HONEYWELL

LEVEL 66 FORTRAN REFERENCE MANUAL

SOFTWARE

SERIES 60 (LEVEL 66)/6000

FORTRAN REFERENCE MANUAL ADDENDUM A

SUBJECT

Additions and Changes to Series 60 (Level 66)/6000 FORTRAN Reference Manual

SPECIAL INSTRUCTIONS

This update, Order Number DG75A, is the first addendum to DG75, Rev. 0, dated July, 1978. The attached pages are to be inserted into the manual as indicated in the collating instructions on the back of this cover. Change bars in the page margins indicate technical additions and changes; asterisks indicate deleted material. These changes will be incorporated into the next revision of the manual.

Note:

This cover should be placed following the manual cover to indicate that the document has been updated with Addendum A.

SOFTWARE SUPPORTED

Series 60 (Level 66)/6000 Software Release FT2.0

ORDER NUMBER

DG75A, Rev. 0

July 1979

24203
1.5779
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
v through viii	v through viii
ix,blank	ix,blank
2-1 through 2-4	2-1 through 2-4
2-7 through 2-18	2-7 through 2-18
2-21 through 2-26	2-21 through 2-26
2-29 through 2-32	2-29 through 2-32
3-7, 3-8	3-7, 3-8
3-11, 3-12	3-11, 3-12
3-19 through 3-22	3-19 through 3-22
3-27, 3-28	3-27, 3-28
3-33 through 3-36	3-33 through 3-36
3-43, 3-44	3-43, 3-44
3-53, 3-54	3-53, 3-54
4-3, 4-4	4-3, 4-4
	4-4.1, blank
	4-5, 4-6
4-9, 4-10	4-9, 4-10
6-15 through 6-18	6-15 through 6-18
6-21, 6-22	6-21, 6-22
6-35, 6-36	6-35, 6-36
6-45 through 6-48	6-45 through 6-48
6-63, 6-64	6-63, 6-64
6-69, 6-70	6-69, 6-70
6-73 through 6-78	6-73 through 6-78
B-1 through B-30	B-1 through B-6
B-31, blank	B-7, blank
	B-7.1, B-8
	B-9 through B-36
C-9, blank	C-9, blank
i-1 through i-15	i-1 through i-17

File No: 1P23,1723

SERIES 60 (LEVEL 66)/6000

FORTRAN REFERENCE MANUAL

SUBJECT

Description of the *FORTRAN* Programming Language

SOFTWARE SUPPORTED

Series 60 (Level 66)/6000 Software Release FT1.0

ORDER NUMBER

DG75, Rev. 0

July 1978

Honeywell

PREFACE

This FORTRAN reference manual assumes that the reader is familiar with FORTRAN programming principles and basic concepts. All necessary FORTRAN rules and statements are included in this manual.

FUNCTIONAL LISTING OF PUBLICATIONS for SERIES 60 (LEVEL 66) and SERIES 6000 SYSTEMS

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	ORDER NO.
	Series 60 (Level 66)/Series 6000:	
Hardware reference:		
Series 60 Level 66 System	Series 60 Level 66 Summary Description	DC64
Series 6000 System	Series 6000 Summary Description	DA48
DATANET 355 Processor	DATANET 355 Systems Manual	BS03
DATANET 6600 Processor	DATANET 6600 Systems Manual	DC88
Operating system:		
Basic Operating System	General Comprehensive Operating Supervisor (GCOS)	DD19
Job Control Language	Control Cards Reference Manual	DD31
Table Definitions	System Tables	DD14
Table Definitions	NPS Tables	DE34
I/O Via MME GEINOS	I/O Programming	DB82
Shared Systems Operation	System Operation with Shared Mass Storage	DD97
System initialization:		
System Startup	System Startup	DD33
System Operation	System Operation Techniques	DD50
Communications System	GRTS/355 and GRTS/6600 Startup Procedures	DD05
Communications System	NPS Startup	DD51
Communications System	NPS Configuration Examples	DE76
DSS180 Subsystem Startup	DSS180 Startup (Series 6000 only)	DD34
Program Recovery	Program Recovery/Restart	DC98
Data management:		
File System	File Management Supervisor	DD45
Integrated Data Store (I-D-S)	I-D-S/I Programmer's Guide	DC52
Integrated Data Store (I-D-S)	I-D-S/I User's Guide	DC53
File Processing	Indexed Sequential Processor	DD38
File Input/Output	File and Record Control	DD07
File Input/Output	Unified File Access System (UFAS)	DC89
I-D-S Data Query System	I-D-S Data Query System Installation	DD47
I-D-S Data Query System	I-D-S Data Query System User's Guide	DD46
Coexistent I-D-S	Coexistent I-D-S Overview	DE60
Program maintenance:		
Object Program	Source and Object Library Editor	DD06
System Editing	System Library Editor	DD30
Test system:		
Online Test Program	Total Online Test System (TOLTS)	DD39
Test Descriptions	Total Online Test System (TOLTS) Test Pages	DD49
Error Analysis and Logging	Honeywell Error Analysis and Logging System (HEALS)	DD44
Language processors:		
Macro Assembly Language	Macro Assembler Program (GMAP)	DD08
COBOL-68 Language	COBOL Reference Manual	DD25
COBOL-68 Usage	COBOL User's Guide	DD26
Standard COBOL-68 Language	Standard COBOL-68 Reference Manual	DE17
Standard COBOL-68 Usage	Standard COBOL-68 User's Guide	DE18
JOVIAL Language	JOVIAL	DD23
FORTRAN Language	FORTRAN	DD02
Macro Assembly Language	DATANET 355/6600 Macro Assembler Program	DD01

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	ORDER NO.
	Series 60 (Level 66)/Series 6000:	
Generators:		
Sorting	Sort/Merge Program	DD09
Merging	Sort/Merge Program	DD09
Simulators:		
DATANET 355/6600 Simulation	DATANET 355/6600 Simulator	DD32
Service and utility routines:		
Loader	General Loader	DD10
Utility Programs	Utility	DD12
Utility Programs	UTL2 Utility Routine	DC91
Media Conversion	Bulk Media Conversion	DD11
System Accounting	Summary Edit Program	DD24
FORTRAN	FORTRAN Subroutine Libraries	DD20
FNP Loader	DATANET 355/6600 Relocatable Loader	DD35
Service Routines	Service Routines	DD42
Software Debugging	Debug and Trace Routines	DD43
Time Sharing systems:		
Operating System	TSS General Information	DD22
System Programming	TSS Terminal/Batch Interface	DD21
System Programming	TSS System Programmer's Reference Manual	DD17
BASIC Language	Time Sharing BASIC	DD16
FORTRAN Language	FORTRAN	DD02
Text Editing	Time Sharing Text Editor	DD18
dataBASIC Language	dataBASIC System Language Manual	DD95
dataBASIC Loading	dataBASIC Load/Unload System	DD96
Remote communications:		
DATANET 30/305/355/6600 FNP	Remote Terminal Supervisor (GRTS)	DD40
DATANET 355/6600 FNP	Network Processing Supervisor (NPS)	DD48
DATANET 355/6600 FNP	NPS Micro-Ops Programming	DE35
DATANET 355/6600 FNP	Application Guidelines for NPS	DE77
DATANET 700 RNP	RNP/FNP Interface	DB92
Communication facilities:		
COBOL-74 Communications	Message Control System Site Manual	DC99
Transaction processing:		
User's Procedures	Transaction Processing System User's Guide	DD41
Handbooks:		
System-operator communication	System Console Messages	DD13
Error Messages, Abort Codes	Error Messages and Abort Codes	DC97
NPS Error Messages	NPS Error Messages	DE75
Pocket guides:		
Control Card Formats	Control Cards and Abort Codes	DD04
FORTRAN	FORTRAN Pocket Guide	DD82

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

SECTION I

INTRODUCTION

GENERAL

The name FORTRAN was derived from the original reference to a computer language as a "FORmula TRANslator", and was designed to permit the statement of a problem in terms that closely resemble mathematical notation. Although FORTRAN is a source language which is primarily used for solving scientific and engineering problems, it is also highly suitable for business applications. The FORTRAN statements consist of letters and symbols that provide the programmer with easy manipulation of large sets of equations and variables.

The FORTRAN language is augmented by Subroutine Libraries that contain standard arithmetical functions and provide all input/output for the program (refer to the FORTRAN Subroutine Libraries manual for a description of these routines). The user has the ability to write any special purpose subroutines that may be required for a specific application.

CAPABILITIES

The FORTRAN compiler services both batch and time sharing, using the same compiler modules for both environments. Programs can be developed for eventual use in the batch environment with the convenience of the interactive time sharing environment, and after debug is complete, submitting them to batch without concern for time sharing/batch language incompatibilities.

FORTRAN programs can be entered in exactly the same form regardless of the input medium or location. The only difference in the input stream during user interface is the mandatory presence of GCOS control cards for local and remote batch and the required use of command language in the time sharing environment. Remote accessed use of GCOS, including both time sharing and remote batch, contribute significantly to the job load at the Central Computer Site.

SECTION II

FORTRAN SOURCE PROGRAM CHARACTERISTICS

The FORTRAN compiler, like other higher-level language compilers such as COBOL and PL/I, is a processor that translates a FORTRAN program into machine language (GMAP). The statements and symbols that constitute a FORTRAN program must conform to certain rules and definitions before the source language can be translated to machine language for execution with the General Comprehensive Operating Supervisor (GCOS).

The following conventions are used throughout this manual when illustrating the syntax form of FORTRAN symbols, statements, and keywords.

1. Uppercase words must be entered as specified.

2. Lowercase words indicate user-specified information.

3. Items enclosed within brackets are optional.

4. Items enclosed within braces represent choices or alternatives.

5. An ellipsis (...) indicates that the preceding format may be repeated.

CHARACTER SET

FORTRAN utilizes two character sets - ASCII and BCD. The character set and byte size for the generated object code is controlled by an option on the $ FORTY and $ FORTRAN control cards; the source program requires no options. The byte size is 6 bits for BCD and 9 bits for ASCII (refer to Appendix A for the octal and the punch card representations for each character).

The FORTRAN character set is a subset of the full 128 ASCII character set and can be coded in the following manner:

- FORTRAN statements and the keywords do not differentiate between uppercase and lowercase alphabetic characters.

- No distinction is made between the cases in forming variable, function, common, etc. names.

- Uppercase and lowercase letters are recognized as different only in user character data and literals.

- Any character in the ASCII character set is valid as literal data.

A source program can be written using the characters and digits in Table 2-1. Table 2-2 gives a brief description of the special characters that are used for FORTRAN syntax punctuation.

Table 2-1. FORTRAN Character Set[1]

Upper-Case	Lower-Case	Digits	Special Characters
A	a	0	b̸ (space)
B	b	1	=
C	c	2	+
D	d	3	-
E	e	4	↑ or ^
F	f	5	*
G	g	6	&
H	h	7	/
I	i	8	(
J	j	9	)
K	k		,
L	l		.
M	m		$
N	n		'
O	o		;
P	p		"
Q	q		
R	r		
S	s		
T	t		
U	u		
V	v		
W	w		
X	x		
Y	y		
Z	z		

[1]The correct collating sequence for the ASCII characters listed is shown in Appendix A.

Table 2-2. FORTRAN Syntax Punctuation

Symbol	Function
b̸	The space is only meaningful to the compiler in character constants and can be used freely to enhance the readability of programs.
" '	Quotation Marks and apostrophes are used as character constant delimiters. The apostrophe also precedes the record number in random file input/output statements.
$	The currency symbol identifies statement numbers which are used as arguments in a CALL statement. It also serves as a delimiter of input data for a NAMELIST read.
( )	Parentheses are used to enclose subexpressions, complex constants, equivalence groups, format specifications, argument lists, and subscripts. They are also used to specify the ranges of implied DO loops.
+	The plus sign indicates algebraic addition, printer carriage control, or a unary operator.
-	The minus sign indicates algebraic subtraction, or a unary operator.
,	The comma is used as a separator for data symbols and expressions for parameter lists, equivalence groups, complex constants and format specifications.
/	The slash is used to indicate algebraic division, as a delimiter for data lists, labeled common statements, and as a record terminator in a format statement.
;	The semicolon is used as a delimiter when multiple source statements appear on a single line.
=	The equal sign indicates the assignment operator in arithmetic, character, and logical assignment statements, PARAMETER statements, DO statements, and implied DO statements in I/O and data lists.
* **	The asterisk designates a comment line or an alternate return argument in a subroutine statement. The asterisk is also used as the multiplication operator, and a double asterisk is one of the exponentiation operators. The quantity to the left of the sign is raised to the power indicated on the right.
.	The period is used as a radix point and serves as a delimiter for logical and relational operators as well as logical constants.
↑ ^	The vertical arrow and caret serve as additional exponentiation operators. They are alternates to the double asterisk and can be used interchangeably.
&	The ampersand serves as one of the continuation line indicators.

SOURCE PROGRAM FORMAT

Source Program File Types

Source programs generally originate either as punched cards or as lines entered into a terminal. They can also be the product of, or output from, the execution of a program, or they can be compressed in a compilation activity through the use of the COMDK option. These source programs can be kept in the form of decks, paper tape, magnetic tape files, or permanent mass storage files. To be compiled, decks and paper tape media programs must initially be copied to magnetic tape or mass storage. The mass storage file does not need to be permanent because a normal deck setup produces the compiler input file (S*) on a temporary file. However, the source program file must be recorded in standard system format (see the File and Record Control manual). The FORTRAN compiler will accept magnetic tape or mass storage files in standard system format with any of the following media codes:

0 - BCD print line images, without slew control for the printer (variable length records)
1 - compressed BCD card images (Comdecks)
2 - BCD card images (each record = 80 columns)
3 - formatted BCD printer line images, with trailing printer slew control information
6 - ASCII standard system format preceded by one media code 8 record
7 - ASCII print line images, with trailing printer slew control information
8 - TSS information record

Card images are limited to 80 characters, while line images are limited by the device on which they are prepared. For simplification, wherever "card images" and "line images" can both be used, this document simply uses the term "line".

Source Program File Characteristics

A source program file is composed of statements and comments. A statement is the tool necessary to construct a FORTRAN program, and can be classified as executable or non-executable. The FORTRAN statement can be a maximum of 20 card image lines in length. The first line is referred to as the initial line, and subsequent lines are referred to as continuation lines.

Example

```
1     67                                             72
       REAL  X,Y,Z,TOTAL
       INTEGER  L,M,N
       READ  X,Y,Z
       TOTAL = X*2.0 + Y*3.0 + Z*4.0 +
      &SQRT (X,Y,Z)
       TOTAL = TOTAL +
      1L + M + N +
      2N + 3
         .
         .
         .
       END
```

A comment is composed of a single line of documentary with the letter C in the first column of the line. These lines are not executed and can be placed anywhere within the source program.

Example

```
1     7                                        80
-------------------------------------------------
C  THIS PROGRAM WAS WRITTEN BY C. R. JONES
C  ON JULY 1, 1978
C  IT PRINTS THE DIFFERENCE BETWEEN BIG AND SMALL
C  IF BIG IS GREATER THAN SMALL,
C  OTHERWISE, THE PROGRAM TERMINATES

      READ, BIG, SMALL
      IF (SMALL .GT. BIG) GO TO 100
      DIFF = BIG - SMALL


C     PRINT THE ANSWER

      PRINT, BIG, SMALL, DIFF
100   STOP
      END
```

Every program unit (subprogram, main program, etc.) must terminate with an end line. This line contains an END statement and serves to separate individual program units. Any subsequent units must begin on a new line.

Example

```
      COMMON/LABEL/A,B,C,Y
          .
          .
          .
      STOP
      END
      SUBROUTINE S
      COMMON/LABEL/Q,R,S,T
          .
          .
          .
      RETURN
      END
```

When the first line of a program unit is a comment line, page titles and object deck labels are extracted from that line as follows:

Characters 2-7	are inserted by the compiler into the label field of the heading line printed by the compiler. Only characters 2-5 are used by the compiler to construct the edit name of the compiled module (columns 73-76 of the object deck) which is used by the Source and Object Library Editor to manipulate the module.
Characters 8-72	contain the page title for listings.

When the first line of a program unit is not a comment line, or columns 2 through 5 are blank on the first comment card, the deck label is the first six characters of the program unit's name (...... is used for a main program); no page title is generated. Any trailing digits in the object deck label are used as part of the sequence number field in object decks. To avoid a sequence number error, large source programs should avoid a deck label that ends with a digit.

Format Rules for Lines

A variety of source line formats can be used ranging from the standard 80-character fixed format to the standard line formats used with the time sharing system. Specification of a format is via two options: FORM/NFORM and LNO/NLNO. These options can appear on the $ FORTY or $ FORTRAN control cards, or in the option list of the YFORTRAN or FORTRAN RUN command.

Batch mode source files conforming to the FORTRAN in standard line format defined by ANSI3.9-1966 should be processed using the default option FORM; time sharing source files should normally use the default option NFORM, and LNO.

Line formats have the following characteristics:

1. Initial lines can begin with a statement label.
2. The statement label can begin anywhere on the line but must be in the range $1 \leq n \leq 99999$.
3. There can be a maximum of 19 continuation lines. The statement text continues with the first character following the continuation character.

4. A statement can be terminated by a semicolon on either an initial line or a continuation line. The information following the semicolon is processed as an initial line. The new statement can begin with a statement label and can be continued.

5. The FORM/NFORM options are used to control the following functions:

   a. Elimination of line numbers and sequence identification fields from the lines.

   b. Separation of comment lines from statement lines.

   c. Distinction between initial statement lines and continuation lines.

   d. Determination of the position numbers of the first and last characters of the statement text.

6. Because the FORM formatted files cannot contain line numbers, the LNO option cannot be specified; therefore, the NLNO option is the default.

7. The FORM option has the following characteristics:

   a. Only the first 80 characters on a line are processed - any additional characters are ignored.

   b. Comment lines must have a C or an * in the first character position

   Example

```
1     7
_____________________________
C     COMMENT LINE FORMAT #1
*     COMMENT LINE FORMAT #2
```

   c. Continuation lines must begin with a nonblank, nonzero character in the sixth character position. When an ampersand appears as the first nonblank character anywhere from column 6 on, it will be interpreted as a continuation line.

   Example

```
1234567
_____________________________
      AREA  = (X(2)-X(1))/3*Y(1) + SUM
     &+4.*Y(2*I)
     &-SUM(I)**2
```

   d. Character positions 73-80 of a card image are used for sequence identification and are not considered part of the statement.

   Example

```
1     7                                   73     80
__________________________________________________
      SUBROUTINE SOLVE                    SOLVE001
      COMMON A, B, X1R                    SOLVE002
      DISC = (B**2)-(4.*A)*X1R            SOLVE003
```

8. Lines in NFORM format with no line numbers (NLNO option) have the following characteristics:

   a. Comment lines must have a C or an * in the first character position.

   b. Continuation lines must be designated with an ampersand as the first nonblank character of the line.

   Example

```
1234567________________________________

         B = SQRTF(DISC)
        &X1R = -B/(2.*A)
        &X2R = SQRTF(-DISC)/(2.*A)
```

   c. Character positions 73-80 of a card image can only be used for sequence identification.

9. Lines in NFORM format with line numbers (LNO option) have the following characteristics:

   a. The line number field can begin in character position 1, or can contain leading blanks but must not extend beyond character position 8. The magnitude of this line number is treated as modulo 2^{18} (262,144).

   b. Line numbers less than eight characters must be followed by a nonnumeric character.

   Example

```
12345678_______________________________

10# READ (6,10) IN,OUT
0010;READ(6,10)A,B,C
   10 READ (6,10) RT,SLM
```

   c. Comment lines must begin with a C or an asterisk as the first character following the line number.

   Example

```
12345678_______________________________

10C COMMENT LINE #1
0020* COMMENT LINE #2
      30C  COMMENT LINE #3
```

   d. A continuation line must have an ampersand as the first nonblank character following the line number.

   Example

```
12345678_______________________________

10& SUM = SUM + A*2.*(I/K)
0020& SQRTS = SUM ** 2
      30& X1R = X22/X3R
```

e. Character positions 73-80 can be used for statement text and will be processed.

