WRITER'S MASTER COPY

Reprinted - 8-13-69

SCIENTIFIC DATA SYSTEMS

Operations Manual

SDS FORTRAN II

Price: $1.50

SDS FORTRAN II OPERATIONS MANUAL

SDS 900 Series Computers

90 05 87B

July 1966

SDS

CIENTIFIC DATA SYSTEMS A XEROX COMPANY/701 South Aviation Boulevard/El Segundo, California 90245

REVISIONS

This publication, 90 05 87B, dated July 1966, is a minor revision of the SDS 900 Series FORTRAN II Operations Manual, 90 05 87A.

Changes to the previous edition are indicated by a vertical line in the margin of each affected page.

CONTENTS

RELATED PUBLICATIONS

Name of Manual	Publication No.
SDS FORTRAN II Reference Manual	90 00 03
SDS SYMBOL and META-SYMBOL Reference Manual	90 05 06
SDS MONARCH Reference Manual	90 05 66
SDS 910 Computer Reference Manual	90 00 08
SDS 920 Computer Reference Manual	90 00 09
SDS 925 Computer Reference Manual	90 00 99
SDS 930 Computer Reference Manual	90 00 64

1. INTRODUCTION

The SDS FORTRAN II System consists of four parts: Compiler, Loader, Library and Run-Time.

1. Compiler. The SDS FORTRAN II Compiler is a one-pass routine. It reads the source program only once and simultaneously generates the object program in a form acceptable to the FORTRAN Loader. Since the entire compiler fits into 4096 words of memory, no reloading is necessary to process batch jobs.
2. Loader. The Loader is used to load all object programs that have been either compiled in FORTRAN or written in machine language to be used as FORTRAN subprograms. It also loads the standard library routines that are requested by any program.
3. Library. The Library contains all the standard subroutines that may be called for, explicitly or implicitly. These include elementary mathematical functions (such as, square root, sine, etc.), miscellaneous functions (such as, absolute value, maximum value, modulo, etc.) and system library routines (such as, input/output routines).
4. Run-Time. The Run-Time consists of programmed operators for use by FORTRAN object programs, format scan routines for input/output operations, and control constants for use during execution.

The SDS FORTRAN II System is a complete package for compiling, loading, and executing FORTRAN II programs. The configuration required is an SDS 900 Series Computer, with 4096 words of memory, a paper tape reader and punch, and a console typewriter. No magnetic tapes or other auxiliary storage media are necessary.

This manual assumes that the reader knows basic FORTRAN II syntax and semantics and that he is familiar with one of the SDS 900 Series Computers.

2. COMPILER

COMPILING SOURCE PROGRAMS

The procedure for compiling FORTRAN source programs is:

1. Turn spooler off.
2. Mount system tape on spooler and insert in paper tape reader. Set brake.
3. Turn spooler on.
4. Load the compiler by the standard FILL procedure. When the compiler is completely loaded, the message:

   ```
   SET PAPER
   COMPILER READY
   ```

   is typed and the computer halts.
5. Position typewriter paper at the top of a page. The compiler will now start each new program on a new page.
6. Turn spooler off.
7. Remove system tape from paper tape reader.
8. Insert source program in paper tape reader. Set brake.
9. Set BREAKPOINT switches 1 and 2 as follows:

Switch 1 RESET (up)	Punch object program
SET (down)	Suppress punching
Switch 2 RESET (up)	Type source statements
SET (down)	Suppress typing

   Note: With both breakpoints set, the compiler operates at full speed in the diagnostic mode, typing only control statements, statements with errors and the program summary.
10. Clear the HALT to begin compilation.

The source program need not be a single, physical tape. The compiler ignores leading blank tape, but stops when it encounters a stop code.

Control Statements

FORTRAN source programs may, but need not, begin with compiler control statements. These are statements introduced by an asterisk (in column 1 on cards, or immediately following the carriage return on paper tape), and may be used to indicate the start of a program. Since the compiler begins a new program when a control statement is encountered, these must all precede the first actual source program statement.

This feature permits direct compilation of FORTRAN programs written for large, monitor-controlled machines that admit control statements such as:

```
*XEQ
*LIST
*LIBE
etc.
```

All control statements will be typed, even when switch 2 is set. (See section 3, "LOADER" on page 10.)

The end of the source program is determined by one of the following statements:

1. END statement – any line of source coding consisting solely of an END statement terminates the compilation; thus, an END FILE M statement may not be written:

   ```
    END
   X FILE M
   ```

   This is a restriction in all currently known FORTRAN compilers.

2. Control statement – defines the start of a new program regardless of whether or not a preceding END statement has been encountered.

When the end of the source program is reached, a program summary is typed out consisting of the following:

1. Program diagnostics, if any.
2. Program allocation – relative locations of all variables. The absolute addresses may be determined at run-time. (See Memory Layout at Run-Time.)
3. Common allocation – these addresses are relative to the end of memory on the run-time computer. The loader determines run-time memory size and assigns common storage accordingly. For example, 77777 becomes 7777 on a 4K machine and 17777 on an 8K machine.
4. Required subprograms, if any.

The punching of memory allocation information for the loader now completes the object program generation. To signify this, the message

THE END

is typed, the typewriter carriage advances to the top of the next page, the message

COMPILER READY

is typed, and the computer halts.

To compile another program, repeat steps 8 through 10 of the compiling procedure.

Compiler Halts

All halts during compilation display a "tagged" NOP (bits 1 and 4) in the C register, plus a number to indicate the type of halt. There are two such halts:

22000000	Compiler ready
22000001	If the source program is on two or more separate pieces of tape, the compiler halts, allowing the additional tapes to be mounted in the photo-reader.
22000002	Parity error during input – the incorrectly read character is displayed in A. The character may be corrected and processing continued, if desired.

Compiler Restart

The standard Restart Procedure (see Appendix A) may be used to discontinue compilation at any point and to reinitialize the compiler to step 5 above.

SYMBOL TABLE SIZE

Symbol table storage is dynamically allocated by the compiler, that is, none of the tables have fixed lengths; each may be lengthened, shortened, or relocated as items are added or removed, and no table can be exceeded until there are no unused locations anywhere in memory. The total number of words available for tables is 604 (920/930) and 484 (910/925) for a 4K machine. On larger machines, all the additional memory is used for tables, which means that it is virtually impossible to exceed the symbol tables.

Included in the table storage is working storage for statement translation. This area is expanded during analysis of each statement and contracted as the object program is punched out. Thus, its size fluctuates rapidly according to the size of the statements, making available symbol table storage difficult to predict. A reasonable guess for the upper bound on working storage (W below) in the average program is 150 words.

The following formula defines available table area:

$N + 2S + 6A + 2F + I + 2G + 4L + 2C + 3E + 3D + M + W \leq 604$ (for SDS 920/930)

$N + 2S + 6A + 2F + I + 2G + 4L + 2C + 3E + 3D + M + W \leq 484$ (for SDS 910/925)

where

N = no. of statement numbers
S = no. of scalar variables

A = no. of array variables
F = no. of floating-point constants
I = no. of integer constants
L = no. of LOCAL subprograms (arithmetic statement functions)
G = no. of GLOBAL subprograms (all other than LOCAL subprograms)
C = no. of COMMON identifiers
E = no. of EQUIVALENCEd identifiers
D = no. of DO loops
M = no. of FORMAT statements
W = size of working storage

Note: Since W fluctuates with each statement, an especially complex statement near the end of a long program may cause table overflow even though the symbols would not. In such cases, it is useful to move long, complex statements to the beginning of the program.

OUTPUT FORMAT

Paper tape produced by the compiler is blocked with a maximum of 93 words per block. The first word of each block is a block count and is zero for the first block. The last word of each block is a logical checksum for all words in that block with the exception of the checksum word. Each block is separated by a gap. All words between the block number and checksum are data words.

A layout of the data words on a typical compiler output tape is shown below.

00000000	Block count (Block number)
04000000	02000000, if subprogram
00000000	Entry point
53535353	
53535353	Subprogram name, if not main program
00600000	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	

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

In the SDS 910 Computer, the 146 POP is not converted; it is executed as a run-time CNA POP.

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

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 its identification number and the second word is 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 its 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.

COMPILER DIAGNOSTICS

The compiler checks FORTRAN source program errors extensively and pinpoints those detected to facilitate correction. In general, errors are nonfatal; the object program may still be produced and run, bearing in mind changes introduced by the errors, as described below.

Two types of diagnostics are provided: statement diagnostics and program diagnostics.

Statement Diagnostics

Most errors are caused because one particular statement is faulty. The compiler detects these errors when encountering such a statement and prints an error indication beneath it on the listing. If the compiler is operating in the non-list mode, only the statements in error are listed, along with the error indications.

Statements in error are discarded and compilation then proceeds as if they had never existed.

The compiler proceeds from left to right in translating a source statement. When an error occurs, the compiler notes the character at which the error became evident and prints a △ (delta) underneath it on the listing. The delta may indicate an error of:

1. Omission. The statement has ended and something further is required. The △ will follow the last character in the statement, e.g.,

```
A = B**
       △
```

2. Commission. The flagged character does not make sense where it is. The compiler cannot proceed beyond it, e.g.,

```
A = SQRTF (/B)
           △
```

3. Usage. A number or identifier that is incorrect will be flagged underneath its last character, since at this point the compiler had examined it completely, e.g.,

```
COMMON ALPHA, ALPHA
                  △
```

An error message will also be printed on the following line. These messages are described in the following paragraphs.

