement 100 minus 3 statements). The address field of STL holds the statement label number or zero.

Setting a breakpoint amounts to converting the STL to the breakpoint POP (BKP-152_8). When BKP is executed, control returns to the FOS command processor. The statement label is typed at this time.

Three cells within FOS hold pointers to the three current breakpointed statements. When breakpoints are not set, these cells hold all ones. The three cells are BK1ADR, BK2ADR, BK3ADR.

Removing breakpoint n converts the BKP POP addressed by BknADR back to an STL POP and stores -1 in BKnADR.

FOS SYSTEM MAKE

A save file of the loader and run time is produced in the following manner:

Assemble the symbolic files 1L, 2L, 1R, 2R, 3R, 4R producing the binary files B1L, B2L, B1R, B2R, B3R, B4R.

1L	temporary storage for the loader	2R	run time POP's
2L	loader	3R	format processor
1R	temporary storage for run time	4R	debugger

Use DDT to load the assembled binary.

```
400;t/B1L/.
4000;T/B2L/.
```

Determine the last location, α, used by the loader and the value of BRUNT, β, the location where a branch to the start of run time should be stored.

Delete the loader symbol table and load run time.

```
400;T/B1R/.
α+1;T/B2R/.
;T/B3R/.
;T/B4R/.
β/BRU PS (run time) program start
4001/ BRU RS (run time) program re-start
```

Return to the EXEC and save the code from 4000 to the last location, α, used by the run time system.

SAVE CORE FROM 4000 to α, starting location 4000.

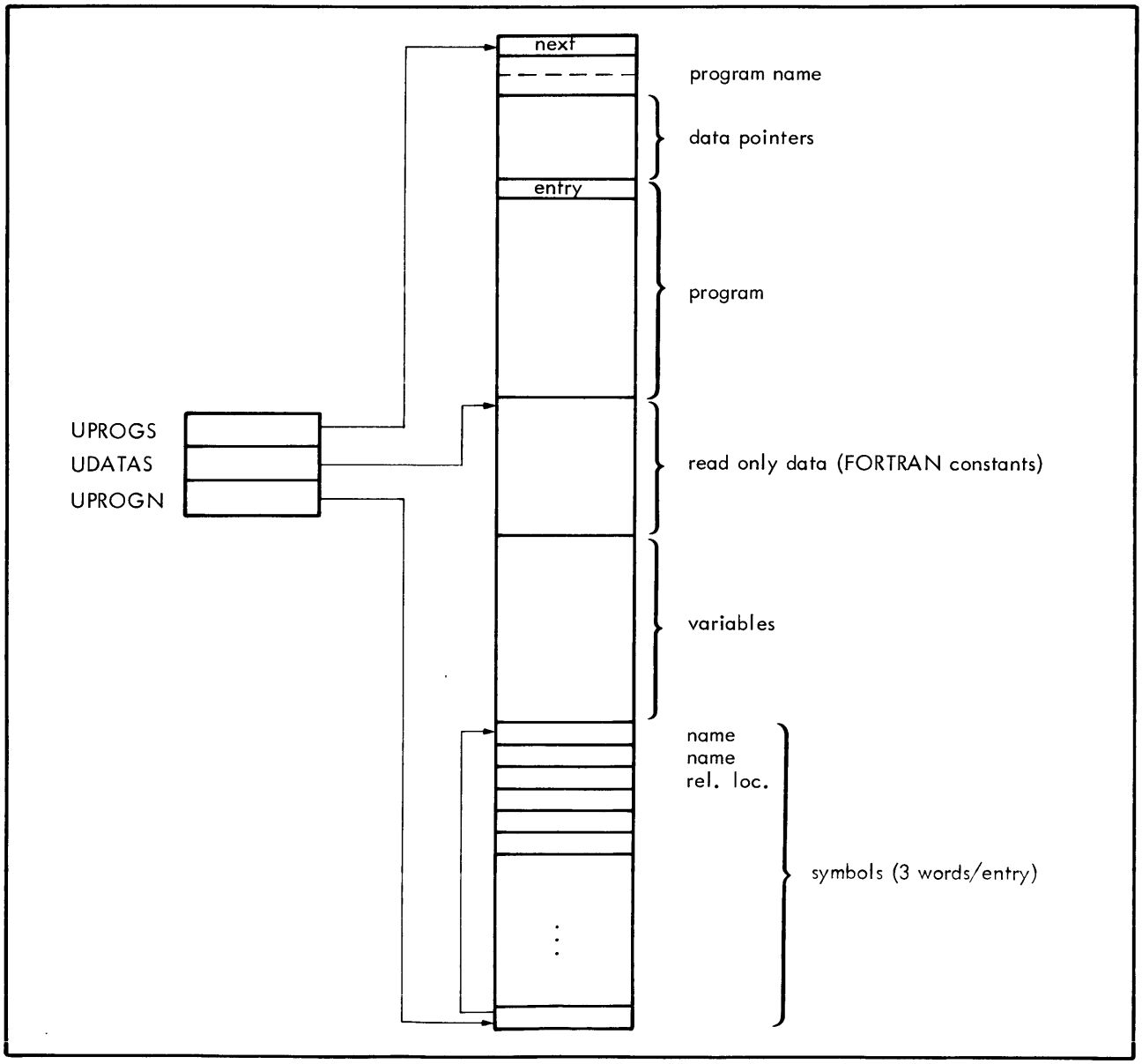


Figure 8. Memory Layout of Compiled Programs and Subprograms