Example

```
1     7                                                        73     80
10    FORMAT(60HCHARACTER POSITIONS 73 TO 80 WILL BE UTILIZED AS
```

10. ALTER statements in batch mode which are used in conjunction with the $ UPDATE control card, have the following characteristics:

a. All alters apply only to the first source program if the compilation activity contains more than one source program.

b. Alter statements must be in ascending numerical order.

c. Source programs must be in media code 1 or 2 (COMDECK or BCD card image, respectively).

d. The alter file must be media code 2 (BCD card image).

e. The correction card(s) that follow the $ ALTER card(s) must be in the same format as the source program (i.e., FORM or NFORM). However, if NFORM is the format used, the correction cards cannot contain line numbers.

f. When using the NFORM option, the source program sequence number (not the line number) must be used when specifying the LNO option.

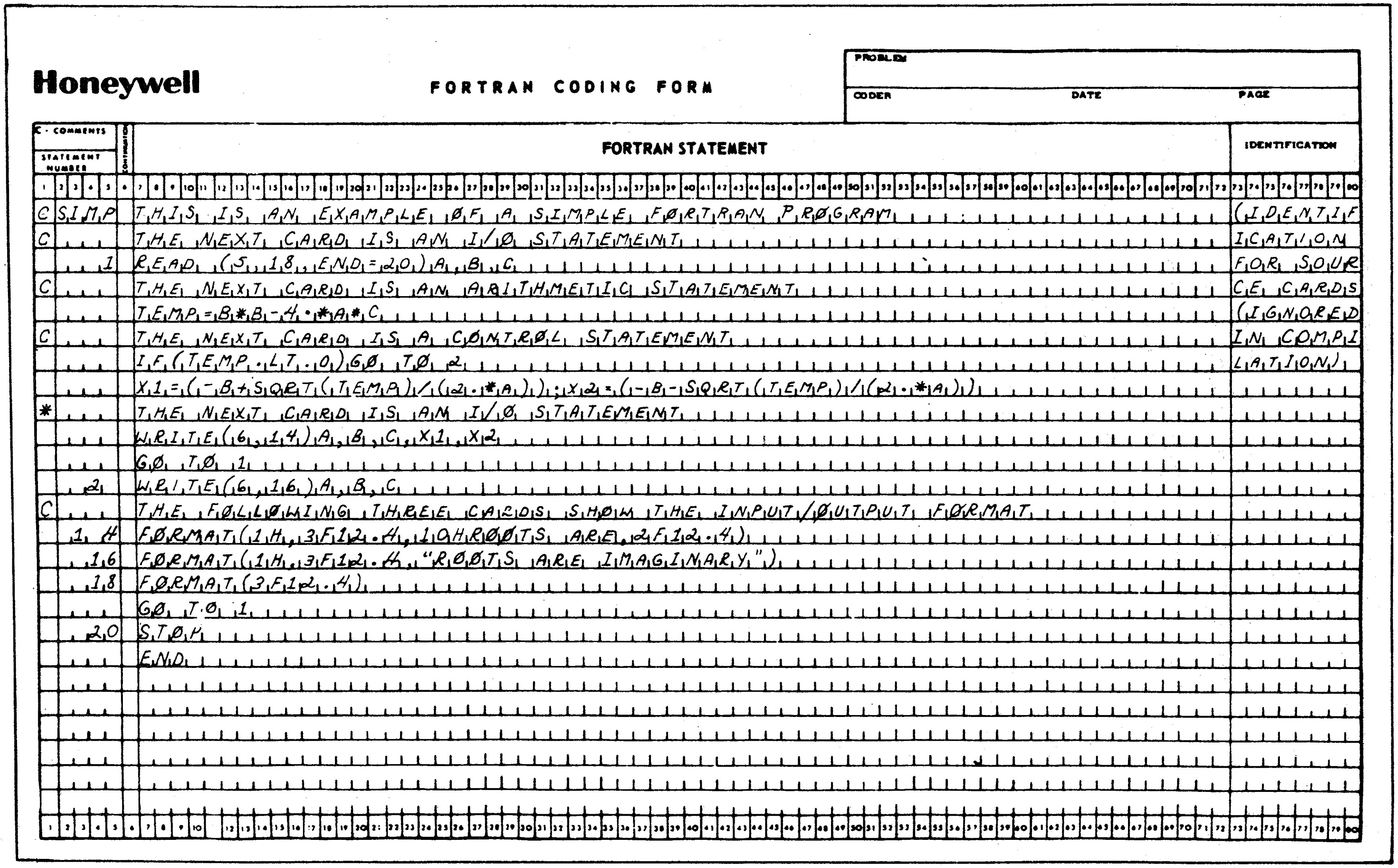
Example

To change line number 25 for

```
10C SAMPLE
20  J = 1
25  PRINT, I
30  STOP; END
```

the $ ALTER card would be coded

```
$ ALTER 3,3
25 PRINT, J
```



Honeywell FORTRAN CODING FORM

PROBLEM
CODER DATE PAGE

C - COMMENTS | STATEMENT NUMBER | CONTINUATION | FORTRAN STATEMENT | IDENTIFICATION

```
CSIMP THIS IS AN EXAMPLE ØF A SIMPLE FØRTRAN PRØGRAM                    (IDENTIF
C     THE NEXT CARD IS AN I/Ø STATEMENT                                 ICATION
    1 READ (5,18,END=20)A,B,C                                           FOR SOUR
C     THE NEXT CARD IS AN ARITHMETIC STATEMENT                          CE CARDS
      TEMP=B*B-4.*A*C                                                   (IGNORED
C     THE NEXT CARD IS A CØNTRØL STATEMENT                              IN COMPI
      IF(TEMP.LT.0)GØ TØ 2                                              LATION)
      X1=(-B+SQRT(TEMA)/(2.*A));X2=(-B-SQRT(TEMP)/(2.*A))
*     THE NEXT CARD IS AN I/Ø STATEMENT
      WRITE(6,14)A,B,C,X1,X2
      GØ TØ 1
    2 WRITE(6,16)A,B,C
C     THE FØLLØWING THREE CARDS SHØW THE INPUT/ØUTPUT FØRMAT
   14 FØRMAT(1H,3F12.4,10HRØØTS ARE,2F12.4)
   16 FØRMAT(1H,3F12.4,"RØØTS ARE IMAGINARY")
   18 FØRMAT(3F12.4)
      GØ TØ 1
   20 STØP
      END
```

Figure 2-1. FORTRAN Coding Sheet and Program

Symbolic Names

A symbolic name is composed of one to eight alphanumeric characters, the first of which must be alphabetic. The data type of the variables that are associated with a symbolic name are defined either implicitly or explicitly. The implicit associations are determined by the first character of the symbol, (i.e., if the name begins with the letters I,J,K,L,M, N, the symbolic name is integer; if it does not, the symbolic name is real. This default implicit associative rule can be changed by the use of the IMPLICIT statement which allows implicit association for all data types - integer, real, double precision, complex, logical, or character. An explicit declaration of type for a symbol always overrides its implicit type.

NOTE: No case distinction is made in forming symbols. The symbol ABC is identical in meaning to the symbols abc and Abc.

DATA TYPES

Data type is explicitly associated with a symbol when it appears in one of the type statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER, or when it appears in a FUNCTION statement with a type prefix (e.g., REAL FUNCTION MPYM(A,B)).

A symbolic name representing a function, variable, or array has only one data type association for each program unit. Once it is associated with a particular data type, a specific name implies that data type for any usage of the specified symbolic name when it requires a data type association throughout the program unit in which it is defined.

The mathematical and representational properties for each of the data types are defined below. The value zero is not considered positive or negative.

1. An integer datum is always an exact representation of an integer value. It can assume positive, negative, or zero integral values. Each integer datum requires one 36-bit word of storage in fixed point format. The permissible range of values for integer type is -(2) to (2)-1.

   Example

   ```
     29        5        - 999999999
   - 25     +444          9999999999
   ```

   NOTE: The largest possible integer (i.e., octal constant +377777777777) is considered a noise word and is printed as an output field filled with blanks.

2. A real datum is a processor approximation to the value of a real number. It can assume positive, negative, zero and sometimes fractional values. A real datum requires one 36-bit word of storage in floating-point format. The permissible range of values for real type data is approximately 10 to 10 with a precision of eight digits.

   Examples

   ```
       29.0      + .00007      7.23456
   - 0004.3        .1         -999999999.
   ```

3. A double precision datum is a processor approximation to the value of a real number. It can assume positive, negative, or zero values. A double precision datum requires two consecutive 36-bit words of storage in double precision floating-point format. The permissible range of values for double precision type is approximately 10 to 10 , with a precision of 18 digits.

   <u>Examples</u>

```
     73.12345               9.2D-2      +.1D4
   - 187.93           -999999999.0D7    7D+6
```

4. A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first datum of the pair represents the real part, and the second datum represents the imaginary part. Each part has, accordingly, the same degree of approximation as a real datum. A complex datum requires two consecutive words of storage, each in floating-point format. Each part of a complex datum has the same range of values and precision as a real datum.

   <u>Examples</u>

```
   ( 1., 5.)              ( .15E+06, 0.6)
   ( - 7.3, 17.4)         ( 0., -.5)
```

5. A logical datum is a representation of a logical value of true or false. The source representation of the logical value true can be either .TRUE. or .T.; in DATA statements, the single character T can also be used. The value false can be represented as .FALSE. or .F., with F being acceptable in the DATA statement(s). A logical datum requires one 36-bit word of storage with the value zero representing false, and nonzero representing true.

   NOTE: When the logical values of true or false are specified in a FORTRAN statement as input data, the single letters T and F (without the periods) must be used for true and false, respectively.

   <u>Examples</u>

```
   .TRUE.      .T.
   .FALSE.     .F.
```

6. A character datum is a processor representation of a string of ASCII or BCD characters. This string can consist of any characters capable of being represented in the processor. The space character is a valid and significant character in a character datum. Character strings are delimited by quotes, apostrophes, or by preceding the string with nH. The character set (BCD or ASCII) is declared by an option on the $ FORTY or $ FORTRAN control card.

   <u>Examples</u>

```
     ABC       DOG       12345
   - XYZ      $CAT$      #INT#
```

The term "reference" indicates an identification of a datum, implying that the current value of the datum will be made available during the execution of the statement which contains the reference. If the datum is identified but is not made available, the datum is said to be "named". One case of special interest in which the datum is named is assigning a value to a datum, which defines or redefines the datum.

CONSTANTS

A constant is a value that does not change during program execution. The three general types of constants are single word, double word, and character. The single and double word constants are divided as follows:

1. Single Word Constants
   a. Integer
   b. Octal
   c. Real
   d. Logical
2. Double Word Constants
   a. Double Precision
   b. Complex

Integer Constant

An integer constant is a numeric designation in fixed point binary format.

Syntax

$\pm$ integer

Syntax Rules

Integer can consist of one to eleven decimal digits.

General Rules

1. The accuracy of an integer constant is ten digits.
2. Integer can be as large as $(2^{35})-1$ (i.e., $\cong$ 3.4×10^{10}).
3. If integer is a subscript, an index, or a DO parameter, the maximum value is $(2^{18})-1$ (i.e., $\cong$ 260,000).

*

Examples

```
-7
843517
```

*

Octal Constants

An octal constant is the designation of a value in octal format.

Syntax

O ±constant

Syntax Rules

1. Constant is a string of one to twelve octal digits (i.e., 0 to 7).
2. Constant must be preceded by the alpha character O, and an optional sign.
3. Constant can be used in preset data lists only (e.g., the DATA statement).

General Rule

The optional sign affects only bit 0 of the resulting literal (i.e., complementation does not take place).

Examples

```
DATA A/O-1/ (results in 400000000001)
DATA J/O 377777777777/
DATA S/O 34567/ (results in 000000034567)
```

Real Constant

A real constant is a numeric representation in floating-point binary format.

Syntax

± integer-1 [.[integer-2]] [E [±] integer-3]

Syntax Rules

1. Integer-1.integer-2 can have a maximum of nine significant decimal digits written with a decimal point.

2. Integer-3 can be a one- or two-digit integer constant.

3. If the E integer-3 option is specified, integer-3 cannot be blank, but the value can be explicitly zero.

4. When the decimal point is omitted, it is assumed to be immediately to the right of the rightmost digit of integer-1.

5. Either the decimal point or the E must be specified.

General Rules

1. A real constant is contained in one computer word (i.e., single precision).

2. A real constant has precision to eight digits.

3. The magnitude of the real constant must be approximately between 10^{-38} and 10^{38}, or it must be zero.

4. In some cases, nine significant decimal digits will generate a double precision constant because the mantissa of the real constant is greater than or equal to 2^{28}, or 268435456.

Examples

```
75.
21.083
-3.28
 7.0E2   (means 7.0 x 10² or 700.)
 7E-3    (means 7.0 x 10⁻³ or .007)
```

Double Precision Constant

A double precision constant is a numeric designation in floating-point format.

Syntax

± integer-1 [.[integer-2]] [D [±] integer-3]

Syntax Rules

1. Integer-1.integer-2 can have a minimum of ten and maximum of eighteen significant decimal digits written with a decimal point.
2. Integer-1 can be up to eighteen significant digits written with or without a decimal point when it is followed by a decimal exponent.
3. Integer-3 can be a one- or two-digit integer constant.
4. When the decimal point is omitted, it is assumed to be immediately to the right of the rightmost digit of integer-1.
5. Integer-3 cannot be blank, but the value can be explicitly zero.
6. Either the decimal point or the D must be specified.

General Rules

1. Double precision constants have precision to eighteen digits.
2. The magnitude of a double precision constant must lie between 10^{-38} and 10^{38}, or it must be zero.
3. In some cases, nine significant decimal digits will generate a double precision constant because the mantissa of the real constant is greater than or equal to 2^{28}, or 268435456.

Examples

```
 12.34567891
-13.57D0
   .1234D0
  7.0D4    (means 7.0 x 10^4 or 70000.)
  7D-3    (means 7.0 x 10^-3 or .007)
```

Complex Constant

A complex constant is composed of an ordered pair of signed or unsigned real constants; the first pair represents the real portion of the constant, and the second pair represents the imaginary portion of a complex constant.

Syntax

```
(±real-1, ±real-2)
```

Syntax Rules

1. Real-1 represents the real part of the complex number; real-2 represents the imaginary part.
2. The parentheses are required, regardless of the context in which the complex constant appears.
3. Real-1 and real-2 must be separated by a comma.

Examples

```
(10.1 , 7.03)   means 10.1 + 7.03i
( 5.41, 0.0 )   means 5.41 + 0.0i
( 7.0E4, 20.76) means 70000 + 20.76i
```

where: *i* is the square root of -1.

Logical Constant

A logical constant is the designation of a value as true or false.

Syntax

$$\left\{\begin{array}{l}\text{.TRUE.}\\ \text{.FALSE.}\end{array}\right\}\text{or}\left\{\begin{array}{l}\text{.T.}\\ \text{.F.}\end{array}\right\}$$

Syntax Rules

1. A logical constant can be represented in a source program in either of the forms noted above.
2. A logical constant can be represented without periods when performing input operations (i.e., when used as input data).

General Rule

The logical constants are represented in the machine as

```
TRUE   ≠ zero
FALSE  = zero
```

Examples

```
L = .T.
A = L .OR. .TRUE.
```

Character Constant

A character constant is either an ASCII or BCD representation of a character string (refer to Appendix A for a description of each character set).

Syntax

```
nHliteral-1
"literal-2"
'literal-3'
```

Syntax Rule

Literal can be a maximum length of 500 characters for ASCII mode and 511 characters for BCD mode.

General Rules

1. The type (ASCII or BCD) for literal is determined by an option on the $ FORTY or $ FORTRAN control card, or the YFORTRAN or FORTRAN RUN command.

2. Literals can be used

   - as arguments to external programs
   - as literals in the DATA statement
   - as part of a FORMAT statement
   - as the display object of the STOP and PAUSE statements
   - in a character assignment statement
   - in a relational expression

3. If two delimiters are placed directly together, it is considered to be a single occurrence of the delimiter (i.e., "abc""ef" is interpreted internally as abc"ef). However, an alternate delimiter type can be used (e.g., 'abc"ef').

Examples

```
'CHAR'
'CONSTANT'
CHARACTER*5  A/"1.0,6"/
CALL SUB('CHAR',J)
X = "Y+Z"
```

VARIABLES

A variable is any quantity referred to by a symbolic name with a value that can be changed during the execution of a program. The type of a variable is specified implicitly by its name, or explicitly by the use of a type statement.

1. Default implicit type association enables the declaration of real and integer variables and function names according to the following rules:

   a. If the first character of the name is I,J,K,L,M, or N, (uppercase or lowercase) it is an integer variable.

      Example

      ```
      INTG
      LIST
      NAME
      ```

   b. If the first character is any other alphabetic character, it is a real variable.

      Example

      ```
      REALA
      ARND
      ZXY
      ```

2. The IMPLICIT type statement redefines the default implied typing.

   Examples

   ```
   IMPLICIT INTEGER (A-H)
   ```

   All program variables beginning with the letters A through the letter H, as well as the default letters I through N, will be type integer.

   ```
   IMPLICIT REAL (I-N)
   ```

   All program variables beginning with the letters I through N, as well as the default letters A through H and O through Z, will be type real.

   ```
   IMPLICIT INTEGER (X,Z), REAL (J)
   ```

   All program variables beginning with the letters X and Z, as well as the default letters I and K through N, will be type integer; variables beginning with J, as well as the default letters A through H and O through Z, will be type real.

3. The explicit type statement assigns a type to a variable or function subprogram.

   Examples

   ```
   REAL  IA,IB,IC
   INTEGER X,Y,Z
   COMPLEX COMP,DDN
   ```

4. Function subprogram names can be typed in the FUNCTION statement by use of the type prefix.

   Examples

   ```
   INTEGER FUNCTION RAND (NUMBER)
   REAL FUNCTION ICUM(FACT)
   ```

Defined Variable

A variable is considered to be defined when it is assigned a value. It can be assigned a value through a non-executable statement (e.g., a DATA statement) or an executable statement (e.g., a READ statement). A variable which is a member of any COMMON block is considered defined, as well as any variable that appears in the argument list of a subroutine CALL statement.

Examples

```
     DATA A/10.1/, B/25.4/, C/5.0/
     READ A,B,C
     CALL SUB(A,B,C)
```

Referenced Variable

A variable in a source program is considered to be referenced if it is required to have a value.

Examples

```
     PRINT 3, A, B
     CALL SUB(A,B)
     SUM = A+B
```

Scalar Variable

The six types of scalar variables are character, integer, real, logical, double precision, and complex. A scalar variable can take on any value its corresponding constant may assume, and occupies the same number of storage locations as a constant of the same type.

Examples

```
     SCA = 99999999.9
     PER = 100.0
     COMP = (1.0, 3.4)
```

External Variable

An external variable is the name of a subprogram that appears as an actual argument in the calling sequence to some subprogram. It must appear in an EXTERNAL statement before its first use in the source program.

Examples

```
     EXTERNAL RAND
        .
        .
        .
     CALL SUB(RAND)
        .
        .
        .
```

where RAND is a subprogram name

Switch Variable

A switch variable is an independent entity derived from a scalar variable and is associated only with an ASSIGN statement. A switch variable has no numeric value and must be type INTEGER; but it can have the same symbolic name as an integer variable.

Examples

```
ASSIGN   6 TO J
ASSIGN 999 TO R
```

Character Variable

Character variables can have an implicit type via the IMPLICIT statement or an explicit type using the CHARACTER statement. Character variables are left-justified and blank-filled. The maximum length specification is 500 characters per variable in the ASCII mode and 511 characters in the BCD mode.

Examples

```
CHARACTER*10 ALPHA,NUM*2(2)/'AB','CD','EF'/
CHARACTER DOG
IMPLICIT CHAR*2(A,B,C)
```

Array Variable

An array is an ordered set of data with one to seven dimensions, which is referenced by a symbolic name. Identification of the entire ordered set is achieved by the use of the array name.