Syntax. At the flagged character, the statement no longer conforms to the syntax of any recognized type of statement.

Subscripts. The number of subscripts being used with the array does not equal the number declared for the array.

ID Declaration. The identifier marked is being used in a manner that contradicts a previous declaration.

Allocation. Allocation errors may occur in three statements:

1. In a DIMENSION statement, either

   A negative or zero dimension is specified;
   The lower limit for a subscript exceeds the upper limit; or
   The requested size of an array exceeds 16K.

2. An identifier appears in COMMON that has previously appeared in either COMMON or EQUIVALENCE.
3. In an EQUIVALENCE set, more than one identifier has previously appeared in either COMMON or EQUIVALENCE.

Number. Number errors are of two types:

1. The magnitude of the integer marked exceeds 8388607.
2. The number marked is a statement label that does not fall between 1 and 99999, inclusive.

Overflow. The statement cannot be compiled due to either:

1. Too many continuation cards.
2. Exhaustion of the compiler's working storage; in this case, compilation is terminated and the compiler initializes for a new job.

Program Diagnostics

Certain errors cannot be detected until the entire source program has been read. These errors will be indicated beneath the source listing, with the summary listing. They are described in the following paragraphs.

DO Nest Errors. The statement numbers listed were meant to close the range of a DO statement. The compiler cannot close the DO loop correctly if:

The closing statement is undefined. (See Labeling Errors, below.)

The closing statement is a transfer. The incrementing and testing of the DO loop will never take place.

The closing statement is within the range of another DO statement that follows this one (i.e., the ranges partially intersect). The results of such a situation can be determined by inspection.

Labeling Errors. The statement numbers listed are either:

Undefined. The program will run normally until a transfer to one of these statements is actually attempted. At this point, the typeout "ERR LABL" will occur, and the program will not proceed.

Multiply defined. All transfers will be made to the last statement encountered with each of the particular numbers.

Errors under COMMON ALLOCATION. If the bounds of COMMON are exceeded by improper use of EQUIVALENCE, those variables that cannot be assigned as requested will appear under COMMON ALLOCATION, preceded by the word "ERROR" instead of an octal location. Such variables will then be assigned again under PROGRAM ALLOCATION as if they had never appeared in the EQUIVALENCE.

The following listing illustrates most of the different types of error diagnostics.

```
#     1  C     THE FOLLOWING STATEMENTS WILL ILLUSTRATE THE ERROR CHECKING
#     2  C     FEATURES OF THE SDS 9ØØ SERIES FORTRAN II
#     3  C
#     4  C     ZERO OR NEGATIVE DIMENSIONS
#     5  C
#     6        DIMENSION ALPHA[Ø]
                               △
ALLOCATION
#     7        DIMENSION BETA[-1,3]
                               △
ALLOCATION
#     8  C
#     9  C     COMMON EXCEEDED [SEE BELOW UNDER COMMON ALLOCATION]
#    1Ø  C
#    11        DIMENSION A[3],R[2Ø]
#    12        COMMON X,Y,Z
#    13        EQUIVALENCE [A,Y]
#    14  C
#    15  C     FUNCTION NAME USED AS ARRAY
#    16  C
#    17     18 X # ROARING[I,B]
#    18        ROARING[2Ø,2Ø] # GOODOLD*GONEBY
                     △
ID DECLARATION
#    19  C
#    2Ø  C     WRONG NUMBER OF SUBSCRIPTS
#    21  C
#    22        Y # A[I,J]
                       △
SUBSCRIPTS
#    23  C
#    24  C     NUMBER TOO LARGE
#    25  C
#    26        J # 123456789
                           △
NUMBER
#    27  C
#    28  C     ARRAY TOO LARGE
#    29  C
#    3Ø        DIMENSION ENORMOUS[1ØØØ,1ØØØ]
                                          △
ALLOCATION
#    31  C
#    32  C     MISSING AND DUPLICATE STATEMENT NUMBERS [SEE BELOW]
#    33  C
#    34     13 X # Y
#    35     13 Y # X
#    36        GO TO 5
#    37  C
#    38  C     DO LOOP ERRORS [SEE BELOW]
#    39  C
#    4Ø        DO 3 I#1,1Ø
#    41        DO 4 J#1,3
#    42      4 IF [X-Y] 18,18,19
#    43     19 DO 6 I#1,1Ø
#    44         DO 7 J#1,1Ø
#    45     6 X # X&R[I]
#    46     7 Y # Y&R[I]
#    47  C
#    48  C    MISCELLANEOUS SYNTAX ERRORS
#    49  C
#    5Ø         READ 41, [R[I], I#1]
                                   △
SYNTAX
#    51         X # 3.*[[2.&Y]*SQRT[3.14159265359/[Y**2&Z**2-4.7[P-Q]]] & ABS[P]
                                                                △
SYNTAX
#    52         X # ALPHA*BETA**[1.&SQRT[12.6*P*-Q]/3.5]-2.**J
                                                △
SYNTAX
#    53         3.*P#Q
                △
SYNTAX
#    54         IF [P-Q] 27,16
                               △
SYNTAX
#    55         X # -[1.&2.8*[R[3]-4.*R[I]*[3.-SQRTF[P&Q/[1.&X**2]]]]]]
                                                                      △
SYNTAX
#    56      14 FORMAT [4F12.5,I7,14HTOTAL VALUES F12.Ø]
                                                      △

SYNTAX
#    57         END


DO NEST ERRORS

     6       4       3

LABELING ERRORS

    13       5       3

COMMON ALLOCATION

  77776 X         77774 Y         77772 Z         ERROR A


PROGRAM ALLOCATION

  ØØØØ5 A         ØØØ13 R         ØØØ63 I         ØØØ64 J
  ØØØ65 B

SUBPROGRAMS REQUIRED

  ROARING
THE END
```

3. LOADER

FORTRAN OBJECT PROGRAM LOADING PROCEDURE

The procedure for loading FORTRAN object programs is:

1. Insert system tape in the paper tape reader. Set brake. (It is assumed that the user has just finished compilation and the system tape is at the beginning of the loader. Otherwise, mount the tape and advance it to the second block.)
2. Turn spooler on.
3. Load the FORTRAN loader by the standard fill procedure. When the loader is completely read in, the message

   LOAD MAIN PROGRAM.

   is typed and the computer halts.
4. Turn spooler off.
5. Remove system tape from reader.
6. Insert main object program in reader. Set brake.
7. Select the desired FORTRAN loader output options by setting the appropriate breakpoint switches.

Switch 1	RESET	Output on typewriter
	SET	Output on printer
Switch 2	RESET	No program maps
	SET	Produce a map of all programs and subprograms loaded.
Switch 3	RESET	No label maps
	SET	Produce a map of all statement labels used in FORTRAN-compiled programs or subprograms.
Switch 4	RESET	No label trace
	SET	Load the special 160 POP. The 160 POPs are produced by the compiler and, if loaded, cause the statement number of a labeled statement to be printed each time that statement is executed during run-time.

8. Clear the halt to read in program. When the main program has been read in, the message

   LOAD SUBPROGRAMS.

   is typed. If the main program requires FORTRAN subprograms that are not in the library, read them in using steps 6 through 8.
9. Replace system tape in the reader. Set brake.
10. Turn spooler on.
11. Clear the halt to read in the library. When the library routines have been read in, the loader proceeds to read in the run-time program, unless there are still some subprograms required (see below). When it is completely loaded, the message

    LOADING COMPLETE

    is typed and the computer halts.
12. Clear the halt to begin execution of the object program.

The loader makes no distinction between library subroutines and those written by the user in FORTRAN or META-SYMBOL/SYMBOL.

Only those subprograms that have been called for are accepted by the loader. All others are ignored.

If two or more subprograms with the same name are presented to the loader, the first one is accepted and subsequent one(s) ignored.

If two subprograms, A and B, both call a third, C, either A or B should precede C in order that C be called before it is read. It is not necessary for both A and B to precede C.

An attempt to load a tape that is not a legitimate object program causes reading to halt and the message

```
ILLEGAL INPUT. RELOAD PROGRAM.
```

to be typed.

If a reading error occurs during loading, the message

```
READ ERROR. RELOAD LAST RECORD.
```

is typed and the computer halts. The tape should be moved back to the nearest gap and loading continued by clearing the halt.

If the memory is exceeded during loading, the message

```
PROGRAM TOO BIG.
```

is typed; the loader simulates loading into an infinite memory.

MISSING SUBPROGRAMS

If the operator has neglected to load all required subprograms before reading in the library, the computer types

```
MISSING SUBPROGRAMS
XXXXXXXX
   .
   .
   .
```

Where XXXXXXXX represents a subprogram name listing of each of the required subprograms that have not been found in the library. The operator should provide the requested subprograms.

The message

```
LOAD SUBPROGRAMS.
```

will continue to be typed out until all required subprograms have been loaded. Some of them may be library subprograms and the system tape may have to be repositioned and the library read in again.

When no further subprograms are required, the computer types

```
LOAD RUN TIME.
```

The operator should remount the system tape (after the library block) and clear the halt to read in the run-time package.

Note: Run-time may also be loaded at any time using the standard fill procedure. The library, however, must be read in by the loader, and the loader and compiler may only be loaded by the standard fill procedure.

LOADER HALTS

All halts during loading are accompanied by a display in the C register of a "tagged" NOP plus a number to indicate the type of halt. These are:

22001000	Load system
22001001	Load subprograms
22001002	Read error
22001003	Load main program

LOADER RESTART

The standard restart procedure (see Appendix A) may be used at any time to reinitialize the loader to step 6 above.

OBJECT PROGRAM RESTART

The standard restart procedure (see Appendix A) may be used at any time after completion of all loading, including run-time, to reinitialize to step 12 above.

FORTRAN LOADER'S OUTPUT

During the loading of an object program, the FORTRAN loader outputs memory maps describing program storage allocation at execution time. The breakpoint switches are used to select the various output options. The breakpoint settings dynamically control the output in the sense that they can turn the various options on or off, when desired, as the loading proceeds.

An abbreviated example of the FORTRAN loader's output is given below.

```
NAME          ENTRY     ORIGIN     LAST     SIZE/10    COMMON    BASE

#    7        03522
#    2        03525
#  232        03547
# 7232        03553
.
.
.
$$$$$$$$      03462     03452      05224     875                 04643
ABSF          05226     05225      05242      14
203SYS        05244     05243      05254      10
.
.
.
*PROGRAM      03462     03452      06101    1304
```

If a label map is requested, it is output immediately below the headings. If a storage map is also requested, it is output following the label map.

Program names or statement numbers appear under the heading NAME.

The entry, *PROGRAM, is always printed; it identifies the line containing the total program storage information. $$$$$$$$ is the compiler-assigned identification for the main program.

The entry location to the program or to the code interpreting a labeled statement appears under the heading ENTRY. Under the remaining headings appear the location of the program origin, the last location occupied by the program, the program size (in decimal notation), the beginning location of COMMON (when applicable) and the base location for determining the location of variables used in a FORTRAN-compiled program.

The value in the column BASE is used to determine the absolute location of variables. At compilation time, variables are assigned locations relative to the end of the program; it is these relative locations that are printed as "Program Allocation" by the compiler. To determine the absolute location of a given variable, add its relative location to the value listed by the FORTRAN loader as "base" for the program containing that variable. For example, assume the variable J was assigned relative location 55. Using the base shown above, the absolute location of J is determined by adding 04643 and 55, which results in 04720.

4. LIBRARY

The library functions are described in this section in the order they appear on the system tape, giving for each:

Preferred name of function
Other acceptable names, if any
Number and mode of arguments
Function performed
Memory required (in words)
Accuracy, where applicable
Timing, for normal cases (in microseconds unless otherwise indicated)

All library functions are closed subroutines; that is, they appear only once in the object program.

ELEMENTARY MATHEMATICAL AND MISCELLANEOUS FUNCTIONS

230SYS. Raises a number to a power. Cannot be explicitly called by the programmer. This routine is called implicitly by the presence of "**" in the source program, and it requires ALOG and EXP. When the power is 2, the first argument is multiplied by itself.

Accuracy: Integer arguments, exact
Floating arguments (except 2.0), see ALOG and EXP

		910	925	920	930
Memory:		113	113	106	106
Timing:	I**N	648(N)-352	142(N)-77	88+144(N)	20+32N
	A**2	5050	1105	1150	252
	A**B	60 ms	13.2 ms	12.5 ms	2.8 ms

ALOG – ALOGF, ELOG, ELOGF. Computes the natural logarithm of a floating-point argument.

Accuracy: $|\ln x| \geq 1$: relative error $< 6 \times 10^{-11}$
$|\ln x| < 1$: absolute error $< 6 \times 10^{-11}$

	910	925	920	930
Memory:	137	137	138	138
Timing:	32 ms	7 ms	5.9 ms	1.3 ms

EXP – EXPF. Computes the exponential (base e) of a floating-point argument.

Accuracy: relative error $< 6 \times 10^{-11} \times 2^{\max[0, (\log_2|X| + 1)]}$

	910	925	920	930
Memory:	163	163	147	147
Timing:	21 ms	4.6 ms	6.5 ms	1.5 ms

SIN – SINF. Computes the sine of a floating-point argument in radians.
COS – COSF. Computes the cosine of a floating-point argument in radians.

Accuracy: relative error $< 6 \times 10^{-11}$ + error arising from loss of significance in the argument (X) as X becomes large and as X approaches the zeros of sin X (cos X)

	910	925	920	930
Memory:	201	201	204	204
Timing:	30 ms	6.6 ms	5.1 ms	1.2 ms

SQRT – SQRTF. Computes the square root of a floating-point argument.

Accuracy: relative error $< 10^{-11}$

	910	925	920	930
Memory:	98	98	83	83
Timing:	3.9 ms	0.9 ms	1100	240

ATAN – ATANF. Given two floating-point arguments, Y and X, the routine computes the arctangent of Y/X, allocating the result in radians to the proper quadrant. The range of this function is $-\pi \leq \arctan < \pi$.

Given one floating-point argument, Y, the routine assumes X = 1.0.

Accuracy: relative error $< 6 \times 10^{-11}$

	910	925	920	930
Memory:	259	259	256	256
Timing:	29 ms	6.4 ms	8.3 ms	1.9 ms

ABS – ABSF. Floating-point absolute value of a floating or integer argument.

	910	925	920	930
Memory:	13	13	13	13
Timing: (floating-point argument)				
positive	184	41	184	41
negative	866	190	320	70

For integer argument, add FLOAT time.

IABS – IABSF. Integer absolute value of an integer or floating argument.

	910	925	920	930
Memory:	13	13	13	13
Timing: (integer argument)				
positive	120	27	120	27
negative	184	41	120	27

For floating point argument, add fix time.

FLOAT – FLOATF. Converts integer argument to floating-point.

	910	925	920	930
Memory:	4	4	4	4
Timing: (+ normalize time)	328	72	152	34

IFIX – IFIXF, INT. Truncates floating-point argument to integer. Positive and negative arguments are both truncated towards zero.

	910	925	920	930
Memory:	8	8	8	8
Timing:	272-1784	60-391	144-624	32-137
	1000 avg.	220 avg.	376 avg.	83 avg.

AINT – AINTF. Truncates floating-point argument to integer, then floats the integer.

	910	925	920	930
Memory:	8	8	8	8
Timing:	Add FLOAT time to IFIX time above.			

SIGN – SIGNF. The algebraic sign of the second argument is assigned to the first argument. Each argument may be of either mode, but the result will be in floating-point.

	910	925	920	930
Memory:	21	21	21	21
Timing: (plus FLOAT time if necessary)	560-1940	125-425	400-690	90-150

ISIGN – ISIGNF. The algebraic sign of the second argument is assigned to the first argument. Each argument may be of either mode, but the result will be in integer form.

	910	925	920	930
Memory:	20	20	20	20
Timing: (plus IFIX time if necessary)	216-352	48-77	224	49

AMOD – AMODF. Requires two floating-point arguments. Returns the remainder when the first is divided by the second, i.e., AMOD(A, B) = A - FLOAT (IFIX(A/B))*B.

	910	925	920	930
Memory:	13	13	13	13
Timing: (plus fix time)	9.2 ms	2.0 ms	3.7 ms	0.8 ms

MOD. Requires two integer arguments. Returns the remainder when the first is divided by the second, i.e., MOD (I, J) = I - (I/J)*J.

	910	925	920	930
Memory:	9	9	9	9
Timing:	1336	293	408	90

The following routines use a common loop that finds the maximum or minimum of any number of arguments, each of which may be of either mode. Each argument is converted to floating-point before comparing. The resulting maximum or minimum is then fixed, if necessary.

AMAX – AMAX0, AMAX1. Floating-point maximum.
MAX – MAX0, MAX1. Integer maximum.
AMIN – AMIN0, AMIN1. Floating-point minimum.
MIN – MIN0, MIN1. Integer minimum.

	910	925	920	930
Memory:	65	65	65	65
Timing: each argument (plus FLOAT time if necessary)	670 +3.5 ms	150 +0.8 ms	500 +1.5 ms	110 +0.4 ms

DIM – DIMF. Requires two floating-point arguments. Returns difference if first greater than second; otherwise, zero, i.e., DIM(A, B) = AMAX(A-B, 0.0)

Note also that AMAX(A, 0.0) = DIM(A, 0.0)
AMIN(0.0, A) = -DIM(0.0, A)
and the DIM routine is much shorter, if this result is needed.

	910	925	920	930
Memory:	10	10	10	10
Timing:	3.0 ms	0.8 ms	1.2 ms	0.3 ms

IDIM – IDIMF. Requires two integer arguments. Returns difference if first greater than second; otherwise, zero, i.e., IDIM(I, J) = MAX(I-J, 0)

	910	925	920	930
Memory:	10	10	10	10
Timing:	144	32	120	27

LOCF. Returns the absolute address of an argument of either mode. This is useful in conjunction with dump routines.

	910	925	920	930
Memory:	4	4	4	4
Timing:	56	13	56	13

IF. Given two floating-point arguments, P and Q, this function returns zero if they are equal to within the four low order mantissa bits; otherwise, it returns an integer with the sign of P-Q.

Given one floating-point argument, P, the function returns zero if it is of magnitude less than 10^{-10}; otherwise, it returns an integer with the sign of P.

The IF function is most useful in conjunction with the IF statement to provide a means of testing equality of decimal numbers in binary.

	910	925	920	930
Memory:	31	31	25	25
Timing: (for one argument)	624	156	432	102
(for two arguments)	3.4 ms	0.8 ms	1.4 ms	0.3 ms

EXIT. Same effect as STOP statement, except that it types *EXIT* and branches to a transfer point. This provides compatibility with 7090 Monitor FORTRAN programs.