Examples

```
ARR (1,2,1)
LIST (I,J,K,L,M,N)
DAT(I,3)
```

ARRAY ELEMENT

An array element is one item of data in an array. It is identified by immediately following the array name with a subscript whose value points to the particular element of the array. In some instances the array name can be used in unsubscripted notation to reference the first element of the array.

SUBSCRIPTS

A variable can be made to represent any element of an array which contains one to seven dimensions by appending one to seven subscripts to the variable name. Subscript expressions are separated by commas, and the number of subscript expressions must correspond with the declared dimensionality (with the exception of the EQUIVALENCE statement). Following evaluation of all of the subscript expressions, the array element successor function determines the identified element.

A subscript expression can take the form of any legal FORTRAN arithmetic expression. The result of any such expression is truncated (not rounded) to an integer before it is used. The value of a subscript expression must be greater than zero and not greater than the corresponding array dimension. The value of a subscript expression containing real variables is truncated to an integer after evaluation. No check is made to verify that the subscript value is within the bounds specified in the DIMENSION statement. The execution of a program containing an error of this nature can cause various abnormal terminations or may give faulty results with a "normal" termination.

ARRAY ELEMENT SUCCESSOR FUNCTION

The general algorithm to linearize a subscript involving n terms (for an array of n dimensions) is:

$$S = \sum_{i=1}^{n} ((e_i - 1) \cdot \prod_{j=0}^{i-1} d_j) + 1$$

where each e_i is a subscript term and each d_j an array dimension.

The term d_0 is the "zero-th dimension" of the array. It reflects the number of words of memory required for one element. For example: integer, logical, and real quantities require one word per element ($d_0 = 1$); double precision and complex quantities require a word pair ($d_0 = 2$); and character variables that use the size in bytes notation to provide the number of characters per element can have a d_0 value of up to 86 in BCD (because this mode has a maximum of 511 characters) and up to 126 in ASCII (because this mode has a maximum of 500 characters). The formula for reducing the size in characters to the size in words is a function of the BCD/ASCII option. Let n be the number of characters specified, and m be the number of characters per word (6 for BCD, 4 for ASCII). Then d_0 is computed as:

$$d_0 = (n+m-1)/m$$

The following are examples using integer variables and using complex variables:

```
INTEGER   X(3,2,4)  (Array X has 3 rows, 2 columns, and 4 planes)
X (2,2,2) = 1
```

Expanding the algorithm for the three dimensions:

$S = (e_1-1)*d_0 + (e_2-1)*d_0*d_1 + (e_3-1)*d_0*d_1 *d_2 +1$

$S = (2-1)*1 + (2-1)*1*3 + (2-1)*1*3*2 + 1$

$S = 1 + 3 + 6 + 1$

$S = 11$

Looking at the array in storage in sequential order, the elements are:

```
X(1,1,1), X(2,1,1), X(3,1,1), X(1,2,1), X(2,2,1),

X(3,2,1), X(1,1,2), X(2,1,2), X(3,1,2),

X(1,2,2), X(2,2,2), ..., X(3,2,4)
```

X(2,2,2) is the eleventh element of the array, the fifth member of plane two.

```
COMPLEX X (3,2,4)
X(2,2,2) = (1.0, 0.0)
```

$S = (2-1)*2 + (2-1)*2*3 + (2-1)*2*3*2 + 1$

$S = 21$

In this example, the first word of the word pair for this element is the twenty-first word of the array.

ARRAY DECLARATOR

An array declarator specifies an array used in a program unit. The array declarator indicates the symbolic name, the number of dimensions (one to seven) and the size of each dimension. The array declarator form can be in a type statement, dimension statement, or common statement. An array declarator has the form:

v(i) or v*n(i)

where: v is the symbolic array name
n is the size-in-bytes of an element
i is the declarator subscript composed of one to seven elements separated by commas; each element can be an integer constant, a parameter symbol, or an integer variable

The appearance of a declarator subscript in a declarator statement informs the processor that the declarator name is an array name. The number of subscripts indicates the dimensions of the array. The magnitude of the value for the subscript expressions indicates the maximum value that the subscript name can attain in any array element reference.

ADJUSTABLE DIMENSIONS

The name of an array and the constants that are its dimensions can be passed as arguments to a subprogram. In this way a subprogram can perform calculations on arrays with sizes that are not determined until the subprogram is called. The following rules apply to the use of adjustable dimensions:

1. Variables can only be used as dimensions of an array in the array declarator of a FUNCTION or SUBROUTINE subprogram, and must be integer. The array name and all the variables used as dimensions must appear as dummy arguments in at least one FUNCTION, SUBROUTINE, or ENTRY statement.

2. The adjustable dimensions cannot be altered within the subprogram.

3. The true dimensions of an actual array must be specified in a DIMENSION, COMMON, or type statement of the calling program.

4. Variable dimension size can be passed through more than one level of the subprogram. The specific dimensions are passed from the calling program to the subprogram as actual arguments cannot exceed the true dimensions of the array.

5. If the variables are not implicitly typed as integer by their initial letters, an INTEGER type statement must precede the dimension statement in which they are used.

6. When an adjustable array name or any of its adjustable dimensions appears in a dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement, that array name and all its adjustable dimensions must also appear in the same dummy argument list.

Example

```
DIMENSION K(4,5),J(2,3)        SUBROUTINE SETFLG(K,J,I,L,M,N)
     .                             .
     .                             .
     .                         DIMENSION K(I,L),J(M,N)
CALL SETFLG (K,J,4,5,2,3)          .
     .                             .
     .                         DO 20 NO = 1,I
     .                         DO 20 MO = 1,L
                               K(NO,MO) = 0
                            20 CONTINUE
                                   .
                                   .
                                   .
                                   .
```

Parameter

A parameter is a constant that is represented as a symbolic name within a source program. The value of this constant is initialized at the beginning of the program. Parameters are used to define any constant whose value might change between compilations (e.g., the pay period ending date for a payroll system).

EXPRESSIONS

Arithmetic

An arithmetic expression is a constant, a variable, a function, or any combination of these items separated by arithmetic operation symbols, commas, and parentheses, to form a meaningful mathematical notation.

Examples

```
A*B
((A+B)/C)**2.5
-(A+B)
(A*B)/(C-D)
```

The following is a list of arithmetic operation symbols:

+	addition or unary addition
-	subtraction or negation
*	multiplication
/	division
** ↑ ∧	exponentiation

The rules for constructing arithmetic expressions are as follows.

1. Constants, variables, and functions that can be combined by the arithmetic operators to form arithmetic expressions are illustrated in Tables 2-3 and 2-4. The intersection of a row and column gives the type of the result of expressions involving the given operators. Table 2-3 gives the valid combinations with respect to the arithmetic operators +,-,*, and /. Table 2-4 gives the valid combinations with respect to the arithmetic operators **, ↑ , or ∧.

Table 2-3. Results for x_1+x_2, x_1-x_2, x_1*x_2 , or x_1/x_2

x_1 \ x_2	I	R	D	C	T
I	I	R	D	C	T
R	R	R	D	C	N
D	D	D	D	C	N
C	C	C	C	C	N
T	T	N	N	N	T

Legend

C - Complex
D - Double precision
I - Integer
N - Nonvalid
R - Real
T - Typeless

Table 2-4. Results for x_1**x_2

		POWER				
	x_1 \ x_2	I	R	D	C	T
	I	I	R	D	N	N
B	R	R	R	D	N	N
A						
S	D	D	D	D	N	N
E						
	C	C	C	C	C	N
	T	N	N	N	N	N

2. Any expression can be enclosed in parentheses.

3. Expressions can be connected by the arithmetic operation symbols to form other expressions, provided that:

   a. No two operators appear in sequence except **, which is a single operator and denotes exponentiation. For example, X+-Y, or X//Y is not valid.

   b. No operation symbol is assumed to be present. For example, (X)(Y) is not valid.

4. Preceding an expression by a plus or minus sign does not affect the type of the expression.

5. In the hierarchy of operations for arithmetic expressions, parentheses can be used to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows:

   a. Function Reference
   b. **, ↑ , or ∧ Exponentiation
   c. + and - Unary Addition and Subtraction
   d. * and / Multiplication and Division
   e. + and - Addition and Subtraction

   This hierarchy is applied first to the expression within the innermost set of parentheses in the statement; this procedure continues through the outer parentheses until the entire expression has been evaluated. For example, in the expression (X-(Y*(2+Z))), (2+Z) is evaluated; then Y* the result of (2+Z) is evaluated; then X- the result of Y*(2+Z) is evaluated.

6. Expressions involving the exponentiation operators are evaluated from right to left. For example, the expression A**B**C is evaluated as A**(B**C).

7. Expressions involving arithmetic operators on the same level (e.g., + and -, or * and /) are evaluated left to right. Parentheses can be used to reorder this sequence if necessary. For example, A/B*C is evaluated as (A/B)*C.

The FORTRAN expression

```
A*6+Z/Y**(W+(A+B)/X**K)
```

represents the mathematical expression

$$6A+\left[\frac{Z}{Y^{\left(W+\frac{(A+B)}{\left(X^{K}\right)}\right)}}\right]$$

Relational

A relational expression consists of two arithmetic expressions connected by a relational operator. Relational expressions always result in a true or false evaluation and can be used in a logical assignment statement, a logical IF statement, a PARAMETER statement, an output list, or as arguments to functions/subroutines.

The six relational operator symbols are:

Symbol	Definition
.GT. or >	Greater than
.GE.	Greater than or equal to
.LT. or <	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to

NOTE: The preceding and following periods are an integral part of the relational operator symbols.

Example

A.GT.B has the value .TRUE. if the quantity A is greater than the quantity B; otherwise, the value is .FALSE.

Logical

A logical expression is a sequence of constants, logical variables, function references, and relational expressions separated by logical operation symbols, that always results in a true or false evaluation.

The logical operation symbols (where a and b are logical expressions) are described below:

Symbol	Definition
.NOT.a	This has the value .TRUE. only if a is .FALSE.; it has the value .FALSE. only if a is .TRUE.
a.AND.b	This has the value .TRUE. only if a and b are both .TRUE.; it has the value .FALSE. if a or b or both are .FALSE.
a.OR.b	(INCLUSIVE OR) This has the value .TRUE. if either a or b or both are .TRUE.; it has the value .FALSE. only if both a and b are .FALSE.

The logical operators NOT, AND, and OR must always be preceded and followed by a period.

Table 2-5. Truth Table Values

a	b	a .AND. b	a .OR. b
.T.	.T.	.T.	.T.
.T.	.F.	.F.	.T.
.F.	.T.	.F.	.T.
.F.	.F.	.F.	.F.

* Logical expression evaluation stops the evaluation as soon as the true/false state for the complete expression has been determined. Thus, it is a distinct possibility that the entire expression may not be evaluated.

Example

```
IF (RAND (X) .GT. 0 .OR.  L) GO TO 100
```

Assuming that RAND is an external function and L is a logical variable, the * expression is true when either RAND(X) is greater than zero or L is true. Since there is no need to evaluate RAND(X) .GT. 0 when L is true, the statement will be optimized into an equivalent pair of statements:

```
IF (L) GO TO 100

IF (RAND(X) .GT. 0) GO TO 100
```

The significance of this is the fact that function RAND is called only when L is false. If evaluation of RAND(X) can have side effects, this may be of consequence. For those applications impacted by this implementation, the solution would be to make the evaluation of RAND(X) unconditional.

Example

```
T = RAND(X)

IF(T.GT. 0 .OR.  L) GO TO 100
```

Logical and Relational Constructions

The following rules are used for constructing logical and relational expressions.

1. The constants, variables, functions, and arithmetic expressions that can be combined by the relational operators to form a relational expression are illustrated in Table 2-6. Y indicates a valid combination and N indicates an invalid combination. The relational expression has the value .TRUE. if the condition expressed by the relational operator is met; otherwise, the relational expression has the value .FALSE.

Table 2-6. Use of Relational Operators (.GT., .GE., etc.)

x_2 / x_1	I	R	D	C	L	H	T	Legend
I	Y	Y	Y	*	N	Y	Y	I = Integer R = Real
R	Y	Y	Y	*	N	N	N	D = Double Precision C = Complex
D	Y	Y	Y	*	N	N	N	L = Logical H = Character
C	*	*	*	*	N	N	N	T = Typeless
L	N	N	N	N	N	N	N	
H	Y	N	N	N	N	Y	N	* = .EQ.,.NE. only Y = Valid
T	Y	N	N	N	N	N	Y	N = Invalid

2. The numeric relationships that determine the true or false evaluation of relational expressions are

   - For numeric values having unlike signs, the positive value is considered larger than a negative value, regardless of the respective magnitude (e.g., $+3 > -5$ and $+5 > -5$).

   - For numeric values having like signs, the magnitude of the values determines the relationship (e.g., $+3 > +2$ and $-8 < -4$).

3. A logical term is a relational expression, a single logical constant, a logical variable, or a reference to a logical function. A logical expression is a series of logical terms or logical expressions connected by the logical operators .AND., .OR., and .NOT.

4. The logical operator .NOT. must be followed by a logical or relational expression, and the logical operators .AND. and .OR. must be preceded and followed by logical or relational expressions.

5. Any logical expression can be enclosed in parentheses.

Typeless Functions

Typeless entities can be combined with an integer or other typeless entities. If a typeless entity is combined by using an arithmetic operator, the result is also typeless and is regarded as a special type of integer. If a typeless entity is combined by using a relational operator, the result is logical; however, a typeless entity cannot be combined using a logical operator.

Whenever the right side of an equal sign yields a typeless result, the assignment operation is integer. For example, if R is real, the statement

```
R = BOOL (R) +1
```

adds one to the least significant bit of the real value of R, using integer-add, and stores a new value in R, using integer-store. This usage is not recommended but is illustrated here to explain the properties of typeless entities.

The typeless functions are listed below:

FLD
AND
OR
XOR
BOOL
COMPL

Evaluation of Expressions

An expression should only be evaluated when it is necessary to determine
* the value of the expression. When two operands are combined by using an operator, the order of evaluation of the operands is undefined as the result of possible reordering during optimization. If the mathematical use of operators is associative, commutative, or both, the orders of combinations can be revised if the parenthesized expressions are not changed. The value of an integer element is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value represented by that element. The associative and commutative laws do not apply in the evaluation of integer terms containing division; hence, the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of a function in an expression cannot alter the value of any other element within the expression, assignment statement, or call statement in which a function reference or subscript appears. No factor can be evaluated that requires a negative valued operand to be raised to a real or double precision exponent, or raising a zero valued primary to a zero valued exponent. An element cannot be evaluated if its value is not mathematically defined.

The evaluation of an arithmetic expression is determined by the following order of type dominance:

1. Complex
2. Double Precision
3. Real
4. Typeless
5. Integer

When two operands are combined by using any of the arithmetic operators other than the exponentiation operator, their respective types are examined according to the stated order of type dominance. The type of the recessive operand is converted to that of the dominant operand (if necessary) and the operation is performed.

<u>Operator Precedence</u>

In the hierarchy of operations, parentheses can be used in logical, relational, and arithmetic expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows:

1.	Function Reference	
2.	**,↑, or ^	Exponentiation
3.	+ and -	Unary Addition and Negation
4.	* and /	Multiplication and Division
5.	+ and -	Addition and Subtraction
6.	.LT., .LE., .EQ., .NE., .GT., .GE.	
7.	.NOT.	
8.	.AND.	
9.	.OR.	

This hierarchy is applied first to the expression within the innermost set of parentheses in the statement and continues through the outermost set of parentheses until the entire expression has been evaluated.

Unary Operators

The unary operators (negative, positive, and logical not) can immediately precede a constant or a variable in an expression. However, if the placement causes the unary negative or positive operator to be adjacent to another operator, it must be enclosed in parentheses with the constant or variable.

Examples

```
A=+1.6
C=D/(-Z)*W
IF(-3.+T4)1,2,3
L1=R2.GT.(-2.)
L2=.NOT.L1
A=B**(-2)
```

SECTION III

FORTRAN STATEMENTS

The basic unit of FORTRAN is the statement which is classified in accordance with the following uses:

1. Arithmetic statements specifying numeric, character, or logical value assignment.
2. Control statements governing the order of execution in the object program.
3. Input/output statements and input/output formats that describe the form of the data.
4. Subprogram statements enabling the programmer to define and use subprograms.
5. Specification statements providing information about variables used in the program, information about storage allocation and data assigned.
6. Compiler control statements direct the compilation activity.

Table 3-1 contains the list of FORTRAN statement types.

Each statement is classified as either executable or non-executable. The executable statement specifies an activity which is to be accomplished; the non-executable statement

- Describes the characteristics, arrangement, and initial values of data
- Contains editing information
- Specifies statement functions
- Classifies program units
- Specifies entry points within subprograms

Table 3-1. Type Listing of FORTRAN Statements

Type	Statement
Arithmetic	Assignment statements Arithmetic statement functions
Control	ASSIGN CONTINUE DO GO TO IF PAUSE STOP
Input/Output	BACKSPACE DECODE ENCODE ENDFILE FORMAT PRINT PUNCH READ REWIND WRITE
Subprogram	BLOCK DATA CALL ENTRY FUNCTION RETURN SUBROUTINE
Specification	ABNORMAL COMMON DATA DIMENSION EQUIVALENCE EXTERNAL IMPLICIT NAMELIST type INTEGER REAL DOUBLE PRECISION COMPLEX LOGICAL CHARACTER PARAMETER
Compiler Control	END

General Format

Most FORTRAN statements, with the exception of the assignment statement, have the following general format:

```
1                  7
___________________________________________________
  statement-label    keyword   syntactic-entities
```

where: statement-label is an unsigned integer constant

keyword is a required FORTRAN word

syntactic-entries are a series of symbols that complete the statement

The assignment statement has two general formats:

Format 1

```
1                  7
_____________________________________________
  statement-label    variable  = expression
```

where: statement-label is an unsigned integer constant

variable is a scalar name or an array element name

expression is an arithmetic, logical, or relational expression

Format 2

```
1                    7
  statement-label-1  ASSIGN  statement-label-2  TO  variable
```

where: statement-label-1 is an unsigned integer constant

statement-label-2 is an executable statement number

variable is an integer switch variable

The statement label is used to reference specific statements within the source program. All FORTRAN statements, with the exception of the END statement, can be labeled. However, only executable and FORMAT statements can be referenced.

The syntax items can be any combination of the following items:

- Constants
- Variable names
- Statement labels
- Operators
- Punctuation symbols

ASSIGNMENT STATEMENTS

An assignment statement is used to give a value to a designated variable. There are four types of assignment statements:

- Arithmetic assignment statement
- Logical assignment statement
- Character assignment statement
- Label assignment (ASSIGN) statement

Arithmetic Assignment Statement

An arithmetic assignment statement instructs FORTRAN to compute the value of an expression on the right of the equal sign and to assign that value to the variable (or array element) on the left of the equal sign.

Format

variable = arithmetic expression

Examples

where: R1 and R2 are real variables

C1 and C2 are complex variables

D is a double precision variable

I is an integer variable

R1 = R2	(R2 replaces the value of R1)
I = R2	(R2 is truncated to an integer and stored in I)
R1 = I	(I is converted to a real variable and stored in R1)
R1 = 3* R2	(3 is converted to a real number, multiplied by R2, and stored in R1)
R1 = R2* D+1	(R2 and D are multiplied using double precision arithmetic; 1 is converted to double precision and added to the product. The most significant digits resulting from the computation are stored in R1 as a real variable)
C1 = C2* (3.7,2.0)	(The result of the complex computation is stored in C1 as a complex number)
C2 = R2	(The real part of C2 is replaced by the value of R2; the imaginary part of C2 is set to zero)

Logical Assignment Statement

A logical assignment statement determines the truth value of a logical expression and assigns it to a logical variable or a logical array element.

Format

logical-variable = logical-expression

Examples

L1 = .TRUE.	(L1 is set to the specified truth value)
L2 = A.GT.25.0	(L2 is set to .TRUE. if A > 25.0; otherwise, L2 is set to .FALSE.)
L3 = I.EQ.0 .OR.A.GT.25.0	(L3 is set to .TRUE. if either I=0, or A > 25.0; otherwise, L3 is set to .FALSE.)

L4 = L5 (L4 is set to the current truth value of L5)

where: L1, L2, L3, L4, and L5 are logical variables

Character Assignment Statement

A character assignment statement stores the characters from a character constant, a variable, a function, or an array element into a declared character variable name.

Format

character-variable = character-expression

General Rules

1. The value of character-expression is stored in character-variable as left-justified with trailing blanks if they are required.
2. If the declared length of character-variable is less than the declared length of character-expression, character-expression is truncated and the leftmost characters are stored in character-variable.

Examples

C1 = "ABCD"	(The characters ABCD are stored in C1)
C2 = C1	(The characters stored in C1 are assigned to C2)
C3 = 'A1B2C3D4'	(The characters A1B2 are stored in C3)

where: C1, C2 and C3 are character variables with a declared length of four characters

Label Assignment Statement

A label assignment statement assigns a statement number to a nonsubscripted switch variable.

Format

ASSIGN statement-no TO switch-variable

General Rule

Statement-no must reference an executable statement number in the same program unit in which the ASSIGN statement appears.

Example

```
ASSIGN 24 to M
    .
    .
    .
GO TO M, (1,22,41,24,36)
```

The next statement to be executed will be statement number 24.

Table 3-2 presents an abbreviated summary of the legitimate combinations of expressions and variables in the assignment statements. When the arithmetic assignment, logical assignment, and character assignment statements are executed, the evaluation of the expression 'e' and the alteration of 'v' is performed in accordance with the rules given in Table 3-3.

Table 3-2. Legal Combinations of Assignment Statements

Variable \ Expression	I	R	D	C	L	H	T	Legend
I	I	I	I	I	N	I	I	I = Integer
R	R	R	R	R	N	R	R	R = Real D = Double Precision
D	D	D	D	D	N	N	N	C = Complex L = Logical
C	C	C	C	C	N	N	N	H = Character T = Typeless
L	N	N	N	N	L	N	L	N = Illegal
H	N	N	N	N	N	H	N	

Table 3-3. Rules for v = e

IF v TYPE IS:	AND e TYPE IS:	THE ASSIGNMENT RULE IS:
Integer	Integer	Assign
Integer	Real	Fix and Assign
Integer	Double Precision	Fix and Assign the Most Significant Part
Integer	Complex	Fix the Real Part and Assign
Integer	Character	Fix and Assign
Integer	Typeless	Assign
Integer	Logical	Illegal
Real	Integer	Float and Assign
Real	Real	Real Assign
Real	Double Precision	Assign the Most Significant Part as Real
Real	Complex	Assign the Real Part
Real	Character	Float and Assign
Real	Typeless	Assign
Real	Logical	Illegal
Double Precision	Integer	Float and Assign as Double Precision
Double Precision	Real	Real Assign as Double Precision
Double Precision	Double Precision	Assign
Double Precision	Complex	Assign Real Part as Double Precision
Double Precision	Character	Illegal
Double Precision	Typeless	Illegal
Double Precision	Logical	Illegal
Complex	Integer	Float and Assign to the Real Part and Assign Zero to the Imaginary Part
Complex	Real	Assign to the Real Part, Assign 0 to Imaginary Part
Complex	Double Precision	Assign the Most Significant Part to the Real Part and Assign 0 to the Imaginary Part
Complex	Complex	Assign
Complex	Character	Illegal
Complex	Typeless	Illegal
Complex	Logical	Illegal
Character	Integer	Illegal
Character	Real	Illegal
Character	Double Precision	Illegal
Character	Complex	Illegal
Character	Character	Assign
Character	Typeless	Illegal
Character	Logical	Illegal
Logical	Integer	Illegal
Logical	Real	Illegal
Logical	Double Precision	Illegal
Logical	Complex	Illegal
Logical	Character	Illegal
Logical	Typeless	Assign
Logical	Logical	Assign

Table 3-3 (cont). Rules for v = e

NOTES:

1. Assign means transmit the resulting value, without change, to the entity.

2. Real assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

3. Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

4. Float means transform the value to the form of a real datum.

5. Double precision float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

6. Assign with respect to character type implies a move operation. When the receiving variable's size is greater than the size of the sending string, the move is performed filling the receiving variable with blanks. When the receiving variable's size is less than that of the sending string, truncation takes place.

FORTRAN KEYWORDS

A description of each FORTRAN keyword, with its associated restrictions, is contained on the following pages in alphabetical order.

ABNORMAL

The ABNORMAL statement is used to qualify the characteristics of a FUNCTION subprogram for optimization purposes.

Format

```
ABNORMAL [function  [,function ] ...]
```

Syntax Rule

Function is a FUNCTION subprogram name.

General Rules

1. The references to the FUNCTION subprogram cannot be treated as a variable or array element reference in an expression. There may be side effects which could alter the function's arguments or locations in common; it performs I/O or it is capable of returning different results when the same arguments are given.

2. Subroutines referenced by CALL statements are always considered abnormal.

3. This statement applies only to programs compiled with the OPTZ option. Otherwise, the presence or absence of ABNORMAL statements is immaterial.

4. If a program unit has FUNCTION references that are not abnormal, an ABNORMAL statement with no argument list may be included. This technique has the effect of setting all functions to 'normal'.

5. If the program unit has no FUNCTIONs typed as ABNORMAL, then all functions are considered abnormal except the supplied functions ir Tables 6-1, 6-2, and 6-3.

Examples

```
ABNORMAL  SINE
ABNORMAL  SUB
ABNORMAL
```

ASSIGN

The ASSIGN statement assigns the value of a statement label to a switch variable. A maximum of 125 assignments of labels can be made to the same switch variable.

Format

ASSIGN label TO switch-variable

Syntax Rules

1. Label must be the statement label of an executable statement.
2. Switch-variable must be an integer switch variable.

Example

```
ASSIGN 17 to J
GO TO J,(5,4,17,2)
```

The next statement to be executed is statement number 17.

BACKSPACE

The BACKSPACE statement positions a file to the record which was the preceding record prior to execution of the backspace command.

Format

BACKSPACE file

Syntax Rules

1. File is the two-character file code which specifies the file to be backspaced.
2. File must be an integer constant, a variable, or an expression.

General Rules

1. If the last READ statement resulted in an end-of-file condition, two BACKSPACE commands are required to position the file prior to the last logical record.
2. If the file is positioned at the initial point, the BACKSPACE statement has no affect.
3. If the device is tape, one BACKSPACE command following a READ that resulted in an end-of-file condition will cause the input file to be set as an output file. Under this condition, the sequence READ, BACKSPACE, READ, will be illegal.

Examples

```
BACKSPACE 05
BACKSPACE 13
```

BLOCK DATA

The BLOCK DATA subprogram is used to enter data into a labeled COMMON block area during compilation of the source program.

Format

```
BLOCK DATA
  .
  .
  .
END
```

Syntax Rules

1. Data cannot be entered into blank COMMON by the BLOCK DATA subprogram.
2. This subprogram can contain only type, EQUIVALENCE, PARAMETER, IMPLICIT, DATA, DIMENSION, and COMMON statements.

General Rules

1. The BLOCK DATA subprogram cannot contain any executable statements.
2. The first statement of this subprogram must be the BLOCK DATA statement. The last statement must be the END statement.
3. All elements of a common block must be listed in the COMMON statement even though they do not all appear in the DATA statement.
4. Data can be entered into a maximum of 63 common blocks in a single BLOCK DATA subprogram.
5. BLOCK DATA subprograms must not be compiled with the DEBUG option.
6. BLOCK DATA subprograms cannot reside on the same random library as a main program referencing its data.
7. If two or more BLOCK DATA subprograms occur for the same application, the data specified by each of them is entered into the appropriate common blocks. The data from the last subprogram is retained for any area of a common block that is referred to more than once.

   NOTE: All the variables in a labeled common block must be listed even though they do not receive a value in the DATA statement.

Example

```
      BLOCK DATA
      DOUBLE PRECISION Z
      COMPLEX C
      COMMON/ELN/C,A,B/RNC/Z,Y
      DIMENSION B(4), Z(3)
      DATA (B(I),I=1,4)/1.1,1.2,2*1.3/,C/(2.4,3.769/,Z(1)/7.6498085D0/
      END
```

CALL

The CALL statement is used to access a subprogram. Upon execution of a CALL statement, control is transferred to the subprogram until the return is made to the calling program.

Format

CALL sub [(arg$_1$, arg$_2$, ..., arg$_n$)]

Syntax Rules

1. Sub is the name of the subroutine subprogram.
2. Arg is the actual argument(s) or the alternate return(s), $ n, where: n is the statement label or switch variable.
3. The arguments must agree in number, order, type, and array size with the corresponding arguments in the SUBROUTINE or ENTRY statement of the subprogram being called.

General Rules

1. For purposes of optimization, all subroutine calls are treated as abnormal function references.
2. The arguments can be any of the following forms:
   - constant
   - subscripted or nonsubscripted variable
   - array name
   - arithmetic or logical expression
   - FUNCTION or SUBROUTINE subprogram name
   - statement number or switch variable preceded by a $ for an alternate return (e.g., $5)
3. An argument can be omitted and indicated by a successive comma in the argument list. Null argument(s) will appear before the alternate returns in the object code, no matter where they appear in the argument list.

Example

```
CALL SUBR (B, J, $30, $5,)
```

4. Any reference within the called subprogram to a null argument in the CALL will be considered undefined.

5. The calling arguments generated for the alternate returns in the object code are in the reverse order from their appearance in the argument list in the source program. This reverse order must be considered if GMAP subroutines are called from FORTRAN programs.

Examples

```
CALL OUTPUT
CALL ABC(X,B,,C,$5,$200)
CALL QST(9.732,Q/4.536,R-S**2,X1)
CALL SUB(A,I,$10,$20,)
```

In the last example, the code generated will be

```
TSX1  SUB
TRA   *+6
ZERO  .E.L..,6
ARG   A
ARG   I
ARG   0
TRA   .S20
TRA   .S10
```

CHARACTER

The CHARACTER statement declares the variable(s) to be of type character and defines the maximum character length of each variable.

Format

CHARACTER [*integer-1] name *integer-2(dim) [/data/] [,...]

Syntax Rules

1. Integer-1 must be an unsigned integer constant which defines the maximum number of characters of all variables in the statement, unless they are specified by integer-2 (see #3 below).
2. Name can be a scalar, an array, or a FUNCTION subprogram name.
3. Integer-2 is an unsigned integer constant whose value determines the maximum number of characters that can be contained within the character variable specified. An adjustable size specification is permitted within a subprogram when both the character variable and its size parameter(s) are included as dummy arguments.
4. Dim specifies the dimensions necessary to allocate storage to an array.
5. Data is the initial data value.

General Rules

1. Adjustable size specifications are not permitted as the size specification for a character function.
2. If a comparison is made between character fields of unequal length, the smallest field will be left-justified and blank-filled to the size of the larger field. Then a comparison is made.
3. Each CHARACTER variable (scalar or array element) begins on a word boundary. For example, CHARACTER*2 A(2) would use two words of storage; the first two bytes would be used for each word.
4. The maximum number of characters in a variable or array element is 500 for ASCII and 511 for BCD.

Example

```
CHARACTER ARRAY*14(10,10)
CHARACTER A*I(J,4),B*I
```

NOTE: The length of A and B are variable.

COMMON

The COMMON statement assigns variables in different program units to the same memory storage location(s). This can be done in a labeled or blank common area.

Format

COMMON[/]x[/]array[,...]

Syntax Rules

1. Common x is a symbolic name if it is labeled common, or null if it is blank common.
2. Array is the non-empty list of scalar names, array names, or array declarators.
3. If common x is empty, the first two slashes are optional.
4. Labeled and blank common can be included in the same COMMON statement.

General Rules

1. A double precision or complex data item is allocated as two consecutive storage locations.
2. A real, logical, or integer entity is allocated as one storage location.
3. A character variable is allocated the number of consecutive storage locations required to contain the specified number of characters.
4. The following rules apply to blank and labeled common blocks with the same number of storage locations.

   a. In all program units giving the same type to a given position (counted by the number of preceding storage units), references to that position refer to the same value.

   b. A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

c. Complex and double precision entities are assigned consecutive storage locations (pairs) such that the first word of the pair has an even storage address.

d. The size of a common block must not exceed 131,071 decimal words.

5. All variables specified in a COMMON statement are assigned to storage in the sequence in which the names appear in the COMMON statement.

<u>Examples</u>

```
COMMON A,B,C,D          (assigned to blank common)
COMMON/X/A,B,C          (assigned to labeled common)
COMMON A,B,C/Y1/D,E     (A, B, C are assigned to blank common;
                        D, E are assigned to labeled common block Y1)
```

COMPLEX

The COMPLEX statement is an explicit type statement which is used to assign the complex numeric properties to specific variables.

Format

COMPLEX name[*size][(dim)][/data/],...

Syntax Rules

1. Name is a variable, an array, or a FUNCTION subprogram name.
2. Size is an optional size in byte designation and is ignored.
3. Dim supplies the dimensions necessary to allocate storage to arrays.
4. Data represents the initial data values.

Examples

```
COMPLEX  T,N1,D1/(5.0,0.0)/
COMPLEX  ARRAY (10,3)/5/
```

CONTINUE

The CONTINUE statement is a dummy statement most often used as the last statement in the range of a DO loop. The presence of the CONTINUE statement enables a continuation of the normal execution sequence.

Format

[label] CONTINUE

Syntax Rule

Label is an unsigned integer constant.

Example

```
10  DO 12 I = 1,10
    IF (ARG - VAL(I)) 12,13,12
12  CONTINUE
```

DATA

A DATA statement is used to assign values to variables and arrays at compilation time.

Format 1

DATA variable/data/[,...]

Syntax Rules

1. Variable specifies the variables to be initialized and may consist of scalars, arrays, and/or array elements.
2. Data specifies the data constants which may be signed or unsigned. Data may also be specified as j*c where j is a repeat modifier which specifies that constant c is to be used j times. j must be an integer constant or parameter symbol.
3. Variable cannot specify dummy arguments or names in blank common.
4. Data can specify any constant type. Type checking is performed to verify that a variable is initialized with a constant of the correct type. However, octal or character constants can be used to initialize variables of any type.

Format 2

DATA (array k(i),i=m_1,m_2,m_3)/data/

Syntax Rules

1. Format 2 is used to initialize values in an array by the use of an implied DO statement.
2. Array designates the name of the array to be initialized.
3. I indicates the induction variable to be used.
4. M_1 indicates the initial parameter of the implied DO statement.
5. M_2 designates the terminal parameter of the implied DO statement.
6. M_3 specifies the increment parameter for the implied DO statement; if not specified, an increment of one is assumed.

   NOTE: Refer to the DO statement for an explanation of DO parameters.

General Rules

1. Character variables must be initialized with character constants. If the sizes of these two elements differ, truncation or blank-filling will occur.

2. DATA defined variables that are redefined during execution assume their new values regardless of the DATA statement.

3. When values are to be assigned to an entire array, the name of the array is specified without any subscript information. The number of constant values assigned must equal the number of elements in the array.

4. DATA statements which appear in a BLOCK DATA subprogram can pre-set data into labeled common storage only. Only 63 common areas can be pre-set from a single BLOCK DATA subprogram.

5. DATA statements which appear in a program unit that is not a BLOCK DATA subprogram can pre-set data into local storage locations for that program unit or into labeled common. The maximum number of common areas in this case is 62.

6. Type statements can also be used to initialize data values, and must follow the rules specified for the DATA statements.

7. Logical constants can be specified in the data list as T or F, as well as .T. or .F.

   NOTE: This could cause confusion if a variable T or F also appears in a variable statement, because the compiler will use the parameter T or F value in the data list.

8. There must be a one-to-one correspondence between the list items and the data constants. If a non-character type variable is to be initialized with a character constant, and the constant is more than one word of storage, the variable must appear as an array element reference. The constant will be assigned to consecutive locations in memory which begin with the referenced location specified in the array.

Examples

```
DATA A,B,C/14.7,62.1,1.5E-20/
            or
DATA A/14.7/,B/62.1/,C/1.5-E20/
```

initially assigns the value 14.7 to A, 62.1 to B, and 1.5E-20 to C

```
DATA I/3/,J/5/
```

initially assigns the value 3 to I, and 5 to J

```
DATA ZERO,(A(I),I=1,5),A(9)/0.0,5*1.0,100.5/
```

assigns the value 0.0 to ZERO, 1.0 to the first five elements of A, and 100.5 to the ninth element of A

```
INTEGER G(5)
DATA G(1)/15HDATA TO BE READ/
```

NOTE: There is a one-to-one relationship specified (one variable and one constant), but locations G(1) through G(3) will be affected if the mode is BCD because the constant is larger than one word of storage can accommodate.

```
DIMENSION B(25)
DATA A,B,C/24*4.0,3.0,2.0,1.0/
```

assigns the value 4.0 to A and the first 23 elements of B, 3.0 to B(24), 2.0 to B(25), and 1.0 to C.

DECODE

The DECODE statement is used to convert a character string which begins in a specified location to a specified data type as designated by a format. This converted character string is then stored in the given list. (Refer to Section V for additional information on the DECODE statement.)

Format

DECODE (char,form,err) list

Syntax Rules

1. Char is a character variable that can be a scalar, an array element, or an array name. It indicates the beginning location of the internal buffer (i.e., the sending field).
2. Form can be a FORMAT statement number, a character scalar, or an array name. It provides the format specification for decoding.
3. Err is the error transfer option which is designated as ERR=S1. S1 is the statement label or switch variable that is to receive control when an error condition is encountered.
4. List is the receiving field with the same requirements as the list which is specified for the READ statement.

General Rule

The format and the list specified should not require more characters than the number of characters assigned to char. If char is an array, the format information should not require more characters than the number of characters in a <u>single</u> element of char.

Examples

```
    CHARACTER*6A(3)
    A(1) = "111111"
    A(2) = "222222"
    A(3) = "333333"
    DECODE (A,100) I1, I2, I3
100 FORMAT (I6/I6/I6)
    DECODE (A,200) J1,J2,J3
200 FORMAT (3I6)
     .
     .
     .
```

After execution, I1 = 111111, I2 = 222222, I3 = 333333; J1 = 111111, but J2 and J3 are undefined.

```
  A(1) = "ƀƀƀ1"
  DECODE (A,4)I
4 FORMAT (I4)
```

After execution, the array A is not altered, but the integer variable I has the value of 1.

```
        CHARACTER A*4(4),B*1(16)
        DATA A/4*"ABCD"/,B/16*"X"/
        PRINT 9,B
        DECODE (A,4,ERR=100)B
      4 FORMAT (4A1/4A1/4A1/4A1)
        GO TO 11
    100 PRINT, "ERROR"
        STOP
     11 PRINT 9,B
      9 FORMAT (1X,16A1)
        STOP
        END

*RUN
XXXXXXXXXXXXXXXX
ABCDABCDABCDABCD
```

The elements of array A have been placed in array B.

The DECODE statement with the ERR= option can be used to scan individual fields to see, for instance, if a particular field is numeric.

```
   CHARACTER TEXT*35(10)
   INTEGER DATA (50)
   DECODE (TEXT,10,ERR=20) DATA
10 FORMAT (5I7)
```

If any elements of the field being decoded contain nonnumeric characters, control will be transferred to statement number 20.

DIMENSION

The DIMENSION statement is used to specify the maximum size of an array and allocate the necessary storage locations for the array. This statement may also be used to assign initial values to array elements.

Format

DIMENSION array(integer)[/ constant [,...] /]

Syntax Rules

1. Array is the name of an array.
2. Integer is the dimension of the array which is composed of one to seven unsigned integer constants, integer parameters, or integer variables.
3. Constant specifies an optionally signed data constant.
4. Integer variables can be used as a dimension for an array only when the DIMENSION statement appears in a subprogram with the dimensions passed as arguments and the array is not in a COMMON area.