	910	925	920	930
Memory:	11	11	11	11

SYSTEM ROUTINES

These routines are not called by name; the compiler sets up references to them by using octal numbers ranging from 201 to 244. The linkage to them is stored in locations 201 - 244 and they are entered by a BRM* instruction. Only those routines called for implicitly in the program are actually loaded.

The system routines are listed in this section, giving for each:

Octal number
Name
Operation performed
Memory storage used
Other system routines required, if any

160SYS. Label Trace POP. Outputs the statement number when a labeled statement is executed at run-time.

Memory: 14 words
Requires: 211SYS and 223SYS

201SYS. Start of dummies. Used by FORTRAN subprograms in obtaining arguments from the calling program.

Memory: 4 words

202SYS. End of dummies. Used in conjunction with 201SYS in obtaining arguments.

Memory: 8 words

203SYS. Stop. Types *STOP* and halts.

Memory: 10 words

204SYS. If Sense Switch. Performs the If Sense Switch test.

Memory: 21 words

205SYS. If Sense Light. Performs the If Sense Light test.

Memory: 20 words

206SYS. Computed GO TO. Performs the Computed GO TO.

Memory: 11 words

207SYS. Accept. Initializes for reading information from the console typewriter.

Memory: 9 words
Requires: 234SYS and 235SYS

210SYS. Accept Tape. Initializes for reading information from paper tape.

Memory: 9 words
Requires: 234SYS and 235SYS

211SYS. Print. Prints on the line printer. Not in the standard library. See SDS Catalog No. 062001 or 062005.

Memory: 43 words
Requires: 235SYS

212SYS. Punch. Punches BCD cards. Not in the standard library. See SDS Catalog No. 032001.

Memory: 50 words
Requires: 235SYS

213SYS. Punch Tape. Initializes to punch paper tape. This routine will also be called by the "PUNCH" statement if the PUNCH routine (212SYS) is not loaded first.

Memory: 10 words
Requires: 235SYS and 240SYS

214SYS. Type. Initializes to type on the console typewriter. This routine will also be called by the "PRINT" statement, if the PRINT routine (211SYS) is not loaded first.

Memory: 10 words
Requires: 235SYS and 240SYS

215SYS. Rewind. Rewinds magnetic tape.

Memory: 6 words
Requires: 242SYS and 244SYS

216SYS. Read. Reads BCD cards. If card reader is not ready, waits 15 seconds, then types ERR CRDS. Continues to type this at 5-minute intervals until reader is made ready.

Memory: 53 words
Requires: 235SYS and 236SYS

217SYS. Read Tape. Initializes for reading magnetic tape in binary mode.

Memory: 5 words
Requires: 241SYS and 242SYS

220SYS. Read Input Tape. Reads from magnetic tape in BCD mode.

Memory: 88 words
Requires: 235SYS, 236SYS, 242SYS, and 244SYS

221SYS. Write Tape. Initializes for writing magnetic tape in binary mode.

Memory: 5 words
Requires: 241SYS and 242SYS

222SYS. Write Output Tape. Writes magnetic tape in BCD mode.

Memory: 70 words
Requires: 235SYS, 242SYS, 243SYS, and 244SYS

223SYS. End Input/Output List. Used by all input/output lists.

Memory: 12 words

224SYS. IF Overflow. Tests status of floating-point overflow and branches accordingly.

Memory: 6 words

225SYS. Backspace. Backspaces magnetic tape one logical record.

Memory: 45 words
Requires: 242SYS and 244SYS

226SYS. End File. Writes an end-of-file mark on magnetic tape.

Memory: 33 words
Requires: 242SYS, 243SYS, and 244SYS

227SYS. Sense Light. Sets Sense Light.

Memory: 19 words

230SYS. Power. See Elementary Mathematical and Miscellaneous Functions at beginning of this section.

231SYS. Fix. Converts floating-point number to integer.

Memory: 3 words

232SYS. Float. Converts integer to floating-point number.

Memory: 3 words

233SYS. Input/Output List Unscripted Array. Used during input and output of arrays when listed without subscripts, (e.g., TYPE 3, A).

Memory: 28 words

234SYS. Accept/Accept Tape. Used with 207SYS and/or 210SYS for inputting information from the console typewriter and/or from paper tape.

Memory: 68 words
Requires: 236SYS

235SYS. Initialize Format Scan. Used by the input/output system routines to initialize the FORMAT scan.

Memory: 78 words

236SYS. BCD to binary. Used by the FORMAT scan routines to convert BCD numbers to binary during input.

Memory: 220 words(920/930); 238 words(910/925)

240SYS. Punch/Type. Used with 213SYS and/or 214SYS for outputting information on paper tape and/or on the console typewriter.

Memory: 46 words

241SYS. Read/Write Tape. Used with 217SYS and/or 221SYS for reading and/or writing of binary information on magnetic tape.

Memory: 367 words
Requires: 243SYS and 244SYS

242SYS. Set Up Input/Output Table. Selects proper tape unit and sets up constants preparatory to all operations involving magnetic tape.

Memory: 58 words

243SYS. Test Write. Checks if ready to write on magnetic tape. Writes a leader if at beginning of tape.

Memory: 45 words
Requires: 244SYS

244SYS. Tape Ready. Checks tape before all magnetic tape operations to assure that it is selected and ready. If unit is not ready within 3 minutes, 17 seconds (the time required for a full-reel rewind), the program types ERR TNR# (see RUN-TIME MAGNETIC TAPE ERRORS). These typeouts recur at the same time interval until the tape unit is made ready.

Memory: 20 words

5. RUN-TIME

PROGRAMMED OPERATORS

SDS FORTRAN II incorporates a set of special-purpose programmed operators designed particularly for FORTRAN programs.

100	XSD	Fixed Setup Dummy
101	FSD	Floating Setup Dummy
114	XFA	Fixed First Argument
115	FFA	Floating First Argument
116	XNA	Fixed Next Argument
117	FNA	Floating Next Argument

The purpose of the XSD and FSD POPs is to procure one address from erasable storage, to store that address in absolute form in the location specified by XSD (or FSD), and to store that address with an index bit of one in the specified location + 1.

XSD and FSD are used by FORTRAN subroutines to locate the address(es) of the argument(s) specified by a CALL statement or a function call in a FORTRAN program. These POPs are used in conjunction with several additional run-time POPs and subroutines whose functions are described below.

Suppose a FORTRAN program contains the following statement:

```
    CALL    FIND(A, N)
```

The machine language code generated by the compiler for this call would be

```
    FFA     A
    XNA     N
    BRM     FIND
```

FFA (Floating First Argument) is run-time POP 115 which places the address of the variable A in the first location, E0, of erasable storage. Since it is the first address to be placed in erasable, FFA also initializes EADR1 to the address, E0, (found in E0ADR) and then increments EADR1. Thus, EADR1 will thereafter contain the next available address in erasable storage. Since the argument is in floating-point, bit 5 of the word placed in erasable is set to one.

XNA (Fixed Next Argument) is run-time POP 116 which places the address of the variable N into the next available location of erasable storage. Since the argument is in fixed-point, bit 5 of the word placed in erasable is set to zero. EADR1 is incremented before leaving.

The machine language code generated at the start of SUBROUTINE FIND (A, N) would be

```
    FIND    PZE
            BRM     201SYS
            FSD     TEMP
            XSD     TEMP+2
            BRM     202SYS
```

Now, the appearance of the variable names A and N in the calling program requires the compiler to allocate storage for these quantities. Since storage for these quantities has already been set aside in the calling program, doing so again in the subroutine would have no meaning. Therefore, the appearance of these names in the subroutine serves only to indicate to the subroutine that there are two arguments, the first of which is floating-point and the second of which is integer. For this reason, variablesappearing in the argument list of a subroutine are called DUMMIES.

201SYS is a library subroutine whose function is to initialize EADR2 to the address in E0ADR. EADR2 is then used by the subroutine to point to that address in erasable where the next argument address may be obtained.

FSD (Floating Setup Dummy) is run-time POP 101 which procures the address from the erasable location specified by EADR2 and places that address and that address, tagged, in (in this case) TEMP and TEMP+1. Bit 5 of this quantity is checked for a one. If bit 5 is not set to one, there is a disagreement of variable mode between the main program and the subroutine and the run-time error message ERR ARGM is typed out. EADR2 is incremented before leaving.

XSD (Fixed Setup Dummy) is run-time POP 100 which procures the address from the erasable location specified by EADR2 and places that address and that address, tagged, in (in this case) TEMP + 2 and TEMP + 3. Bit 5 of this quantity is checked for a zero. If bit 5 is not set to zero, there is a disagreement of variable mode between the main program and the subroutine and the run-time error message ERR ARGM is typed out. EADR2 is incremented before leaving.

202SYS is a library subroutine whose function is to compare EADR1 and EADR2. EADR1 indicates how many addresses were placed into erasable by the calling program and EADR2 indicates how many were taken out by the subroutine. If they are not equal, there is a discrepancy in the number of arguments and the run-time error message ERR ARGN is typed out.

Note that it is absolutely necessary to initiate this procedure at the beginning of every subroutine to preserve those addresses that have been placed in erasable storage. If the first statement of the subroutine had been another CALL, the setup would destroy the original addresses placed there by the main program.

FFA and FNA double the contents of the index register before determining the effective address of an argument.

110 DOX DO Fixed

111 DOF DO Floating

These run-time POPs are generated by FORTRAN DO statements. The POP adds the increment to the variable and compares it with the limit and skips if the DO loop is not finished.

Example:

```
DO  3 X = A, B, C
```

The coding generated is:

```
      LDP    A
      STD    X
      BRU    L2
L1    LDP    C    (increment)
      DOF    B    (limit)
      PZE    X    (variable)
      BRU    L3   finished
L2                loop
       .           .
       .           .
       .           .
      BRU    L1
L3                finished
       .
       .
       .
```

Example:

```
DO  3 I = M, N, J
```

The coding generated is:

```
      LDA    M
      STA    I
      BRU    L2
L1    LDA    J    (increment)
      DOX    N    (limit)
      PZE    I    (variable)
      BRU    L3   finished
L2                loop
       .           .
       .           .
       .           .
      BRU    L1
L3                finished
       .           .
       .           .
       .           .
```

An exception is:

```
    DO  3 I = M, N
```

The coding generated for this case, where the increment is understood to be one, is as follows:

```
         LDA    M
         STA    I
         BRU    L1
    L2   MIN    I
         LDA    N
         ADD    ONE   standard constant
         SKG    I
         BRU    L3
    L1                loop
          .            .
          .            .
          .            .
         BRU    L2
    L3                finished
          .            .
          .            .
          .            .
```

150 ALX   Assign Label to Fixed

```
    ASSIGN 3 TO M

    ALX    M
    BRU    (3)    Address of statement 3
```

The ALX puts its own address in M, e.g., if the ALX instruction is executed in location 3460, M contains 3460. Following the ALX POP is a BRU to the start of the assigned statement.

112 AGX   Assigned GO TO Fixed

```
    GO TO M
```

This POP checks the address in M. In this address, there should be an ALX POP (or ALF POP) showing that a statement label was assigned. If so, it transfers control to the location after the one specified by the variable. That location should contain a BRU to the assigned statement. If the word at the address specified by the variable is not an ALX or ALF POP, the message ERR AGTO is typed and the computer halts.

Mode is not checked.

151 ALF   Assign Label to Floating

Similar to ALX but doubles the index register before determining the effective address of the argument.

113 AFG   Assigned GO TO Floating

Similar to AGX but doubles the index register before determining the effective address of the argument.

120 XIO   Fixed Input/Output

This POP communicates with the FORMAT processor to transmit the address of an integer input/output argument.

Use is similar to a MIW or WIM in machine language.

Example:

```
FORTRAN - Generated Code     Machine Language

BRM      PRINT               EOM
PZE      FORMAT
  .                            .
  .                            .
XIO      ARG1                MIW/WIM  ARG1
  .                            .
  .                            .
XIO      ARG2                MIW/WIM  ARG2
  .                            .
  .                            .
BRM      ENDIOL              TOP0/DSC0
```

121 FIO Floating Input/Output

Similar to XIO but doubles the index register before determining the effective address of a floating, input/output argument.

DOUBLING THE INDEX REGISTER

The compiler handles subscripted variables in the following manner. If XXX is the base address of a floating-point array and i is the value of the subscript, the location of any variable can be found by:

$$\text{LOC} = \text{XXX} + 2(i - 1) = \text{XXX} - 2 + 2i$$

Multiplying the subscript by two is necessary because two locations are used for each floating-point variable.

The compiler calculates a basic address, YYY = XXX - 2, and generates code similar to the following

```
LDX     I
POP     YYY,2
```

If the POP is a floating run-time POP, it will double the index before execution and restore the original value after. Thus, part of the array indexing is done by the compiler in calculating the basic address YYY, and part is done by the floating-point run-time POP by doubling the index.

As for integer arrays, the location of a variable is given by XXX + (I - 1), or XXX - 1 + I. The basic addressas calculated by the compiler would be YYY = XXX - 1.

Fixed-point run-time POPs do not double the index register.

125 LDP Load Double Precision

Loads the B and A registers with the contents of Memory and Memory + 1, respectively.

Doubles the index register before determining the effective address of the argument.

107 STD Store Double Precision

Stores the contents of the B and A registers in Memory and Memory + 1, respectively.

Doubles the index register before determining the effective address of the argument.

105 FST Float and Store

Floats the integer in A and stores the contents of the B and A registers in Memory and Memory + 1, respectively.

Doubles the index register before determining the effective address of the argument.

126 FTA Float then Add

Floats the integer in memory and then adds it to A, B.

132 FTS Float then Subtract

Floats the integer in memory, then subtracts it from A, B.

136 FTM Float then Multiply

Floats the integer in memory, then multiplies it by A, B.

142 FTD Float then Divide

Floats the integer in memory and divides A, B by it.

104 XST Fix and Store

Fixes the floating-point number in A, B and stores it in memory.

122 LTF Load then Float

Loads A with an integer in memory and then floats it.

123 LTX Load then Fix

Loads A, B with the floating-point number in memory and fixes it, leaving the integer result in A.

The index register is doubled before determining the effective address of the argument.

131 FLA Floating Add

135 FLS Floating Subtract

141 FLM Floating Multiply

145 FLD Floating Divide

Perform the indicated floating-point operation with memory and the A, B register; the result is left in A, B. The floating-point number in memory is reversed. The high-order part is in M+1 and the low-order part and the exponent is in M.

The index register is doubled before determining the effective address of the argument.

147 FLN Floating Negate

Negates the floating-point number in A, B. The effective address of the POP is ignored.

154 DPA Double-Precision Add

153 DPS Double-Precision Subtract

155 DPM Double-Precision Multiply

Performs the indicated machine operation. Treats the A, B register as one register. The number in memory is stored in reverse order as is the case with floating-point numbers. M + 1 contains the most significant and M contains the least significant part of the number.

These POPs are not generated by the compiler, but are used by the run-time package and the library.

140 XMP Fixed Multiply

Multiplies the integer contents of A by memory and puts the resulting integer in A. This is an integer multiply with the binary point at 23.

144 XDV Fixed Divide

Divides the contents of A by the integer in memory and puts the resulting integer in A. The integer remainder is left in B. This is an integer divide with the binary point at 23.

All of the floating-point POPs can set the overflow indicator. The results given in an overflow are the maximum value of the variable with the proper sign. An underflow returns a zero result.

RUN-TIME HALTS AND ERRORS

The following conditions cause suspension of FORTRAN object program execution. In some cases, execution is resumed immediately after a typeout.

STOP Statement. When this statement is executed, the computer will type:

```
*STOP*
```

and will not continue.

CALL EXIT Statement. When this statement is executed, the computer will type:

```
*EXIT*
```

and branch to location 1. This will cause "LOADING COMPLETE" to be typed out and the computer will halt. The statement is provided primarily to allow linkage with other (e.g., monitor) systems.

PAUSE Statement. When this statement is executed, the computer will halt and display the integer constant, if any, in the C register.

ERROR Conditions. There are a number of conditions which cause error typeouts of the form:

```
ERR XXXX
```

Following the typeout the computer will either halt or take remedial action and continue. The following table indicates, for each of the errors,

Message typed.

Whether a halt occurs.

Cause of the error.

Contents of registers at time of halt, if such information may be useful or if it may be changed before proceeding.

Result if program is allowed to continue.

RUN-TIME ERRORS

Message	Halt	Explanation
AGTO	X	Assigned GO TO – Variable never assigned. Variable displayed in X. Result: Branch to (effective address determined by variable) + 1.
ARGM	X	Argument Mode – Argument of wrong mode given to FORTRAN subprogram. Proper mode is fixed if A = 0, floating if A = 01000000. Dummy address of argument displayed in X. Result: Argument used as if its mode were correct.
ARGN		Argument Number – Wrong number of arguments given to FORTRAN subprogram. Result: If too many, extra ones ignored. If too few, whatever arguments remain in erasable storage will be used.
CARD	X	Card "READ CHECK" or "FEED CHECK" error – If "READ CHECK" light is on, the last card read was in error. Place it back in the hopper. If "FEED CHECK" light is on, the offending card is still in the hopper. It probably has a wrinkled leading edge. Result: Try to read the card again.
CGTO		Computed GO TO – Value outside allowable range. Result: Go to first statement number in list.
CRDS		Card Reader Not Ready – Program has waited 15 seconds for reader. Place cards in reader and press start. Result: Program continues to wait for reader. Typeouts occur in 5-minute intervals.
EFIA		E, F, I, or A Needed in FORMAT – Unable to output variables. Result: Proceeds without outputting variables.
EIOL	X	End Input/Output List encountered without prior initialization. Result: Proceeds without taking any I/O action.
EXP		Exponential Function – Argument greater than 176. Result: Answer set to maximum floating-point value.
FCHt	X	FORMAT Character Illegal – The illegal character is displayed as the fourth character (t) in the message typed out. Result: Begins scan for next specification, i.e., treats character as if it were a comma.
FORL	X	FORMAT Label Error – The scalar variable referenced by an I/O statement has not been assigned a FORMAT statement label. Result: The contents of the (effective address determined by the variable) + 1 is used as the address of the start of the FORMAT statement.
FORM	X	FORMAT Missing – I/O statement references something else. X = address of supposed FORMAT. A = first word of supposed FORMAT. Result: Scans supposed FORMAT.
FORP	X	FORMAT Pointer Error – The address in the I/O list pointing to the FORMAT statement is not in an acceptable form, i.e., HLT, BRU or AGT (112 or 113). X = address of pointer. A = bad pointer. Result: Pointer at address specified by X is treated as if it had form specified by a HLT.
FXIO	X	Floating or Fixed Data requested for I/O without prior initialization. Result: Proceeds without taking any I/O action.
ICHt	X	Input Character Illegal – The illegal character is displayed as the fourth character (t) in the message typed out. Result: Begins scan for next field, i.e., treats character as if it were a comma.
IFSL		If Sense Light – Value not 1-24. Result: Assume sense light off.
IFSS		If Sense Switch – Value not 1-4. Result: Assume sense switch off.