General Rules

1. The DIMENSION statement must precede the first use of the array in an executable statement.
2. One DIMENSION statement can specify the dimensions for several arrays.
3. If the dimensions for a variable are designated in a DIMENSION statement, they cannot be designated in any other statement.
4. Dimensions can also be declared in a COMMON or a type statement. Under these conditions, all the rules for the DIMENSION statement apply.
5. The data constants are optional and apply to the array that immediately precedes them in the DIMENSION statement.

Examples

```
DIMENSION A(50)
DIMENSION B(1,2,3),C(10)/10*1./
DIMENSION D(2,2,3,3,4,4,5)

SUBROUTINE SUB (A,B,I,J)
DIMENSION A(I,4,J),B(J)
```

DO

The DO statement is used to execute a section of a program unit repeatedly with an automatic change in the value of a variable between repetitions.

Format

DO label variable = m_1,m_2,m_3

Syntax Rules

1. Label is the statement label of the terminal statement of the DO loop.
2. Variable is the induction variable and must be a nonsubscripted integer variable.
3. M_1, m_2, and m_3 are referred to as induction parameters or control parameters and can be specified as arithmetic expressions. These parameters are truncated to integer values before execution of the DO.
4. If m_3 is omitted, its value is assumed to be 1.
5. The values of m_1, m_2, and m_3 must all be non-negative but m_3 must not have a value of zero.
6. M_1 cannot be the constant zero, but it can be a variable with the value of zero. If $m_2 \leq m_1$, the loop will be executed once.

General Rules

1. A DO statement is used to define a loop. The action which occurs during the execution of a DO statement is described in the following steps.

   - Variable is initially assigned the value of the initial parameter, m_1.
   - The instructions, as specified within the range of the DO loop, are executed $\frac{m_2 - m_1 + 1}{m_3}$ times.
   - The induction variable is incremented by the value specified by the step parameter, m_3.

- If the value of the induction variable $\leq$ the terminal parameter, m_2, the instructions specified within the range of the DO loop are executed again. If the value is $>$ m_2, the DO loop has been satisfied and control passes to the statement following the terminal statement of the range of the DO.

- If the above situation applied to a nested DO loop which had the same terminal statement, control would pass to the next outer DO loop. The induction variable of this DO statement will be incremented by the corresponding m_3. This process continues until all DO loops which reference the termination statement are satisfied.

- If the exit from a DO loop occurs through a transfer statement, the value of the induction variable is equal to the most recent value assigned which occurred prior to the exit. The DO is said to be "not satisfied" and the induction variable is defined.

  NOTE: If the upper limit of the induction variable is reached, the DO is satisfied, and the induction variable is undefined.

2. The terminal statement cannot be a GO TO, RETURN, STOP, or DO statement.

3. The terminal statement can be an arithmetic IF statement with at least one null field. The null path is a simulated CONTINUE statement which terminates the DO loop.

4. The range of a DO loop begins with the first executable statement following the DO and ends with the terminal statement specified in the DO statement (i.e., label).

5. Another DO statement can be included within the range of a DO loop. However, the range of the inner DO loop must be contained within the range of the outer DO loop. This condition is referred to as a nested DO loop.

6. The control parameters (i.e., variable , m_2, m_3) cannot be redefined within a loop or within the extended range of a loop.

7. A DO statement has an extended range if both of the following conditions exist:

   - There exists at least one transfer statement inside the range of a DO which will cause control to pass out of the DO loop, or out of the nest if the DO loop is nested.

   - There exists at least one transfer statement not in the range of a DO loop or a nested DO loop which can cause control to return into the range of this loop.

If these conditions exist, the extended range consists of all the executable statements that can be executed between the two control statements. The statements which satisfy the first condition are not included in the extended range; the statements which satisfy the second condition are in the extended range.

NOTE: The use of extended range DO loops should be minimized, especially when global optimization is desired.

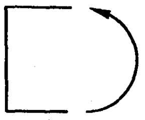
8. A transfer statement cannot cause control to pass into the range of a DO loop unless the transfer statement being executed is part of the extended range of that particular loop. In addition, the extended range of a DO loop may not include another DO statement which contains an extended range, or a DO statement that has the same induction variable.

9. When a procedure reference occurs in the range of a DO loop, the actions of that procedure are considered to be temporarily within that range (i.e., during the execution of that reference).

Examples

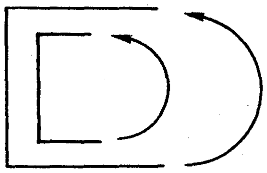
Standard DO Statement

```
     DO 6 I=1,10
     .....
   6 CONTINUE
```



Nested DO Loop

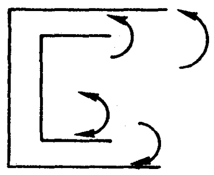
```
      DO 60 I=10,20,2
      K=I+3
      DO 10 J=2,50,10
   10 M=J+6
   60 CONTINUE
```



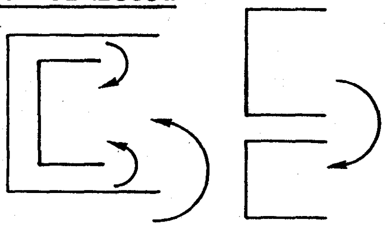
Transfer of Control

The following configurations show permitted and nonpermitted transfers.

Permitted



Not Permitted



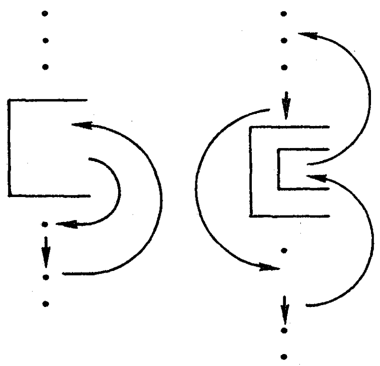
Extended Range

```
          .
          .
          .
          DO 20 I=1,K
          DO 20 J=N,M
          .
          .
          .
          IF (J-JJ),80,
          .
          .
          .
       20 CONTINUE
          .
          .
          .
       80 .          (extended range of a nested
          .
          .           DO loop)
          GO TO 6
```

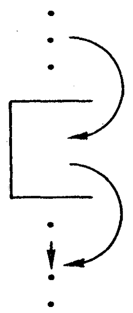
Transfer of Control for Extended Range

The following configurations show permitted and nonpermitted transfers for an extended range.

Permitted



Not Permitted



DOUBLE PRECISION

The DOUBLE PRECISION statement is an explicit type statement which is used to assign double precision numeric properties to specified variables.

Format

```
DOUBLE PRECISION  variable[*size][(dim)][/data/] ,...
```

Syntax Rules

1. Variable is a scalar, an array, or a FUNCTION subprogram name.
2. Size is an optional size in bytes qualification, and it is ignored.
3. Dim gives the dimensions needed to allocate storage for the arrays.
4. Data gives the initial data value(s).

General Rules

Variables that are declared in this statement could also be declared via the REAL statement with a size qualifier $\geq$ 8.

Examples

```
DOUBLE PRECISION DENOM,PREF/1.6D0/
DOUBLE PRECISION DB(10)
```

ENCODE

The ENCODE statement is used to convert data under the control of a specified format and store the encoded data as type character.

Format

ENCODE (char,form,err) list

Syntax Rules

1. Char is a character variable that can be a scalar, an array element, or an array name. It indicates the starting location of the internal buffer which is the receiving field for encoding.
2. Form can be a FORMAT statement number, a character scalar, or an array name that provides the character formatting information of the sending field for encoding.
3. Err is the error transfer option, designated as ERR=S1, where S1 is the statement label or switch variable that is to receive control when an error condition is encountered.
4. List is the sending field for encoding and has the same requirements as the list specified for the WRITE statement.

General Rules

1. The number of characters generated by form and list should not exceed the number of characters designated by char.
2. ENCODE does not blank-fill the word to the word boundary like a READ statement does.
3. Any numerical variable in the list with a value requiring more space than specified by form, will be replaced by asterisks in the storage locations beginning with char (refer to Numeric Field Descriptions in Section V). If this procedure is necessary, as in the case of developing leading zeroes for the character form of a numeric data item, then the CALL NASTRK and CALL YASTRK statements will be required to enable the ENCODE statement to function as desired (refer to Section VI for a list of supplied SUBROUTINE subprograms).

Example

```
          CHARACTER A*4
          I=1
          ENCODE (A,3,ERR=100)I
        3 FORMAT (I4)
          GO TO 11
      100 PRINT, "ERROR"
          STOP
       11 PRINT 9,A
        9 FORMAT (1X,A4)
          STOP
          END
```

After execution, A will contain

```
ØØØ1
```

where: Ø indicates a blank.

END

The END statement is used to indicate the physical end of the source program.

Format

END

General Rules

1. END must be the last statement of every source program unit.
2. END creates no object-program instructions.

Example

```
.
.
.
STOP
END
```

Syntax Rules

1. There cannot be any other non-blank characters in the END statement (e.g., END 05 is illegal).
2. END; cannot be specified as the first statement of a multi-statement line.

ENDFILE

The ENDFILE statement is used to close a sequential file with an end-of-file record indicator.

Format

<u>ENDFILE</u> file

Syntax Rules

1. File is a two-character file code which references the file to be closed.
2. File is the file reference for a sequential output file.
3. File must be an integer constant, an integer variable, or an expression.

General Rules

1. When the ENDFILE statement is encountered, the buffer(s) is flushed and a file-mark is written for the output file.
2. Executing an ENDFILE on an input file with read only permission will result in an error message:

   "Impermissible perm-write" in batch
   "Write attempted read only file" in TSS

Examples

```
ENDFILE 5
ENDFILE JPAY
```

ENTRY

The ENTRY statement is used to define alternate entry points into a subroutine or a function subprogram.

Format

ENTRY name [(arg [,...])]

Syntax Rules

1. Name is the symbolic name of an entry point into a subroutine or function subprogram. Name must be unique within the first six characters.
2. Arg is a dummy argument which corresponds to an actual argument in a CALL statement or a function reference. Entry into a FUNCTION subprogram must have at least one argument.
3. An asterisk can be used as an argument in an ENTRY statement of a SUBROUTINE subprogram to indicate an alternate return.

General Rules

Multiple entry points must conform to the following rules:

1. In a FUNCTION subprogram, only the FUNCTION name can be used as the variable to return the function value to the using program. The ENTRY name cannot be used for this purpose.
2. An ENTRY name can appear in an EXTERNAL statement in the same manner as a FUNCTION or SUBROUTINE name.
3. Entry into a subprogram defines all arguments in the ENTRY statement, for the entire subprogram, from the argument list of the corresponding CALL statement or FUNCTION reference.
4. The appearance of an ENTRY statement does not alter the rules for placing arithmetic statement functions in subroutines.
5. Arg cannot appear in an EQUIVALENCE or COMMON statement in the same subprogram.

Examples

```
ENTRY NAM(A,*,X)
ENTRY SUB2
```

EQUIVALENCE

An EQUIVALENCE statement is used to assign two or more variables within the same program unit to the same storage location.

Format

EQUIVALENCE $(\text{var}_1,\text{var}_2\ [,\ldots]\ )$

Syntax Rule

Var can be either a scalar, an array, or an array element. If var is an array element, the subscripts must be integer constants or parameter symbols.

General Rules

1. Each pair of parentheses must enclose the names of two or more variables that are to be assigned the same location during execution of the object program; any number of equivalences (sets of parentheses) can be given.

2. When var is an array element, the subscript can be specified in two ways. D(1,2,1) or D(p) can be used to specify the same element, where D(p) references the p_i element of the array in storage (refer to Section II for a description of the Array Element Successor Function).

3. Quantities or arrays not specified in an EQUIVALENCE statement are assigned unique storage locations.

4. Storage locations can only be shared by variables; not by constants.

5. There are six statements in FORTRAN which cause a new value to be stored in a location (i.e., defined or redefined):

   - The execution of an arithmetic assignment statement stores a new value in the location assigned to the variable which is on the left side of the equal sign.

   - The execution of a DO statement or an implied DO in an input/output list will sometimes store a new value for the induction variable.

   - The execution of a READ or DECODE statement stores new values in the locations which are assigned to the variables in the input list.

   - The execution of an ENCODE statement stores new values in the character variable or the array location(s) which are named as the internal buffer (i.e., receiving field).

- The execution of a CALL statement or an abnormal function reference may assign new values to variables in common or to arguments which are passed to that subprogram.

- An initial value can be stored in a location via a DATA statement, or a data clause in a type statement.

6. Variables which are brought into a common block through an EQUIVALENCE statement can increase the size of the block indicated by the COMMON statement.

   Example

   ```
   COMMON /X/A,B,C
   DIMENSION D(3)
   EQUIVALENCE (B,D(1))
   ```

   The layout of storage indicated by this example (extending from the lowest location of the block to the highest location of the block) is

   ```
   A
   B,D(1)
   C,D(2)
     D(3)
   ```

7. Because arrays must be stored in consecutive locations, a variable cannot be made equivalent to an element in an array if it would cause the array to extend below the beginning of a common block.

8. To make a double-word variable equivalent to a single-word variable, the following rules apply:

   - The effect of the EQUIVALENCE statement(s) must cause the first word of any double-word variable to be an even number of locations from the beginning of the space allocated for data (common or local).

   - The effect of the EQUIVALENCE statement must cause the first word of any double-word variable to be an even number of words from the beginning of any other double-word variable which is linked to it through an EQUIVALENCE statement.

9. Two variables in the same common block or two different common blocks cannot be made equivalent.

10. The EQUIVALENCE statement does not make the data items specified mathematically equivalent.

11. Var cannot be specified as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement.

Examples

```
EQUIVALENCE (A,B,C)
COMMON /X/A,B,C
DIMENSION D(3)
EQUIVALENCE (A,D(1))
```

The same storage locations will be shared by

```
A and D(1)
B and D(2)
C and D(3)
```

```
DIMENSION B(5),C(10,10),D(5,10,15)
EQUIVALENCE (A,B(1),C(5,4)),(D(1,4,3),E)
```

The same storage locations will be shared by

```
A,B, and C(5,4)
D(1,4,3) and E
```

EXTERNAL

The EXTERNAL statement is used to distinguish a FUNCTION or SUBROUTINE name from a variable name when it is used as an argument to a subprogram call.

Format

```
EXTERNAL  sub [(ABNORMAL)][,...]
```

Syntax Rules

1. Sub is a subprogram name.
2. If the ABNORMAL option is specified, the subprogram is defined as both EXTERNAL and ABNORMAL.

General Rules

1. A SYMREF is generated for the subprogram name in the object code.
2. An EXTERNAL statement must be included when a subprogram is used as an argument in a CALL statement.

Examples

```
EXTERNAL SUB,SQRT(ABNORMAL)
```

Main Program

```
   EXTERNAL SIN, COS
   CALL SUBR (2.0, SIN, RESULT)
   WRITE (6, 10) RESULT
10 FORMAT ("0 SIN(2.0 = ", F10.6)
   CALL SUBR (2.0, COS, RESULT)
   WRITE (6,20) RESULT
20 FORMAT ("0 COS(2.0) = ", F10.6)
   STOP
   END
```

SUBROUTINE Subprogram

```
SUBROUTINE SUBR (X,F,Y)
Y = F(X)
RETURN
END
```

FORMAT

The FORMAT statement is used to specify the conversion and editing information for variable lists in I/O statements, as well as DECODE and ENCODE statements.

Format

$$\text{label } \underline{\text{FORMAT}}\left(\left\{ \begin{array}{l} sl_1\ des_1\ sep_1[,\ldots] \\ V \\ \cancel{b} \end{array} \right\} \right)$$

Syntax Rules

1. Label is a unique statement label which identifies each format statement that is referenced by an input/output or ENCODE/DECODE statement.

2. Sl_1 can be a series of slashes to indicate the number of lines or input records to be skipped.

3. Des_1 is one or more of the following field descriptors:

nPr D w.d nPr E w.d nPr F w.d nPr G w.d	Numeric and Logical Field Descriptors
r A w r I w r L w r O w r R w w H h h ... h "h h ... h_w" 'h h ... h_w'	Character Field Descriptors
Tt nX	Field Positioning Descriptors

where: P is an optional scale factor designator
r is an optional repeat count
w is the field width expressed in number of characters
d is the number of fractional places (characters)
h is a single character
t is a character position where the positions of a line are numbered 1 through the number indicated
n is a signed integer constant in the range
$-8 \leq n \leq 8$ for nP
$n \leq 1$ for nX
F,E, and G indicate REAL values
D indicates double precision
O indicates octal conversion is necessary
I indicates an integer value

L indicates LOGICAL values
A,R, and H are for character values
X and T indicate text to be skipped

NOTE: H, T, and X do not require a variable in the I/O list, but all others do.

4. Sep_1 is a field separator (i.e., a comma, a slash, or a series of slashes).

5. If the V option is used, the formatted I/O is under list control. List directed input and output can also be performed by omitting a FORMAT reference (i.e., "READ", "PRINT", or "PUNCH").

6. The ( ) option is the same as the (V) option.

General Rule

The field descriptors are formed in the following ways:

Fw.d	= Real mode without an exponent
Ew.d	= Real mode with an exponent
Gw.d	= F or E editing code is taken dependent upon the value of the output item
Dw.d	= Double precision mode with an exponent
Ow	= Field occupies w print positions and is represented as an octal number of up to 12 digits
Iw	= Integer mode and field occupies w print positions
Lw	= Right-most position of field w contains T or F for a logical variable
Aw	= Field occupies w print positions with left-justified character data
Rw	= Field occupies w print positions with right-justified character data
nH	= Hollerith field which occupies n print positions
Tt	= Next operation begins with position t of the record
nX	= Field of n characters is blank-filled for output; skipped for input

Examples

```
10 FORMAT (E17.2,F20.0)

   WRITE (6,12)PAY
12 FORMAT (//15HPAY IS EQUAL TO, F6.2)

14 FORMAT (V)

   READ (5,16)HRS,RATE,NO
16 FORMAT (F3.2,F4.2,I6)
```

FUNCTION

The FUNCTION statement is used to define a FUNCTION subprogram.

Format

$$\left[\left\{\begin{array}{l}\underline{\text{REAL}}\\ \underline{\text{INTEGER}}\\ \underline{\text{DOUBLE PRECISION}}\\ \underline{\text{COMPLEX}}\\ \underline{\text{LOGICAL}}\\ \underline{\text{CHARACTER}}\end{array}\right\}\right]\quad \underline{\text{FUNCTION}}\quad \text{name}\ (\ \text{arg}[,\ldots]\ )$$

Syntax Rules

1. Name is the symbolic name of a single-valued function.
2. Arg is an argument which can be a non-subscripted variable or array name, or the dummy name of a SUBROUTINE or FUNCTION statement.
3. Name must be a unique name which does not exceed six characters.
4. The length of a character function can be specified through a type statement, or calculated within the function subprogram.

   Example

   ```
   FUNCTION X(A,B)
   CHARACTER X*12
   ```

General Rules

1. The FUNCTION statement must be the first statement of a FUNCTION subprogram. At least one argument must be specified.
2. Name must appear at least once in some assignment context so that the value of the function is returned to the calling program.

3. Arg can be considered as a dummy variable name(s) that is replaced at the time of execution by the actual arguments which are given in the function reference in the calling program. The actual arguments must correspond to the dummy arguments in number, size, and type.

4. When a dummy argument is an array name, a statement with dimension information must appear in the FUNCTION subprogram, and the corresponding actual argument must be a dimensioned array name.

5. A dummy argument cannot appear in an EQUIVALENCE, NAMELIST, or COMMON statement in the FUNCTION subprogram.

6. The FUNCTION subprogram must be logically terminated by a RETURN statement and physically terminated by an END statement.

7. The FUNCTION subprogram can contain any FORTRAN statements except SUBROUTINE, BLOCK DATA, another FUNCTION statement, or a RETURN statement with an alternate return specified (e.g., RETURN 1).

8. A FUNCTION subprogram is referred to by using its name as an operand in an arithmetic expression and following it with the required actual arguments enclosed in parentheses.