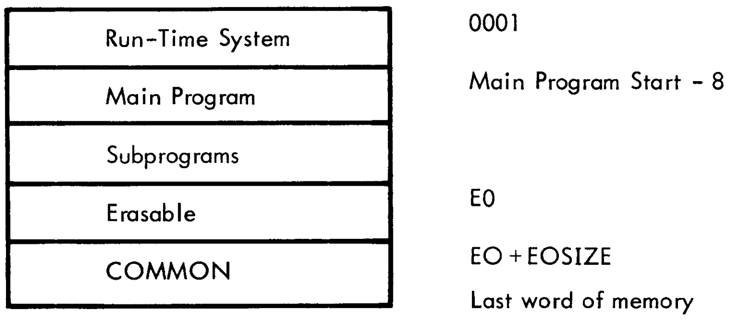
RUN-TIME ERRORS (cont.)

Message	Halt	Explanation
INOV		Integer Overflow – Input value of integer quantity exceeds 8, 388, 607. Result: Number truncated to the least significant 24 bits.
LABL	X	Label Undefined – Result: Computer will not proceed.
LCRD	X	Last Card Read in Error – May or may not be caused by a validity check. Place the last card back in the hopper. Result: Try to read the card again.
LOG		Logarithm Function – Argument negative or zero. Result: Answer set to zero.
NO[	X	No left parenthesis in FORMAT statement – Result: Computer will not proceed.
N**F		Negative Number Raised to Nonintegral Power. Result: Computes (|N| **F).
OCTt		Non-Octal Character (t) encountered during input under octal FORMAT specification. Result: Character is truncated to 3 least significant bits.
PNCH		Card Punch Not Ready – Program has waited 15 seconds for punch. Make the punch ready. Result: Continues to wait. Typeouts occur in 5-minute intervals.
PRNT		Printer Not Ready – Program has waited 15 seconds for printer. Make the printer ready. Result: Continues to wait. Typeouts occur in 5-minute intervals.
PRTY	X	Parity Error During Input – Result: Processing continues using incorrect character.
REP[		Repeat Count Precedes Outermost [in FORMAT – Result: Where applicable, group repeat count is applied to entire FORMAT specification.
REP$		Repeat Count Precedes $ in FORMAT – Result: Repeat count is ignored.
SIZE	X	Size of Erasable Storage Exceeded – There is no unused memory in which to transfer arguments to subroutines. Result: Erasable storage will run into COMMON, if any, or out of memory.
SNLT		Sense Light – Value not 0-24. Result: Statement has no effect.
SQRT		Square Root Function – Argument negative. Result: Square root of absolute value of argument.
XPOV		Exponent Overflow on Input Datum – Result: List item set to positive maximum (approximately .579 x 10^{77}).
XPUN		Exponent Underflow on Input Datum – Result: List item set to zero.
0**N		Zero Raised to Nonpositive Power – Result: (0**0) will be 1 or 1.0, and (0**NEGATIVE) will be the maximum possible integer or floating number, as the case may be.
[OVF	X	Nesting Level Exceeded – Limit on number of parenthesized groups of FORMAT specifications is normally 4 levels. Result: Higher levels of nesting are disregarded.

RUN-TIME MAGNETIC TAPE ERRORS

Message	Halt	Explanation
TPNO	X	Tape Number Not 0-7 – Tape number displayed in A. Result: Number will be truncated and the low-order octal digit (0-7) will be used. For the remainder of the tape errors, the tape unit in error will be indicated as the fourth character (#) of the message typed out.
BKS#	X	Backspace – Failed 10 Times. Result: Proceed as if backspace has successfully taken place.
EOF#	X	End of File Reached During Reading – Result: Continue to read past end of file.
ETR#	X	End of Tape While Reading – Remove the finished tape and replace with next reel. Result: Continue reading.
ETW#	X	End of Tape While Writing – Remove the finished tape and replace with next reel. Result: Continue writing. This, in conjunction with ETR, facilitates writing and reading of multiple reels.
FPT#	X	File Protect – Attempted to write on tape which is file protected. Result: Check again.
LRR#	X	Long Record Read – READ TAPE (binary) has read a logical record which contains more information than is required by the I/O list. Result: The remainder of the record is skipped.
RDT#	X	Read Tape Error – Failed to read 10 times. Result: Proceed assuming read to have been satisfactory.
SRR#	X	Short Record Read – READ TAPE (binary) attempting to read more information from a logical record than is present. Result: Remaining items in the I/O list are supplied with words of zero.
TNR#		Tape Not Ready – Program has waited 3 minutes, 17 seconds for tape unit. Ready the tape unit. Result: Program continues to wait for tape unit. Typeouts occur in 3 minute, 17 second intervals.
WEF#	X	Write End of File Error – Result: Try again.
WRT#	X	Write Tape Error – Failed to write 5 times. Result: Proceed, assuming write to have been satisfactory.

MEMORY LAYOUT AT RUN-TIME



Main Program Start

The entrance point to the main program may be determined from the branch instruction in location 00400. Run-Time ends ten locations below this to allow room for the heading. (See below.)

E0

The starting address of erasable storage is contained in E0ADR in 00071.

E0SIZE

The amount of unused memory is contained in 00072.

The address of the first word of each subprogram is contained in the first word of the preceding subprogram. (See Heading, below.)

Each FORTRAN-written program consists of the following segments:

1. Heading

   This consists of nine words.

PP0AAAAA	where AAAAA is the beginning of the next program, and PP determines program type: PP = 10 for machine language subprograms; PP = 04 for the main program; PP = 02 for subprograms.
NNNNNNNN	These two words give the BCD name of the program.
NNNNNNNN	For the main program, this is all dollar signs. (OCT 53)
000CCCCC	Address of beginning of linkage table. (See Linkage Table, below.) Note: This is the base address to which the relative addresses of the variables (given at compile time under program allocation) are added to obtain their absolute memory locations.
000DDDDD	Address of beginning of dummies and temporary storage locations.
000EEEEE	Address of beginning of equivalenced variables. (See EQUIVALENCEd Variables.)
000FFFFF	Address of beginning of arrays. (See Arrays.)
000GGGGG	Address of beginning of fixed scalar variables. (See Integer Scalar Variables.)
000HHHHH	Address of beginning of floating scalar variables. (See Floating Scalar Variables.)

2. Text

   The instructions and constants which comprise the program.

3. Linkage Table

   Transfer vector to subroutines called.

4. Dummies and Temporary Storage

5. EQUIVALENCEd Variables

   All variables, including arrays, which appear in EQUIVALENCE statements. These are stored in the order in which they appear in EQUIVALENCE statements.

6. Arrays

   Stored independent of mode in the order in which they appear in DIMENSION statements. All arrays are stored forward in memory, e.g., if M(1) is in location 4000, M(2) would be in 4001.

7. Integer Scalar Variables

   Stored in the order of their appearance in the source program.

8. Floating Scalar Variables

   Stored in the order of their appearance in the source program.

Subprograms written in SYMBOL/META-SYMBOL are preceded in memory by only one word, a pointer to the beginning of the next program. The external definitions and references of the SYMBOL/META-SYMBOL programs (defining their names and required subroutines) are used only by the loader at load-time and are then discarded. (See Machine Language FORTRAN Subroutines.)

The use of words 1-4 in the heading of FORTRAN programs enables the user to locate programs and variables readily by name. This process is handled automatically by the FORTRAN Run-Time Debug utility routine (Catalog No. 012001). In addition, the user may obtain a memory map of the layout at run-time by selecting this option in the FORTRAN loader. (See FORTRAN Loader's Output.)

ARRANGEMENT OF VARIABLES IN COMMON

Variables are stored in COMMON in exactly the order in which they appear in COMMON, starting at the end of memory and working back. EQUIVALENCE is not allowed to affect the order or spacing of COMMON. Variables which are equivalenced to variables in COMMON will simply overlay COMMON. Thus, the arrangement of COMMON is determined solely by the COMMON statements. Consider the following examples:

DIMENSION B(3)	77777	I		
COMMON I, A, B, J	77775	A		
	77773	B(3)		
	77771	B(2)		
	77767	B(1)		
	77766	J		
DIMENSION B(3)	77776	A	B(3)	
COMMON A, C, D, I	77774	C	B(2)	Q
EQUIVALENCE (C, Q), (D, B)	77772	D	B(1)	
	77771	I		
DIMENSION B(3)	*		B(3)	
COMMON A, C, D	77776	A	B(2)	
EQUIVALENCE (A, B(2))	77774	C		
	77772	D		

*Not Allowed — B(3) would not lie within memory.

6. MACHINE LANGUAGE SUBROUTINES

In interfacing machine language subroutines with FORTRAN calling programs the following conventions apply:

1. The contents of the index register X should be saved upon entry to the subroutine and X should be restored prior to return.
2. The contents of the A and B registers are not readily predictable upon entry to the routine and need not be preserved.
3. The value of a FUNCTION (as opposed to a subroutine) is returned to the calling program via the A register (for integer functions) or the A and B registers (for floating functions). The most significant part of the fraction is in A; the least significant part of the fraction and the exponent are in B.

Linkage of Machine Language Subroutines to FORTRAN

The linkage of machine language (M-L) subroutines to FORTRAN programs is simple with the external definition and reference capabilities of SYMBOL and META-SYMBOL. If the FORTRAN program has the instruction

```
    CALL  SUBRNM (A,B,C,...)
```

then the M-L subroutine should have, as a first location symbol, the name of the CALLed subroutine, preceded by a $.

Example 1:

```
    $SUBRNM     PZE
                 .
                 .
                BRR   SUBRNM
```

The FORTRAN loader tags SUBRNM as an unsatisfied subroutine reference, and when the M-L program is loaded, since $SUBRNM is externalized, linkage is set up. Multiple entry points to the M-L subroutine could be established by externalizing the label at each of these points.

Example 2:

```
    $SUBRMN     PZE
                 .
                 .
    $ENTRY2     PZE
                 .
                 .
    $ENTRY3     PZE
                 .
                 .
```

Thus, a statement such as: CALL ENTRY3 (X,Y,Z,...) is possible in the FORTRAN program.

FORTRAN Run-Time POPs

Each M-L subroutine must be preceded by the identification-by-OPD of any run-time POPs used in the subroutine. For example, if XSD, FSD, LDP, STD are to be used in the M-L subroutine SUBR, the following must be done:

```
    XSD         OPD       010000000
    FSD         OPD       010100000
    LDP         OPD       012500000
    STD         OPD       010700000

    $SUBR       PZE
                 .
                 .
                XSD
                 .
                 .
                FSD
                 .
                 .
                etc.
                 .
                 .
                END
```

No POPs from the standard SDS POP library may be called by the M-L subroutine.

Accessing FORTRAN Program Arguments

The M-L subroutine cannot access the FORTRAN variables by name. For example, LDP VSTAR would not pick up the variable VSTAR from the main program. It is necessary to employ the FORTRAN run-time POPs XSD (Fixed Set-up Dummy) and FSD (Floating Set-up Dummy) to make the address of the FORTRAN variable available to the M-L program.

Each time XSD or FSD is used in the M-L program, the address of a variable in the calling sequence is placed in the effective address of the POP reference line.

Example:

FORTRAN Program

```
CALL MLSUBR (A,I,R)
```

M-L Program

```
 1.  XSD        OPD   010000000
 2.  FSD        OPD   010100000
                 .
                 .
                 .
 3.  $MLSUBR    PZE
                 .
                 .
                 .
 4.             BRM   201SYS
 5.             FSD   VAR1
 6.             XSD   VAR2
 7.             FSD   VAR3
 8.             BRM   202SYS
                 .
                 .
                 .
 9.             LDA   *VAR2
                 .
                 .
                 .
10.             LDP   *VAR1
                 .
                 .
                 .
11.             LDX   =5
12.             LDP   *VAR3+1
                 .
                 .
                 .
13.  VAR1       RES   2
14.  VAR2       RES   2
15.  VAR3       RES   2
                 .
                 .
                 .
                END
```

Note that a pair of locations is reserved for each variable address (lines 13-15), whether the variable is fixed or floating. The first location of the pair (e.g., VAR1) contains the address of the variable; the second (e.g., VAR1+1) contains the address of the variable with the index tag set.

When line 5 has been executed, VAR1 (line 13) will contain the address of the first variable in the calling sequence (i.e., A); VAR1+1 will contain the address of A with the index tag set. Line 6 will set the address and address tagged of I into VAR2 and VAR2+1; line 7 will set the address of R into VAR3 and VAR3+1.

Line 9 will load the A register with variable I. Line 10 will load the A and B registers with variable A. Suppose R is an array variable (i.e., DIMENSION R (100)), then lines 11 and 12 will result in the loading of A and B registers with R(6). If we LDX with an N, then the sequence

```
LDX    =N
LDP    *VAR3+1
```

will load R(N+L), where L is defined in a DIMENSION statement: DIMENSION R(L/U), into A, B, since VAR3+1 contains the address of the beginning location of the array R, with index tag set. Note that LDP takes care of the doubling of the index register necessary for floating-point variables that occupy two cells each.

Writing POPs for SYMBOL/META-SYMBOL Subroutines Called from FORTRAN Programs

The programmer may supply his own POP to be used in the SYMBOL/META-SYMBOL subroutine. However, the following conventions must be followed to ensure that linkage is established at load time.

Let LXR be a POP defined in a subroutine written by the programmer, with POP transfer location 175. The POP LXR is to be used in a SYMBOL subprogram called $MLSUB. The SYMBOL subprogram will have the following form:

Example 1:

```
LXR       OPD   017500000    Defines Opcode
$MLSUB    PZE                Start of SYMBOL subroutine
          .
          .
          .
          LXR   ALPHA        POP reference
          .
          .
          .
          PZE   175SYS       Establishes external reference to be satisfied at load time
          .
          .
          .
          END
```

Note that although mnemonic LXR forces Opcode 175 to be used (because of OPD), and transfer to that location will occur at execution time, the FORTRAN loader cannot interpret the reference LXR. For this reason, the POP subroutine LXR, instead of being designated $LXR POPD, must be labeled $175SYS in order for the FORTRAN loader to load it and set up the linkage through location 175. This explains the need for PZE 175SYS in the SYMBOL subprogram to generate the required external reference.

Example 2:

```
*DEFINITION OF LXR POP USED IN MLSUB
$175SYS   STX
          .
          .
          .
          BRR   0
          END
```

The FORTRAN loader will place a BRU to $175SYS in location 175. LXR will be executed as a POP in MLSUB.

The programmer is restricted to use of POP locations 162-177. Therefore, the only labels that may be used in programmed operator definitions are:

```
$162SYS
$163SYS
.
.
.
$177SYS
```

It is suggested that the programmer assigns from 177 down, to leave the 162 upward available for possible expansion of FORTRAN run-time POPs.

When a FORTRAN program calls a subroutine, the addresses of the arguments are placed into the erasable storage area (all memory not used by programs or data). Since the subroutine must obtain its arguments by using these addresses, the following run-time locations provide the necessary information.

Name	Octal Location	Contents
EADR1	15	Location of E(N), which follows the last argument.
EADR2	16	Pointer to location in erasable where XSD(FSD) gets next argument address.
E0ADR	71	E0, the first location of the argument address vector.
E0SIZE	72	Total size of erasable storage.
E0TAG	73	Same as E0ADR plus tag bit.
E0IND	74	Same as E0ADR plus indirect bit.
FLTIND	254	Floating indicator, octal 01000000.
E0	L	Location of first argument – Bit 5 is 1 if floating.
E1	L+1	Location of second argument – Bit 5 is 1 if floating.
⋮	⋮	⋮
E(N-1)	L+N-1	Location of Nth argument – Bit 5 is 1 if floating.

Argument mode may be determined as follows:

```
LDA*    E0ADR    }  Skip if first argument is integer; otherwise
SKA     FLTIND   }  argument is floating-point.

LDA     FLTIND   }
LDX     (N-1)    }  Skip if Nth argument is integer; otherwise
SKA*    E0TAG    }  argument is floating-point.
```

The following methods may be used to access partial or double-word arguments. (Less-significant half, more-significant half, or the whole double-precision word. In the case of integer arguments, the address of the less-significant half should be used.)

	Method 1	Method 2	Method 3
Arg. 1, lsh or integer	LDX E0ADR LDA* 0, 2	LDX* E0ADR LDA 0, 2	LDA* E0IND
Arg. 1, msh		LDX* E0ADR LDA 1, 2	
Arg. 1, both			LDP* E0IND
Arg. 2, lsh or integer	LDX E0ADR LDA* 1, 2		LDA E0IND ADD ONE STA TEMP LDA* TEMP
Arg. 2, msh	LDX E0ADR LDX 1, 2 LDA 1, 2		
Arg. 2, both			LDA E0IND ADD ONE STA TEMP LDP* TEMP

The operations FIX and FLOAT may be done in SYMBOL-written subroutines without calling the library functions, as follows:

```
        BRM*  FIXL                    BRM*  FLOATL
FIXL    EQU   0266          FLOATL    EQU   0267
```

with the argument and result transferred in the A, B registers.

APPENDIX A
ADDITIONAL OPERATING PROCEDURES

STANDARD FILL PROCEDURE

This procedure is used for loading a paper tape that will not be read in by another program already in memory.

1. Move compute switch to IDLE.
2. Set register switch to C.
3. Push START.
4. Move compute switch to RUN.
5. Raise FILL switch.

STANDARD RESTART PROCEDURE

All programs (compiler, loader, and object program at run-time) may be restarted, as follows:

1. Move compute switch to IDLE.
2. Set register switch to C.
3. Push START.
4. Move compute switch to STEP and RUN.

CLEARING A HALT

This term refers to the operation of moving the compute switch from RUN to IDLE and back to RUN.

APPENDIX B
COMPATIBILITY

SDS FORTRAN II is, for the most part, completely downward compatible with standard FORTRAN II compilers. That is, they are mostly a direct subset of SDS FORTRAN II and programs written for them are fully compatible; programs that use the expanded features of SDS FORTRAN II, however, are not compatible with other compilers.

SDS FORTRAN II does not, in general, compile programs written in:

FORTRAN IV

1620 FORTRAN I (including the Western Region FORTRAN, and all others except FORTRAN with FORMAT)

COMPACT (unless its few differences from FORTRAN II are not used)

ACT, ALGOL, GE WIZ, JOVIAL, MAD, NELIAC, etc.