9. A FUNCTION subprogram cannot call itself, either directly or indirectly, through some other called subprogram.

10. The FUNCTION must be assigned a value before the return to the calling program.

11. The actual arguments given in the function reference can be any of the following:

    - Constant
    - Scalar variable or nonsubscripted array name
    - Arithmetic or logical expression
    - FUNCTION or SUBROUTINE subprogram name
    - Omitted or null argument, which is indicated by successive commas (e.g., FUNCTION CALC (A,,B,,)). References to null arguments from within the called function are undefined.

NOTE: Refer to Tables 6-2 and 6-3 for a list of the Supplied FUNCTION Subprograms.

Examples

```
FUNCTION ARSIN (RADIAN)
REAL FUNCTION ROOT (A,B,C)
INTEGER FUNCTION CONST (ING,SG)
DOUBLE PRECISION FUNCTION DBLPRE (R,S,T)
COMPLEX FUNCTION CCOT (ABI)
LOGICAL FUNCTION IFTRV (D,E,F)
```

```
Calling Program          Called Function

       .
       .                 FUNCTION CALC (A,B)
       .                        .
X=Y**2+D*CALC(F,G)              .
       .                        .
       .                 CALC=A**B/2
       .                        .
STOP                            .
END                             .
                         RETURN
                         END
```

GO TO

The GO TO statement is used to indicate the next statement in the same program unit to be executed. The GO TO may be expressed as an unconditional, an assigned, or a computed statement.

Format 1

Unconditional

GO TO label-1

Syntax Rules

Label-1 is an executable statement label and will be the next statement to be executed.

Format 2

Assigned

GO TO var [,(label-2 [,...])]

Syntax Rules

1. Var is a switch variable.
2. Label-2 is a list of one or more executable statement labels. If this option is specified, var must have been assigned the value of one of the labels in label-2 by the ASSIGN statement.
3. The next statement to be executed will be the one with the statement label equal to var.
4. If a statement label has been assigned to var that is not in the label-2 list, a compile time diagnostic is generated.

Example

```
ASSIGN 23 TO I
   .
   .
   .
GO TO I,(12,23,48)
```

will result in a run-time Q6 abort.

Format 3

Computed

GO TO (label-3 [,...]), exp

Syntax Rules

1. Label-3 is the label of an executable statement or a switch variable.
2. Exp is an arithmetic expression which is truncated to an integer value at the time of execution.
3. The next statement to be executed will be label-3_i, where i is the integer value of exp.
4. In the expression $0 \leq i \leq n$, if i is out of the range, a diagnostic is generated and execution is terminated.

Example

```
J=3
GO TO (5,4,17,1),J     (Statement 17 is executed next)

I = 4
GO TO (5,4,4,1,3),I    (Statement 1 is executed next)
```

General Rules

1. Label-1, label-2, and label-3 can be the label of any executable statement within the same program unit that appears before or after the GO TO statement, but is subject to the rules for transferring into and out of DO loops.
2. Control is transferred unconditionally to the statement number.

Examples

```
GO TO 5                  (Statement 5 is executed next)

ASSIGN 17 TO J
   .
   .
   .
GO TO J,(5,4,17,2)       (Statement 17 is executed next)

J=2
GO TO (5,4,17,1),        (Statement 4 is executed next)
```

IF

The IF statement is used to determine a path in the execution sequence. An arithmetic IF statement causes a change in the execution sequence based upon the resulting value of an arithmetic expression. A logical IF statement causes a conditional change in the execution sequence based upon the true or false value of a logical expression.

Format 1

Arithmetic IF

```
IF ( exp-1 ) label-1,label-2,label-3
```

Syntax Rules

1. Exp-1 is an arithmetic expression.
2. Label-1, label-2, label-3, can be a statement label, switch variable, or null. If label-1, label-2, and label-3 are null, control will pass to the first executable statement directly following the IF statement.
3. Execution will branch to
   - label-1 if the value of exp-1 < zero
   - label-2 if the value of exp-1 = zero
   - label-3 if the value of exp-1 > zero
4. A maximum of two statement labels can be null.

Format 2

Logical IF

```
IF ( exp-2 ) state-2
```

Syntax Rules

1. Exp-2 is a logical or relational expression.
2. State-2 may be any executable statement except a DO statement or another logical IF statement. It is called the truth clause.

3. When the IF statement is executed, exp-2 is evaluated. If the result is true, state-2 is executed. Otherwise, control passes to the first executable statement which follows the IF statement.

General Rules

1. If the operator .NE. or .EQ. is contained in a logical IF expression, and the operands are not type integer or type character, a warning message appears at the end of the source program listing. Because floating-point arithmetic is not exact for some fractions, the equality or non-equality relation between the operands may not be meaningful.

2. If a relational IF expression compares two character strings of unequal length, the shorter string is left-justified and blank-filled to equal the length of the longer string before the comparison is made.

IMPLICIT

The IMPLICIT statement is used to redefine the default implied data types of all variable and function names (with the exception of supplied intrinsic and supplied mathematical functions) in the program unit that begin with the letters specified.

Format

<u>IMPLICIT</u> type*size (arg[,...]) [,...]

Syntax Rules

1. Type must be one of the following keywords:

   - INTEGER
   - REAL
   - COMPLEX
   - DOUBLE PRECISION
   - LOGICAL
   - CHARACTER

2. Size is an optional unsigned integer constant that designates the length of the associated data type for REAL and CHARACTER; this field is ignored for all other types. When type is REAL, a specified length of eight or more implies DOUBLE PRECISION. When type is CHARACTER, the specified length is as defined for the CHARACTER statement.

3. Arg is one or two alphabetic characters. If two characters are indicated, they are separated by a dash (e.g., A-B).

General Rules

1. An IMPLICIT statement supersedes all other previous IMPLICIT statements referencing the same letters.

2. The IMPLICIT statement <u>must</u> appear before any use of the variable being typed. However, it does not override explicit type statements.

3. Supplied intrinsic and supplied mathematical functions are not affected by IMPLICIT statements.

4. The IMPLICIT statement will apply to all variable names which begin with the letters indicated, or the series of letters indicated by the dash (i.e., A-H will apply to all variable names which begin with the letters A through H).

Examples

```
IMPLICIT INTEGER (A-F,X,Y)
```

Any variable name not typed by an explicit type statement, and first appearing in the program following this statement, and beginning with the letters A through F, or X, or Y, is implicitly typed INTEGER. This typing also applies to variable names beginning with the lowercase letters a through f, x, and y.

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

Any variable name not typed by an explicit type statement, and first appearing in the program following this statement, and beginning with the letters A through H, or O through Z, is implicitly typed DOUBLE PRECISION. This typing also applies to variable names beginning with the lowercase letters a through h, and o through z.

NOTE: If the IMPLICIT statement immediately follows either a SUBROUTINE or FUNCTION statement, the arguments of the subroutine or function are affected by the IMPLICIT typing. This practice is not recommended.

Example

```
SUBROUTINE SUB(J,N)
IMPLICIT REAL(I-N)
B = J*N
RETURN
END
```

Within the subroutine SUB, the variables J and N would be typed as REAL. However, if another statement were inserted between the SUBROUTINE and IMPLICIT statements, J and N would be typed as INTEGER. This may cause confusion for the programmer as to the typing of the variables which are used as arguments.

INTEGER

The INTEGER statement is an explicit type statement which is used to assign integer numeric properties to specified variables.

Format

```
INTEGER  var [*size(dim)/data/][,...]
```

Syntax Rules

1. Var can be a scalar, an array, or a FUNCTION subprogram name.
2. Size is an optional size in bytes that is ignored.
3. Dim supplies the dimensions to allocate the necessary storage for arrays.
4. Data gives the initial data value.

Examples

```
INTEGER I,ABC
INTEGER CALC(10),J/6/,XYZ
```

LOGICAL

The LOGICAL statement is an explicit type statement which assigns logical properties to specified variables.

Format

LOGICAL var [*size(dim)/data/][,...]

Syntax Rules

1. Var can be a scalar, an array, or a FUNCTION subprogram name.
2. Size is an optional size in bytes that is ignored.
3. Dim gives the dimensions to allocate the necessary storage for array(s).
4. Data is the initial data value.

Examples

```
LOGICAL A1,K
LOGICAL CALC(25),L/.TRUE./
```

NAMELIST

The NAMELIST statement is used to associate variables and/or arrays for input/output.

Format

NAMELIST /name/var/ [,...]

Syntax Rules

1. Name must be a unique name consisting of one to eight alphanumeric characters, and must be unique for the first six characters.
2. Var must be a list of variables and/or array names to be associated with the corresponding NAMELIST name.
3. Var can belong to one or more namelist names, and can be up to eight characters but the first six must be unique.

General Rules

1. Name cannot be the same as any other variable name in the source program.
2. The NAMELIST statement defining namelist names must precede any reference to the namelist names in the program.
3. Var cannot be a dummy argument in a subprogram.
4. Var cannot be an array name of more than seven dimensions.

Examples

```
NAMELIST/LIST/R,S,T,U,V

DIMENSION A(10),I(5,5),L(10)
NAMELIST/NAM1/A,B,I,J,L/NAM2/A,C,J,K
```

The arrays A, I, and L, and the variables B and J belong to the NAMELIST name, NAM1; the array A and the variables C, J, and K belong to the NAMELIST name, NAM2.

PARAMETER

The PARAMETER statement is used to define program constants as the result of an expression at compilation time.

Format

<u>PARAMETER</u> symbol = exp [,...]

Syntax Rules

1. Symbol is a parameter symbol whose type is dependent on the type of exp.
2. Exp is an arithmetic expression which contains only constants and previously defined parameter symbols.

General Rules

1. The value of the parameter symbol cannot be redefined during the execution of a program.
2. Symbol cannot appear in a FORMAT statement or in any other statement where a constant cannot appear.
3. The significant difference between symbol and an ordinary integer variable that can be initialized with a DATA statement is in the usage. For example, a parameter symbol can be used to specify dimensions.
4. The appearance of a parameter symbol in any context is interpreted as though its equivalent value had appeared instead.

Example

```
PARAMETER I=5/2,J=I*3,K=3.14159,L=.T.,M="06171"

where:  I and J are INTEGER
        K is REAL
        L is LOGICAL
        M is CHARACTER
```

The parameter symbol I is initialized to the value 2, the parameter symbol J is initialized to 6, and the parameter symbol K is initialized to the real value 3.14159. L has the value .TRUE., while parameter symbol M is assigned a CHARACTER value.

Example

```
      PARAMETER I=20
      PARAMETER J=I*4
      DIMENSION A(I,J)
          .
          .
          .
      DO 100 II=1,I
      DO 100 JJ=1,J
  100 A (II,JJ)=0.
          .
          .
          .
```

A is not an adjustably dimensioned array. It has constant dimensions of 20 and 80, respectively. The two DO statements have constant terminal parameter values of 20 and 80, respectively (refer to the DO statement in this section). I and J are compile time variables, while II and JJ are execute time variables. The program properties change as the value of the parameter symbol I changes. To operate on a 10 by 40 array, only the first line needs to be changed.

PAUSE

The PAUSE statement is used to cause a temporary halt in the execution of a program until the operator resumes execution.

Format

PAUSE [char]

Syntax Rule

Char can be a positive integer constant ≤ five digits, an integer variable whose value is ≤ five digits, a character constant, or a character variable. If char is a character constant, it must consist of a character string enclosed by apostrophes or quotation marks.

General Rules

1. When the PAUSE statement is executed, a message is printed on the operator console or TTY terminal consisting of the word PAUSE and the value of char. Execution is continued when the operator hits the carriage return.

2. If char is not specified, or if char is an integer, the snumb and activity number of the job is printed, unless executing under TSS; then they are omitted.

Examples

```
    PAUSE
    PAUSE "TOO BAD"
    PAUSE I
    PAUSE 77777

    SUBROUTINE PAWS(IDENT,MESSAGE)
    CHARACTER MESSAGE*8
    IF (IDENT),100,
    PAUSE IDENT
    RETURN
100 PAUSE MESSAGE
    RETURN
    END
```

A call to the above subroutine

```
     CALL PAWS (77777,0)
```

might display

```
     PAUSE 77777 SNUMB 1234T-02
```

A call of the form

```
     CALL PAWS (0,"ERROR 27")
```

would display

```
     PAUSE ERROR 27
```

PRINT

The PRINT statement is used to direct output to the standard system output device (file code 42).

Format 1

List Directed Output

PRINT, [list]

Syntax Rule

List is the list of variables and/or expressions that is to be directed to the standard system output device (file code 42).

Format 2

Formatted Output

PRINT form [,list]

1. Form must be a FORMAT statement label, a character scalar, or an array name.
2. If list is specified, the information will be converted according to the format specified by form, and directed to the standard system output device (file code 42).

Format 3

NAMELIST Output

PRINT namelist

Syntax Rule

Namelist must be a name specified in a NAMELIST statement. The output will be directed to the standard system output device (file code 42).

Examples

PRINT,	(print a blank line)
PRINT, A	(list directed output)
PRINT 20	(formatted output)
PRINT 20,A	(formatted output)
PRINT LIST	(namelist output)

PUNCH

The PUNCH statement is used to transmit output to the standard system punch device (file code 43).

Format 1

List Directed Output

PUNCH, list

Syntax Rule

List is the list of variables and/or expressions that is to be directed to the standard system punch device (file code 43).

Format 2

Formatted Output

PUNCH form [,list]

Syntax Rules

1. Form must be a FORMAT statement number, a character scalar, or a character array name.
2. If list is specified, the information will be converted according to the format specified by form, and directed to the standard system punch device (file code 43).

Format 3

NAMELIST Output

PUNCH namelist

Syntax Rule

Namelist must be a name specified in a NAMELIST statement.

Examples

```
PUNCH, A       (list directed punch output)
PUNCH 20,A     (formatted punch output)
PUNCH LIST     (namelist punch output)
```

READ

The READ statement is used to direct input data from the standard system input device (file code 41) to be utilized by the program unit.

Format 1

List Directed Input

READ, list

Syntax Rule

List is the input information that will be read from the standard system input device (file code 41).

Format 2

Formatted Input

READ form [,list]

Syntax Rules

1. Form must be a FORMAT statement label, a character scalar, or an array name.
2. If list is specified, the information will be read from the system standard input device (file code 41) and converted to the specified format.

Format 3

NAMELIST Input

READ namelist

Syntax Rules

1. Namelist must be specified in a NAMELIST statement.

2. Input will be directed to the standard system input device (file code 41).

Format 4

Formatted with File Reference Input

READ (file,form [,opt1,opt2]) list

Syntax Rules

1. File is a file reference which is also the file code that can be a positive integer constant, an integer variable, or an integer expression in the range 01 $\leq$ file $\leq$ 63. If 5 or 41 is specified, reference is to the standard system input device.

2. Form must be a FORMAT statement label, a character scalar, or an array name.

3. The opt1 option is designated as END=S1, where S1 is the statement label or switch variable to be executed when an end-of-file condition is encountered.

4. The opt2 option is designated as ERR=S2, where S2 is the statement label or switch variable to be executed when any I/O error is encountered.

5. The options opt1 and opt2 can be specified in any order.

6. List is the input file that will be read from the file.

Format 5

Unformatted with File Reference Input

READ (file[,opt1,opt2]) list

Syntax Rules

1. File is file reference which is also the file code that applies to a word-oriented serial access file (binary sequential).

2. Opt1, opt2, and list follow the rules specified in Format 4.

Format 6

Random File Input

READ (file'n[,opt1,opt2]) list

Syntax Rules

1. File is a file reference which applies to a random binary file. files.

2. N must be an integer constant, a variable, or an expression that specifies the sequence number of the logical record to be accessed.

3. Opt1, opt2, and list follow the rules specified in Format 4.

Format 7

NAMELIST Input with File Reference

READ (file,namelist[,opt1,opt2])

Syntax Rules

1. Namelist must be specified in a NAMELIST statement.

2. File, opt1, and opt2 follow the rules specified in Format 4.

General Rule

A READ statement cannot be executed after a WRITE statement if it uses the same file reference. The sequence, WRITE, REWIND, and then READ is legal.

Examples

```
READ, A                        (list directed input)
READ 20,A                      (formatted input)
READ LIST                      (namelist input)
READ (5,20,END=90,ERR=95)A     (formatted file input)
READ (5,END=90,ERR=95)A        (unformatted file input)
READ (8'I)A                    (random binary file input)
READ (5,LIST)                  (namelist file input)
```

REAL

The REAL statement is an explicit type statement which is used to assign real numeric properties to specified variables.

Format

```
REAL  var [*size(dim)/data/][,...]
```

Syntax Rules

1. Var must be a scalar, an array, or a FUNCTION subprogram name.
2. Size is a size specification and the type is treated as DOUBLE PRECISION if size is > 7.
3. Dim is the dimension information required to allocate the necessary storage for arrays.
4. Data is the initial data value.

Examples

```
REAL J
REAL IARR, MEN
```

RETURN

The RETURN statement is used to denote the logical termination of a subprogram, and thus, return control to the calling program.

Format

RETURN [integer]

Syntax Rules

1. If present, integer must be a positive integer constant or an integer variable.
2. Integer indicates the nth alternate return in the CALL statement argument list, from left to right, of the calling program.
3. Integer cannot be greater than the number of alternate returns in the argument list of the calling program.

General Rules

1. There can be any number of RETURN statements in a subprogram.
2. If integer has a value of zero, a normal return is executed.
3. Integer cannot be specified in RETURN statements that are used in FUNCTION subprograms.

Examples

```
RETURN
RETURN 3
```

REWIND

The REWIND statement is used to position a sequential file to its initial point.

Format

REWIND file

Syntax Rule

File is a file reference and must be specified as an integer constant or an integer variable in the range $01 \leq$ file ≤ 43.

General Rules

1. File must be a sequential file.
2. If file is an output file, an EOF is written on the file before it is rewound. The file is then closed.

Examples

```
REWIND 5
REWIND STAT
```

STOP

The STOP statement is used to halt the execution of an object program unit and return control to the operating system.

Format

STOP [line]

Syntax Rule

Line must be a positive integer constant $\leq$ five digits, an integer variable whose value $\leq$ five digits, a character constant, or a character variable. If line is a character constant, it must consist of a character string enclosed by apostrophes or quotation marks.

General Rule

1. If line is specified, the standard output device prints

   - STOP AT LINE line if line is integer
   - STOP line if line is character.

2. If line is not specified, the program simply terminates.

Examples

```
STOP
STOP 100
STOP STAT
STOP "MESSAGE"
```

SUBROUTINE

The SUBROUTINE statement is used to define a SUBROUTINE subprogram.

Format

SUBROUTINE name [(arg,...)]

Syntax Rules

1. Name is the symbolic name of the subprogram and must be unique within the first six characters.
2. Arg is the subprogram argument and can be a dummy variable or array name, a dummy subprogram name, or a * or $ to indicate an alternate return.

General Rules

1. The SUBROUTINE statement must be the first statement in the subprogram.
2. One or more of the arguments can be used to return values to the calling program. These arguments must be defined within the subprogram in other than a DATA statement.
3. The arguments of the calling program must agree in number, order, size, and type with the subroutine arguments (sometimes referred to as dummy arguments). The arguments can be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement which refers to the SUBROUTINE subprogram.
4. If arg is an array name, a statement with the appropriate dimension(s) must appear in the subroutine subprogram; the corresponding argument in the CALL statement must be a dimensioned array name.
5. Arg cannot be specified in a COMMON, EQUIVALENCE, NAMELIST, or DATA statement within the subroutine.
6. The SUBROUTINE subprogram must be logically terminated by a RETURN statement and physically terminated by an END statement.
7. The SUBROUTINE subprogram can contain any FORTRAN statements except FUNCTION, BLOCK DATA, or another SUBROUTINE statement.

Examples

```
     SUBROUTINE COMP (X,Y,*,$,P)
     SUBROUTINE QUADEQ (B,A,C,H,ROOT)
     SUBROUTINE OUTPUT
```

WRITE

The WRITE statement is used to direct output to an output device.

Format 1

Formatted Output

WRITE (file,form[,opt]) list

Syntax Rules

1. File is a file reference that must be a positive integer (i.e., a constant, a variable, or an expression) of the range $01 \leq$ file ≤ 63.
2. Form must be a FORMAT statement label, a character scalar, or a character array element.
3. The opt option is designated as ERR=S1 where S1 is a statement label, or a switch variable that is to be executed when an error condition is encountered, or an end-of-file is found.
4. List contains the output variables whose values are to be directed to the output device.