Note that operations that are not defined by the FORTRAN language and that rely on the hardware configuration of a particular computer cannot be expected to produce the same result when compiled on another computer; for example, reading in alphanumeric characters and using their internal numeric representation as a number, or doing modulo arithmetic dependent on the size of the number at which the machine overflows. Such "machine language" coding via FORTRAN cannot be retained from any one machine to any other.

Since many existing FORTRAN II compilers are, for the most part, mutually incompatible, it is impossible to be compatible with all of them. Accordingly, SDS FORTRAN II was designed to allow maximum compatibility with the most widely used compilers. For example, either SIN or SINF will call the library sine function; alphanumeric FORMAT specifications may, but need not, have commas after them.

The remaining differences are few and, in most cases, rarely found. These differences are detailed for the IBM 1620 and 7090.

1620 FORTRAN II

LOGF must be changed to ALOG, ALOGF, ELOG, or ELOGF.

IF statements with floating-point numbers should not, in general, have a unique branch for zero. That is, "IF (A-B) 4, 5, 6" may not branch to 5 even though A and B are equal to 11 decimal digits. The problem, which is characteristic of all binary machines, is that most decimal fractions do not possess terminating binary representations. Moreover, uncertainties in the low-order bits are compounded by round-off and truncation errors. SDS FORTRAN II provides a library function, called IF, which, as a function of two arguments, checks for equality to within four bits. When given only one argument, it checks whether the argument is less than 10^{-10} in magnitude. Thus, the following changes may be made:

```
IF (A-B) 4,5,6        becomes       IF (IF (A,B)) 4,5,6

IF (A) 4,5,6          becomes       IF (IF (A)) 4,5,6
```

An alternate method that does not make use of the IF library function is as follows:

```
IF (A-B) 4,5,6        becomes          IF (ABSF (A-B) -1.0 E-10) 5,5,10
                                    10 IF (A-B) 4,5,6
```

Variable Precision — Integers that exceed 8, 388, 607 are truncated at the high-order end. Floating-point numbers must be in the range 10^{-77} to 10^{77}. Accuracy greater than 12 decimal digits is not available.

FORMATs specifying more than 80 characters per typewritten line compile correctly but the extra 1-7 characters are lost when the line is typed.

709/7090 FORTRAN II

The following differences are mostly anomalies within 709/7090 FORTRAN II that have been recognized and removed in SDS FORTRAN II.

Function Names

The function names below on the left must be changed to those on the right to obtain the proper mode of the result:

LOGF	ALOG, ALOGF, ELOG, or ELOGF
XABSF	IABS or IABSF
XFIXF	IFIX or IFIXF
XSIGNF	ISIGN or ISIGNF
XDIMF	IDIM or IDIMF
XINTF	INT
INTF	AINT or AINTF
XMODF	MOD
MODF	AMOD or AMODF
MAX0F, MAX1F	AMAX, AMAX0, or AMAX1
XMAX0F, XMAX1F	MAX, MAX0, or MAX1
MIN0F, MIN1F	AMIN, AMIN0, or AMIN1
XMIN0F, XMIN1F	MIN, MIN0, or MIN1

Overlap of Variables in COMMON and EQUIVALENCE

Since arrays are stored backward in 7090 FORTRAN II, problems arise if variables that are equivalenced are not of the same dimension or if an attempt is made to access one variable by calling another with a subscript outside its range.

Note also that in 7090 FORTRAN II, EQUIVALENCE takes precedence over COMMON, which is not true in SDS FORTRAN II.

Handling of Relative Constants

Relative constants are treated as any other variable. Thus, for example, the index of a DO loop, if used as an index in an I/O statement, will be changed.

B, I, F, or D in Column 1

Boolean and complex statements and function names transferred to subroutines are not included in SDS FORTRAN II. Double-precision computation is performed automatically.

Sense Switch 5 or 6

The SDS 900 Series Computers have four breakpoint switches.

Unacceptable Statements

IF DIVIDE CHECK
IF ACCUMULATOR OVERFLOW
IF QUOTIENT OVERFLOW
	replaced by:
	IF FLOATING
	OVERFLOW

FREQUENCY

Evaluation of Extended Integer Expressions

The expression:

```
I*J*K/L*M
```

for example, is evaluated incorrectly in 7090 FORTRAN II as:

```
(((M*I)/L)*K)*J
```

which will, in general, give the wrong result since integer quantities are truncated after each operation. SDS FORTRAN II evaluates this from left to right as:

```
(((I*J)*K)/L)*M
```

Hollerith Constants

e.g., J = 4HABCD is not allowed in the basic compiler.

Dimensioned Variables without Subscripts

e.g., X to mean X(1) or X(1, 1) is not allowed.

More than 3 Continuation Cards are not allowed in the basic compiler.

FORMATs Read in at Object Time are not allowed in the basic compiler.

"A" FORMAT with Integer Variables

The 7090, having a 36-bit word, allows a width of 6 characters. SDS FORTRAN II allows 4 with integers and 8 with floating-point variables. Extra characters are lost at the left.

SDS

Scientific Data Systems A XEROX COMPANY

701 South Aviation Blvd./El Segundo, California 90245 (213) 772-4511 Cable: SCIDATA / Telex: 674839 / TWX: 910-325-6908

EASTERN TECHNOLOGY CENTER
12150 Parklawn Drive
Rockville, Maryland 20852
(301) 933-5900

PRINTED CIRCUITS DEPT.
600 East Bonita Avenue
Pomona, Calif. 91767
(714) 624-8011

TECHNICAL TRAINING
5250 West Century Blvd.
Los Angeles, Calif. 90045
(213) 772-4511

SALES OFFICES

Western Region

5045 N. 12th St.
Phoenix, Arizona 85007
(602) 264-9324

1360 So. Anaheim Blvd.
Anaheim, Calif. 92805
(714) 774-0461

*5250 West Century Blvd.
Los Angeles, Calif. 90045
(213) 772-4511

Vista Del Lago Office Center
122 Saratoga Avenue
Santa Clara, Calif. 95050
(408) 246-8330

3333 South Bannock
Suite 400
Englewood, Colo. 80110
(303) 761-2645

Fountain Professional Bldg.
9004 Menaul Blvd., N.E.
Albuquerque, N.M. 87112
(505) 298-7683

El Paso Natural Gas Bldg.
Suite 201
315 E. 2nd South Street
Salt Lake City, Utah 84111
(801) 322-0501

Dravo Bldg., Suite 501
225 108th Street, N.E.
Bellevue, Wash. 98004
(206) 454-3991

Midwestern Region

*2720 Des Plaines Avenue
Des Plaines, Illinois 60018
(312) 824-8147

17500 W. Eight Mile Road
Southfield, Michigan 48076
(313) 353-7360

4367 Woodson Road
St. Louis, Missouri 63134
(314) 423-6200

Seven Parkway Center
Suite 238
Pittsburgh, Pa. 15220
(412) 921-3640

Southern Region

State National Bank Bldg.
Suite 620
200 W. Court Square
Huntsville, Alabama 35801
(205) 539-5131

Orlando Executive Center
1080 Woodcock Road
Orlando, Florida 32803
(305) 841-6371

2964 Peachtree Road, N.W.
Suite 350
Atlanta, Georgia 30305
(404) 261-5323

First National Bank Bldg.
Suite 311-B
7809 Airline Highway
Metairie, Louisiana 70003
(504) 721-9172

8383 Stemmons Freeway
Suite 233
Dallas, Texas 75247
(214) 637-4340

*3411 Richmond Avenue
Suite 202
Houston, Texas 77027
(713) 621-0220

Eastern Region

20 Walnut Street
Wellesley, Mass. 02181
(617) 237-2300

Brearley Office Building
190 Moore Street
Hackensack, N. J. 07601
(201) 489-0100

*1301 Avenue of the Americas
New York City, N.Y. 10019
(212) 765-1230

673 Panorama Trail West
Rochester, New York 14625
(716) 586-1500

P.O. Box 168
1260 Virginia Drive
Ft. Washington Industrial Park
Ft. Washington, Pa. 19034
(215) 643-2130

Washington (D.C.) Operations

*2351 Research Blvd.
Rockville, Maryland 20850
(301) 948-8190

Canada

864 Lady Ellen Place
Ottawa 3, Ontario
(613) 722-8387

Oil Exchange Building
309 7th Avenue, S.W.
Calgary 2, Alberta
(403) 265-8134

280 Belfield Road
Rexdale, Ontario
(416) 677-8422

INTERNATIONAL MANUFACTURING SUBSIDIARY

Scientific Data Systems Israel, Ltd.
P.O. Box 5101
Haifa, Israel
04-530253
04-64589
Telex: 922 4474

INTERNATIONAL OFFICES & REPRESENTATIVES

European Headquarters

Scientific Data Systems
I.L.I. House
Olympic Way
Wembley Park (London)
Middlesex, England
(01) 903-2511
Telex: 27992

France

Compagnie Internationale pour l'Informatique, C.I.I.

EXECUTIVE AND SALES OFFICES

66, Route de Versailles
78-Louveciennes
Yvelines
951 86 00 (Paris area)

MANUFACTURING AND ENGINEERING

Rue Jean Jaures
78-Les Clayes-sous-Bois
Yvelines
950 94 00 (Paris area)

MANUFACTURING

Rocade Sud du Mirail
31-Toulouse-03

Israel

Elbit Computers Ltd.
Subsidiary of Elron Electronic Industries Ltd.
88 Hagiborim Street
Haifa
6 4613

*Regional Headquarters

90 05 87B