Format 2

Unformatted Output

WRITE (file[,opt]) list

Syntax Rules

1. File is a file reference that applies to the output of word-oriented serial access files (binary sequential).
2. Opt and list follow the rules specified by Format 1.

Format 3

Random Output

WRITE (file'n,opt) list

Syntax Rules

1. File is a file reference that applies to a random binary file.
2. N must be a positive integer (i.e., a constant, a variable, or an expression) that specifies the logical record to be written.
3. Opt and list follow the rules specified in Format 1.

Format 4

Namelist Output

WRITE (file,namelist,opt)

Syntax Rules

1. Namelist must be specified in a NAMELIST statement.
2. If namelist is specified, character-oriented records will be directed to the output device.
3. Opt follows the rules specified in Format 1.

General Rule

If file is specified as 6 or 42, the output will be directed to the standard system output print device; 43 will direct it to the standard system output punch device.

Examples

```
WRITE (6,30,ERR=34)A                   (formatted file output)
WRITE (6,ERR=34)A                      (unformatted file output)
WRITE (6,LIST)                         (namelist file output)
WRITE (8'I)A                           (random binary output)
WRITE (N,FMT,ERR=10,END=500)A,B,C      (formatted output)
```

SECTION IV

USER INTERFACES

Programs are created by entering FORTRAN statements into remote and local peripheral or terminal devices connected to a computer operating under GCOS. This procedure is referred to as user interface, with three modes of operation available to the programmer: local batch, remote batch, and time sharing.

Each mode of operation is unique in

- The I/O device assignments for the system input and output files
- The specification of GCOS communication by way of control cards in batch, or a command language for time sharing
- The default compiler options for the compilation process.

Part of the user interface procedure between the programmer and the FORTRAN compiler results in transmitting compilation error messages and run-time diagnostics to a specified I/O device. These messages enable the programmer to locate the line in the source program at which the error occurred; the form of the message defines the type of error that resulted.

BATCH MODE

The system I/O devices for the local batch mode are the card reader, card punch, and line printer. The user communicates directly with GCOS for system services via the GCOS control cards and the usable slave mode instructions. Because the execution of programs submitted via the local batch mode is carried out directly under GCOS, the program exists under GCOS as a separate batch job. Input processing is performed by System Input and allocation by the GCOS allocator.

The remote batch mode is equivalent to the local batch mode in capability. However, the system I/O device is assigned to the remote computer as remote files rather than to the local card reader and local printer/punch.

Batch Call Card

The system call card for FORTRAN in batch mode is:

```
1      8       16
$      FORTY   Options
          or
$      FORTRAN Options
```

Operand Field:

The operand field specifies one or more of the following system options which are available with batch FORTRAN (default options are underlined):

LSTIN	- A listing of source input is prepared by the FORTRAN compiler.
NLSTIN	- No listing of the source input is prepared.
LSTOU	- A listing of the compiled object program output is prepared.
NLSTOU	- No listing of the compiled object program output is prepared.
DECK	- A binary object program deck is prepared as output.
NDECK	- No binary object program deck is prepared.
COMDK	- A compressed source deck is prepared as output.
NCOMDK	- No compressed source deck is prepared as output.
MAP	- A storage map of the program labels, variables, and constants is prepared as output. (Error message 233 is printed for all unreferenced variables.)
NOMAP	- No storage map is prepared.
XREF	- A cross-reference report is prepared as output. A TO-FROM transfer table is generated. The GMAP offset is printed on the LSTIN report.
NXREF	- No cross-reference report is prepared.
DEBUG	- A run time debug symbol table (.SYMT.) is included in the object program.
NDEBUG	- No debug symbol table is prepared.
BCD	- The execution time character set is BCD (see Appendix A).
ASCII	- The execution time character set is ASCII (see Appendix A). Refer to ASCII/BCD Considerations in Section IV for a description of the JCL to obtain BCD output.
FORM	- The source program is in standard statement format.
NFORM	- The source program is "free form".

LNO	- The source input records are line numbered beginning in Column 1 and terminating with the first nonnumeric character. This option is only operable with the NFORM option (assumed option for NFORM).
NLNO	- The source records are not line numbered (assumed option for FORM).
NJREST	- This job is not restarted following system interruption.
JREST	- This job is restarted following system interruption.
NREST	- This job is not restarted with current activity following system interruption.
REST	- This activity is restarted following system interruption.
OPTZ	- A global optimization procedure is performed, so that the object program produced is highly efficient. It should be noted that this option slows the compilation rate, though not significantly.
NOPTZ	- Global optimization of the object program is not performed.
DUMP	- Slave memory dump is given if the compilation activity terminates abnormally.
NDUMP	- Program registers, upper SSA, and slave program prefix is dumped if the compilation activity terminates abnormally.
NWARN	- No compilation warning messages are printed.
FDS	- Enables the FORTRAN Debugging System (FDS) (Refer to Appendix F for an explanation of the FDS).
DML	- Invokes the Data Manipulation Language (DML) facility of I-D-S/II (ASCII is the only default option when the DML option is specified). Refer to the Data Management-IV (FORTRAN) Reference Manual.
DDLST	- Generates a listing of the subschema source text when I-D-S/II is used (ASCII and DML are the default options when the DDLST option is specified).

NOTES: 1. Independent of the DUMP/NDUMP option, FORTRAN has the capability of producing a symbolic dump of the internal tables in the event of a compiler abort. The presence of a $ SYSOUT *F control card activates this process.

2. To run a FORTRAN job on a DPS ASCII-only system, the ASCII option must be explicitly specified on the $ FORTY control card.

Sample Batch Deck Setup

The following deck setup illustrates the required control cards for the compilation and execution of a batch FORTRAN activity. The control cards are fully described in the Control Cards Reference Manual.

```
1      8       16
$      SNUMB   ......
$      IDENT   FORTRAN
$      OPTION  FORTRAN
$      FORTY   Options
          or
$      FORTRAN Options
         .
         .          FORTRAN Source Deck(s)
         .
$      EXECUTE Options
$      FILE Card(s)
$      FFILE Card(s)
$      ENDJOB
```

Sample Batch Link/Overlay JCL

```
1      8        16
$      OPTION   FORTRAN,NOGO
$      SELECT   main-object-permfile
$      LINK     link1
$      SELECT   suba-object-permfile
$      LINK     link2,link1
$      SELECT   subb-object-permfile
$      PRMFL    H*,W,R,hstar-permfile
$      EXECUTE
$      ENDJOB

       the main program contains:

            .
            .
            .
       CALL LLINK("link1")
       CALL SUBA
            .
            .
            .
       CALL LLINK("link2")
       CALL SUBB
            .
            .
            .
```

REMOTE BATCH INTERFACE

Refer to the Network Processing Supervisor (NPS) and Remote Terminal Supervisor (GRTS) manuals for descriptions of deck setups required for submitting a batch job from a remote computer.

FILE SYSTEM INTERFACE

The file system provides multiprocessor access to a common data base. The file system allocates permanent file space and controls file access for users in local and remote batch and time sharing. The file system is fully described in the File Management Supervisor manual.

TERMINAL/BATCH INTERFACE

The JRN time sharing subsystem allows a batch job to be submitted from a time sharing terminal. This capability is provided in Appendix B, and is fully described in the TSS Terminal/Batch Interface Facility reference manual.

ASCII/BCD CONSIDERATIONS

FORTRAN enables the programmer to choose the character set that is most convenient for the normal mode of execution, or best meets the needs of the application.

Specification of BCD or ASCII is possible in both batch and time sharing. In batch, the $ FORTY in the $ FORTRAN card provides BCD by default; in time sharing, the RUN command provides ASCII by default. The selection is made at compile time and need not normally be designated for execute-only runs.

When BCD is selected,

- Internal character data and formats are carried in BCD
- Storage is allocated at a rate of six characters per word
- Library calls are made to the entry names that work with BCD for I/O, ENCODE, PAUSE, etc.

When ASCII is selected,

- Internal character data and formats are carried in ASCII
- Storage is allocated at a rate of four characters per word
- Library calls are made to the entry names that work with ASCII for I/O, ENCODE, PAUSE, etc.

Therefore, one generally cannot mix object modules of different character sets because conflicts arise over which routines are to be loaded from the library, how to index through character arrays, how to analyze FORMAT statements, etc.

BCD or ASCII programs execute in either batch or time sharing with certain automatic convenience functions for dealing with the variety of file and device types accessible to the program. In terms of specific problems, automatic file transliteration and/or reformatting on a logical record basis is provided for the following:

1. Execution of an ASCII program.
    a. Input and output can be directed to the reader, printer, punch, or SYSOUT.
    b. Input files can be BCD (media code 0, 2, or 3) or ASCII (media code 6).
2. Execution of a BCD program. Input files can be ASCII (media code 6).
3. Execution of an ASCII program under time sharing. Input files can be ASCII (media code 6) or BCD (media code 0, 2, or 3).
4. Binary input/output files (media code 1) can be read and/or written with either character option.

Use of the word "can" in the lists above implies an optional capability. This is based on the existence of a collection of alternate entry names in the File and Record Control called from FORTRAN library modules. Specification for this optional capability in batch is under the programmer's control. The proper linkage is accomplished when the following control card is presented to the General Loader:

```
$  USE  .GTLIT
```

Files not requiring transliteration and/or reformatting are acceptable as input. Output files are recorded in the media code relative to the internal character set of the executing program independent of the environment. BCD programs output files with media codes 0, 2, and 3; ASCII programs output files with media codes 6 and 7.

FILE FORMATS

All output files generated by FORTRAN, whether formatted or unformatted, ASCII or BCD, sequential or random[1] are in standard system format (as described in the File and Record Control reference manual).

Files generated in time sharing in the build-mode or by Text Editor can be used directly as ASCII input data files for a FORTRAN object program. BCD file output can be listed (using the SCAN subsystem) at either the user's terminal or at a high speed online printer (BATCH verb of SCAN).

GLOBAL OPTIMIZATION

Global optimization gives the user some control over the balance between compilation and object program efficiency. This analysis has been collected into a single optional compiler phase that is elected by the OPTZ option on the language processor control card or the RUN command. The analyses performed include:

1. Common Subexpression Analysis - This analysis provides a determination of multiple occurrences of the same subexpression within a program block. The goal is to perform a given computation only one time.

2. Expression Compute Point Analysis - This analysis provides a determination of the optimal place and time for the computation of some expression in relation to the loop structure of the program and the redefinition points of the expression's constituent elements.

3. Induction Variable Expression Analysis - This analysis determines the optimal computation sequence. Its intent is to reduce expressions to simple operations upon an index register at the loop boundaries.

4. Loop Collapsing Analysis - This analysis attempts to reduce two or more nested loops into a single loop.

5. Register Management Analysis - This analysis determines how registers and temporary storage are to be allocated. Priorities are assigned according to the number of references to an expression and the loop level of these references. Candidates for global assignment over one or more program loops are selected.

6. Induction Variable Materialization Analysis - This analysis determines the necessity for materializing in memory the current value of a DO index.

Memory Conflicts

FORTRAN utilizes the memory designated as open in the slave prefix by calling a common routine to manage memory (.GCORE). Therefore, conflicts will arise when another system program (e.g., SORT/MERGE) attempts to use the same area without calling the common memory management routine. Conflicts with SORT/MERGE can be avoided by dividing the free area of memory. (Refer to the SORT/MERGE manual).

[1]Random files can optionally be treated as nonstandard format. The file format consists of fixed length records without record control words and block control words. See Section V, "Unformatted Random File Input/Output Statements".

The use of global optimization does not always result in a faster running program; furthermore, there are situations where the object code generated by global optimization is not an exact functional equivalent of no-global-optimization generated code using the same source.

Example

If a program contains multiple references to invariant expressions, code for the evaluation of that expression follows the program prologue. This placement could result in the unnecessary evaluation of the expressions if references were from statements conditionally executed (i.e., the conditions can be such that the expressions are not to be referenced).

```
      COMMON A,B,C, L1,L2,L3
      .......
      IF(L1) 1,2,1
 1    Z=A+B
      Y=A+B
 2    IF(L2) 3,4,3
 3    Z2=(B+C)
      .......
      Z3=(B+C)
      .......
 4    IF(L3) 5,6,5
 5    Y1=(A+C) + (A+C)**2
      .......
      Y2=(A+C)
 6    CONTINUE
```

Expressions (A+B), (B+C) and (A+C) have multiple references under conditional code, and are pre-calculated following the prolog. However, if L1, L2, and L3 were all zero, this evaluation will have been done unnecessarily.

Another example demonstrates how results can actually be different (OPTZ vs NOPTZ). Consider the following example where the programmer is attempting to avoid a divide check fault (i.e., division by zero).

```
      FUNCTION FX(A,B)
      .......
 10   IF(B) 1,2,1
  1   FX=A/B+(A/B)**2+(A/B)**3
      GO TO 3
  2   FX=A+A**2+A**3
  3   CONTINUE
      .......
      END
```

Divide Check

The OPTZ generation may produce a divide check even though a test is made for zero division. If B=0, this is the case in the previous example when (A/B) is evaluated prior to the zero test for B.

This situation can be avoided in either of two ways.

a. The previous example could be rewritten as:

```
      FUNCTION FX(A,B)
      .......
   10 IF(B.NE.0.)FX=A/B+(A/B)**2+(A/B)**3
      IF(B.EQ.0)FX=A+A**2+A**3
      CONTINUE
      .......
      END
```

The optimization phase is "sensitive" to logical IF statements. Expressions that are only referenced within the truth clause of a logical IF statement are not removed from such a conditional setting.

b. The following modification to the original example eliminates the side effect.

```
      FUNCTION FX(A,B)
      .......
   10 IF(B) 1,2,1
    1 Z=A/B
      FX=Z+Z**2+Z**3
      GO TO 3
    2 FX=A+A**2+A**3
    3 CONTINUE
      .......
      END
```

Another situation results from using certain outdated library "flag" routines. For example, if a program uses FLGEOF or FLGERR to set an end-of-file or error flag, expressions involving these flag variables may appear to the optimizer as invariant over some range of statements when there actually may be a redefinition due to input/output.

Example

```
      INTEGER UNT
      CALL FLGEOF(UNT,IF)
      DO 100 I=1,N
      READ(UNT)V1,V2
      IF(IF.EQ.0)READ(UNT)V3,V4
      IF(IF.EQ.0)READ(UNT)V5,V6
  100 CONTINUE
```

Since the optimizer does not consider each of the READ statements as a potential redefinition point for the variable IF, the expression (IF.EQ.0) is removed from the DO 100 I=1,N loop. Thus, in this case, the EOF is never sensed; however, the use of the END= clause avoids this problem.

Example

```
      DO 100 I=1,N
      READ(UNT,END=10)V1,V2
      .......
      READ(UNT,END=10)V3,V4
      .......
100   READ(UNT,END=10)V5,V6
      .......
 10   PRINT,"END OF FILE ON",UNT
```

In summary, global optimization does not guarantee the generation of faster running programs, and in some instances undesirable faults can be introduced. However, analysis of this optimization technique has shown that, in general, significant improvement in the object code usually results.

BATCH COMPILATION LISTINGS AND REPORTS

The following compilation listings and reports produced by the system are controlled by options on the $ FORTY or $ FORTRAN control card (default options are underlined).

Option	Listing or Report Produced
LSTIN	Source Program Listing
LSTOU	Source and Object Program Listing with a Program Preface Summary
XREF	Cross Reference Report, TO-FROM Transfer Table, and GMAP offset on LSTIN report
MAP	Storage Map and Program Preface Summary
DEBUG	Debug Symbol Table

The following report codes are used for batch compilation:

Report Code	Compilation
74	Print on execution report which includes: ● source program listing ● diagnostic report if NLSTIN option is present ● reports produced by LSTOU,XREF, MAP, and DEBUG options ● compilation statistics report
75	Punch compressed deck (COMDK option)
76	Punch object deck (DECK option)
77	Print alter input list

Any diagnostics pertinent to the program are included in the LSTIN report if it is not suppressed. When the NLSTIN option is present, the diagnostics appear as a free-standing report.

The Compilation Statistics Report is produced if any other report is produced or the DECK or COMDK options are utilized.

Figure 4-1 contains an example of a program with all reports. The following descriptions explain each report in more detail, using Figure 4-1 as a base for the description.

Source Program Listing (LSTIN)

Each line of this report, (page 1 of Figure 4-1), is divided into three fields. The leftmost field contains the line or alter number for each source line. If the source program is line-numbered (NFORM and LNO options specified), the actual line number is displayed in this field. If the source program is not line-numbered (FORM or NFORM and NLNO options specified), this field contains the alter number (relative sequence number of the line).

The second field contains the text of the source statement and is separated from the first field by six blank characters.

The third field is separated from the second by six blank characters and contains optional sequence/identification information (columns 73-80) from the source line.

Diagnostics are recorded immediately following the source line to which they apply. Diagnostics that do not apply to a particular source line appear at the end of the source listing. Comment cards may appear between the source line and the appropriate diagnostic.

Each diagnostic line begins with five asterisks followed by the character W to indicate a warning, F for a fatal error, or T for a premature termination of the compilation (refer to Appendix C for a description of the diagnostics generated by the compiler).

In Figure 4-1, a warning diagnostic appears after line 5; the correct object code is generated.

If the XREF option is on, this report then contains four fields with the GMAP offset printed as the leftmost column of the report. The line or alter number is then printed as the second field, followed by the text as the third field, and the optional sequence information as the fourth field. This gives the relative location in the object code of each executable source statement.

To-From Transfer Table (XREFS)

The To-From Table (page 2 of Figure 4-1), lists the transfers that exist in the source program logic. The report is sorted into descending line number sequence, keying on the originating line number, and displays up to five transfers on one report line. The destination line number field may indicate the word EXIT or RETURN if the transfer statement is a STOP or RETURN statement. For assigned GO TO statements, where the label list is not provided, the label variable name is displayed. Line 29 contains the transfer statement GO TO 7, which is indicated as the first entry in the transfer report (NOTE: statement 7 begins on line 10); line 28 contains the transfer statement STOP, which is the second entry in the report; etc.

If the line numbers of the source file are not sequentially increased by one, the actual line number is that of the first executable statement whose line number is less than or equal to the line number printed.

Program Preface Summary (LSTOU)

The Program Preface Summary (page 3 of Figure 4-1), documents the object module preface (card) information in a format similar to that printed by GMAP. The source program memory requirements and blank common size are displayed in octal and decimal followed by the number of the V count bits as used in the instructions with special (type 3) relocation.

The SYMDEFs entry denotes, in octal, the relative offset of the internal location corresponding to that symbol definition. This entry is followed by a list of labeled common blocks which are referenced by this module. Associated with each symbol are three octal fields and one decimal field. The first field gives the global symbol number associated with the common name for this compilation. This is the number that appears in the V field of any instruction referencing this labeled common region. The number is justified according to the V field. Thus, if labeled common SPACE is global symbol 2, and the V field is five bits wide, the display is 020000 (bit zero is the sign bit). If the V field is six bits wide, the display is 010000. The second field contains the size, in octal, of the labeled common region. The third decimal field contains the same size in decimal.

Two labeled common regions, .DATA. and .SYMT., receive special treatment by the loader. Although they are not actually labeled common names, they are included in this portion of the Program Preface Summary. .DATA. is allocated enough space to contain all local data required by the program. This includes arrays and scalars not appearing in common as arguments, constants, encoded FORMAT information, NAMELIST lists, temporary storage for intermediate results, argument pointers, the error linkage pair (E.L..), etc. .SYMT. is generated when the DEBUG option is used. This block contains a symbol table for all program variables and statement numbers and can be used for symbolic debugging.

A list of external symbol references (SYMREFs) is also included with their associated global symbol number, justified as described above, for labeled common names.

Storage Map (MAP)

The Storage Map (page 4 of Figure 4-1), provides information on the allocation of storage for identifiable program elements, and generates any error messages (#233) for all the variables that are defined but never referenced in the program unit. This report is divided into three parts: variables and arrays, statement numbers, and constants.

The first part of the report which lists all program variables and arrays in alphabetical order contains four fields:

1. The first field contains the global symbol name relative to which variable is defined. Local variables and arrays are defined relative to the origin of the .DATA. space. When a variable or array belongs to some labeled common block, the name of its common is shown; when it belongs to blank common, the field is empty. Argument variables and arrays appear as variables of .DATA., and the indicated location is reserved for a pointer to the actual argument and is initialized on entry to the procedure.

2. The two OFFSET fields provide the location of the variable or array relative to the assigned global name. For arrays, this is the starting location; subsequent elements of the array are allocated the higher order locations. The offset is provided in both octal and decimal for the convenience of the programmer.

3. The MODE field provides the type associated with each identifier. Switch variables are indicated by an empty field.

The second part of the report lists all referenced statement numbers in numerical order. The four fields to the right of each entry are the same as defined above. The ORIGIN fields for FORMAT statement numbers are always .DATA. and the MODE field indicates FORMAT. For executable statement numbers, the MODE field is always blank. The ORIGIN field is eight dots (........) if this is a main program, or the first SYMDEF if this is a subprogram. The OFFSET field is the same as described above.

The third part of this report lists all numeric and character constants requiring unique storage. All constants are allocated storage relative to the .DATA. block. The two OFFSET fields and the MODE field are as described for variables and arrays. Only the first 17 characters are displayed for character constants.

Object Program Listing (LSTOU)

The Object Program Listing (pages 5-8 of Figure 4-1), gives a full listing of the generated object program. The original source statement is identified in the object listing by "SOURCE LINE xxx" and the source line. The individual instruction line format is similar to that produced by GMAP. The first field is the location field followed by the compiled machine language instruction, which is usually divided into address, operation code, and modifier fields. The location field and machine language instruction field are in octal. The next three digits are the relocation bits applicable to the instruction.

The symbolic equivalent of the generated instruction is contained in the next field. This instruction consists of a label field, an operation code field, and a variable field for address and modifier symbols. Referenced statement numbers appear in the label field prefixed by the characters ".S". SYMDEF symbols (such as ENTRY names) also appear in the label field. Operation code and modifier mnemonics are the same as the standard GMAP mnemonics with the exception of some of the pseudo-operation codes.

Data initialization, constants, formats, symbol table entries, etc. are displayed at the end of the report following the source END line. No object END instruction is produced.

Debug Symbol Table (DEBUG)

A table of all symbols used in the source program is given on page 9 of Figure 4-1.

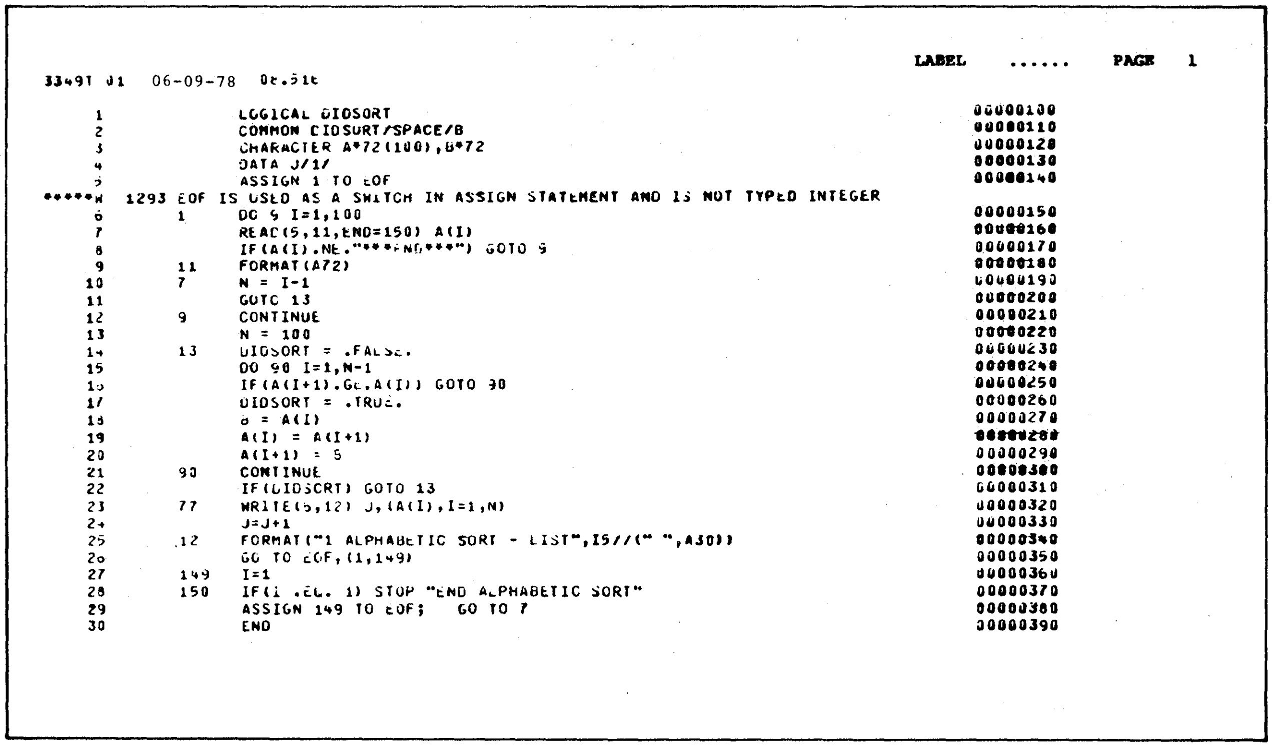
Cross-Reference List (XREF)

The Cross-Reference List (page 10 of Figure 4-1), lists in alphabetical order all referenced variables, arrays, statement numbers, SYMREFs and SYMDEFs. Each element results in four or more entries being produced across the line. The first field is the octal location (offset) of the item relative to its global symbol. The second field is the item name or symbol. Statement numbers are shown with a prefix of ".S". The third field is the applicable global symbol. The fourth field is the line number (alter number) of the first reference. When there are more references, additional line numbers are displayed across the line, and where required, additional lines are written.

The second part of the report lists the statement labels; the first part of the report contains all other information required for cross referencing.

Miscellaneous Data

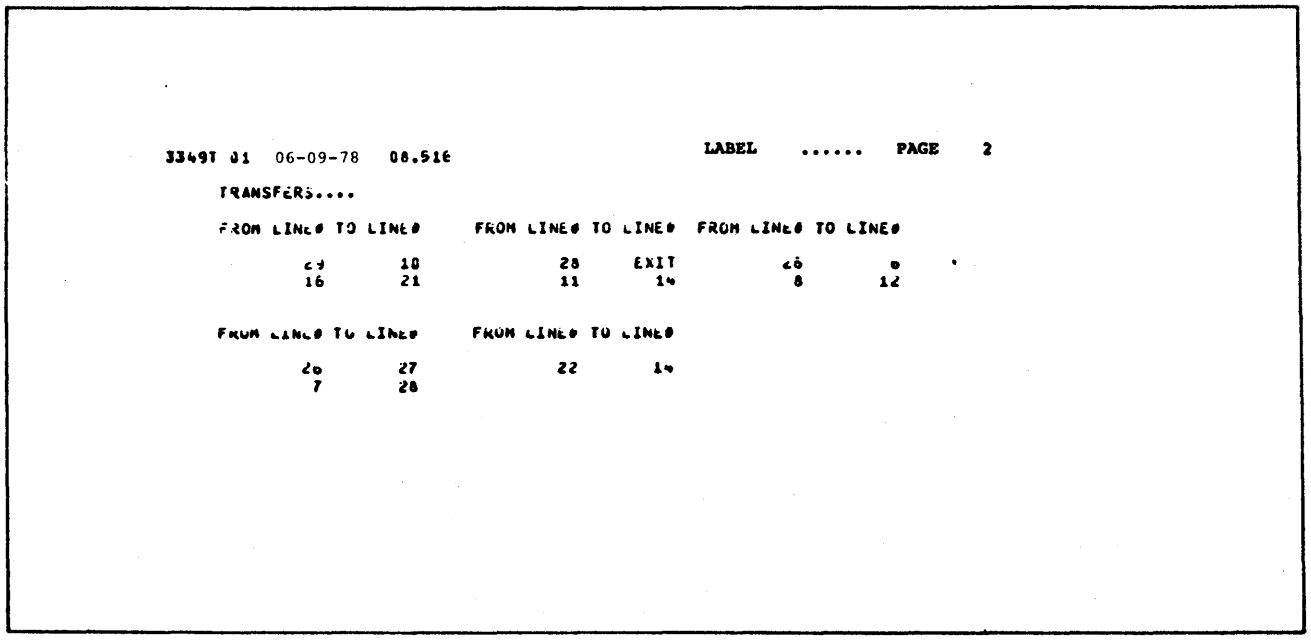
Additional compilation data is printed at the end of the report listing. This data consists of the edit date, the software release level of the compiler, the processor time and compilation speed in terms of source lines per minute, the number of diagnostics printed, and the amount of memory space required for the compilation.



```
                                                                      LABEL     ......    PAGE   1
33491 01   06-09-78  06.51E

    1              LOGICAL DIDSORT                                          00000100
    2              COMMON DIDSORT/SPACE/B                                   00000110
    3              CHARACTER A*72(100),B*72                                 00000120
    4              DATA J/1/                                                00000130
    5              ASSIGN 1 TO EOF                                          00000140
*****W  1293 EOF IS USED AS A SWITCH IN ASSIGN STATEMENT AND IS NOT TYPED INTEGER
    6      1       DO 9 I=1,100                                             00000150
    7              READ(5,11,END=150) A(I)                                  00000160
    8              IF(A(I).NE."***END***") GOTO 9                           00000170
    9      11      FORMAT(A72)                                              00000180
   10      7       N = I-1                                                  00000190
   11              GOTO 13                                                  00000200
   12      9       CONTINUE                                                 00000210
   13              N = 100                                                  00000220
   14      13      DIDSORT = .FALSE.                                        00000230
   15              DO 90 I=1,N-1                                            00000240
   16              IF(A(I+1).GE.A(I)) GOTO 90                               00000250
   17              DIDSORT = .TRUE.                                         00000260
   18              B = A(I)                                                 00000270
   19              A(I) = A(I+1)                                            00000280
   20              A(I+1) = B                                               00000290
   21      90      CONTINUE                                                 00000300
   22              IF(DIDSORT) GOTO 13                                      00000310
   23      77      WRITE(6,12) J,(A(I),I=1,N)                               00000320
   24              J=J+1                                                    00000330
   25      12      FORMAT("1 ALPHABETIC SORT - LIST",I5//(" ",A30))         00000340
   26              GO TO EOF,(1,149)                                        00000350
   27      149     I=1                                                      00000360
   28      150     IF(I .EQ. 1) STOP "END ALPHABETIC SORT"                  00000370
   29              ASSIGN 149 TO EOF;   GO TO 7                             00000380
   30              END                                                      00000390
```

Source Program Listing

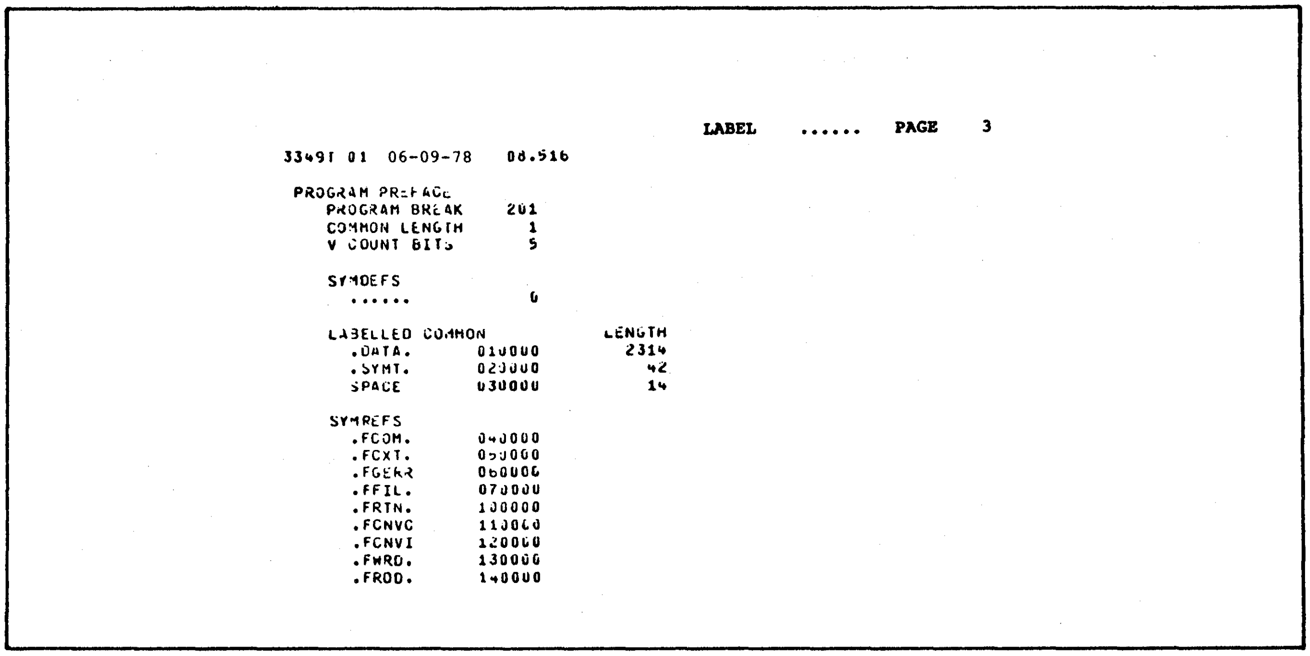
Figure 4-1. Compilation Listings and Reports



```
33491 01   06-09-78   08.516                         LABEL      ......    PAGE     2

     TRANSFERS....

     FROM LINE# TO LINE#       FROM LINE# TO LINE#  FROM LINE# TO LINE#

             29        10                28     EXIT           26         6
             16        21                11       14            8        12

     FROM LINE# TO LINE#       FROM LINE# TO LINE#

             26        27                22       14
              7        28
```

To-From Transfer Table

Figure 4-1 (cont). Compilation Listings and Reports



```
                                                     LABEL     ......  PAGE    3
33491 01  06-09-78    08.516

 PROGRAM PREFACE
    PROGRAM BREAK     201
    COMMON LENGTH       1
    V COUNT BITS        5

    SYMDEFS
      ......            0

    LABELLED COMMON             LENGTH
      .DATA.       010000         2314
      .SYMT.       020000           42
      SPACE        030000           14

    SYMREFS
      .FCOM.       040000
      .FCXT.       050000
      .FGERR       060000
      .FFIL.       070000
      .FRTN.       100000
      .FCNVC       110000
      .FCNVI       120000
      .FWRD.       130000
      .FROD.       140000
```

Program Prefix Summary

Figure 4-1 (cont). Compilation Listings and Reports

```
                                           LABEL     ......   PAGE    4
33497 31  06-09-78    08.516

    STORAGE MAP

    SYMBOLIC  ORIGIN    OFFSET(10)   MODE        OFFSET(8)

    .E.L..    .DATA.       1201       DOUBLE      2261
    A         .DATA.          0       CHARACTER      0
    B         SPACE           0       CHARACTER      0
    DIOSORT                   0       LOGICAL        0
    EOF       .DATA.       1204                   2264
    I         .DATA.       1205       INTEGER     2265
    J         .DATA.       1203       INTEGER     2263
    N         .DATA.       1213       INTEGER     2275

      STATEMENT NUMBERS

          1  .........        2                      2
          7  .........       32                     40
          9  .........       36                     44
         11  .DATA.        1208       FORMAT      2270
         12  .DATA.        1217       FORMAT      2301
         13  .........       42                     52
         90  .........       80                    120
        149  .........      116                    164
        150  .........      118                    166

      CONSTANTS (.DATA.)

       5                   1207       INTEGER     2267
      ***END***            1210       CHARACTER   2272
                           1212       CHARACTER   2274
       6                   1216       INTEGER     2300
      END ALPHABETIC SO    1224       CHARACTER   2310
```

Storage Map

Figure 4-1 (cont). Compilation Listings and Reports

```
                                                                                        LABEL    ......   PAGE    5
33431 11  06-09-78   08.516

                         000000    ........ NULL
                SOURCE LINE      1                      LOGICAL DIDSORT
                SOURCE LINE      2                      COMMON DIDSORT/SPACE/8
                SOURCE LINE      3                      CHARACTER A*72(100),B*72
                SOURCE LINE      4                      DATA J/1/
                SOURCE LINE      5                      ASSIGN 1 TO EOF
 000000  000002 6200 00  010           EAX0  .S1
 000001  012264 7400 00  030           STX0  EOF
                SOURCE LINE      6                1     DO 9 I=1,100
                         000002    .S1         NULL
 000002  000001 2360 07  000           LDQ   1,DL
 000003  012265 7560 00  030           STQ   I
 000004  000014 4020 07  000           MPY   12,DL
                SOURCE LINE      7                      READ(5,11,END=150) A(I)
 000005  012266 7560 00  030           STQ   .DATA.+1206
 000006  140000 7010 00  030           TSX1  .FRDD.
 000007  000015 7100 00  010           TRA   *+6
 000010  012261 000007   030           ZERO  .E.L..,7
 000011  012267 0000 00  030           ARG   .DATA.+1207
 000012  012270 0000 00  030           ARG   .S11
 000013  000000 0000 00  000           ARG   0
 000014  000166 7100 00  010           TRA   .S150
 000015  012266 7220 00  030           LXL2  .DATA.+1206
 000016  410014 6350 12  030           EAA   A-12,2
 000017  110000 7010 00  030           TSX1  .FCNVC
 000020  000110 0110 07  000           NOP   72,DL
 000021  110000 7010 00  030           TSX1  .FRTN.
                SOURCE LINE      8                      IF(A(I).NE."***END***")GOTO 9
 000022  012265 2360 00  030           LDQ   I
 000023  000014 4020 07  000           MPY   12,DL
 000024  000000 6220 06  000           EAX2  0,QL
 000025  012272 6270 00  030           EAX7  .DATA.+1210
 000026  410014 6210 12  030           EAX1  A-12,2
 000027  005640 5602 01  000           RPD   2,1,TNZ
 000030  000000 2350 17  000           LDA   0,7
 000031  000000 1150 11  000           CMPA  0,1
 000032  000004 6010 04  000           TNZ   4,IC
 000033  012274 2350 00  030           LDA   .DATA.+1212
 000034  024240 5202 01  000           RPT   10,1,TNZ
 000035  000000 1150 11  000           CMPA  0,1
 000036  000040 6000 00  010           TZE   *+2
 000037  000044 7100 00  010           TRA   .S9
                SOURCE LINE      9                11    FORMAT(A72)
                SOURCE LINE      10               7     N = I-1
                         000040    .S7         NULL
 000040  012265 2360 00  030           LDQ   I
 000041  000001 1760 07  000           SBQ   1,DL
 000042  012275 7560 00  030           STQ   N
                SOURCE LINE      11                     GOTO 15
 000043  000052 7100 00  010           TRA   .S15
                SOURCE LINE      12               9     CONTINUE
                         000044    .S9         NULL
```

Object Program Listing

Figure 4-1 (cont). Compilation Listings and Reports

```
33491 01  06-09-78   06.516                                                          LABEL     ......     PAGE    6

      000044  012265 2360 00  030              LDQ   I
      000045  000001 0760 07  000              ADQ   1,DL
      000046  000145 1160 07  000              CMPQ  101,DL
      000047  000003 6040 00  010              TMI   *-36
                  SOURCE LINE      13                            N = 100
      000050  000144 2360 07  000              LDQ   100,DL
      000051  012275 7560 00  030              STQ   N
                  SOURCE LINE      14                    13    DIDSORT = .FALSE.
                        000052    .S13         NULL
      000052  000000 2360 07  000              LDQ   0,DL
      000053  000000 7560 00  020              STQ   DIDSORT
                  SOURCE LINE      15                            DO 90 I=1,N-1
      000054  000014 2220 03  000              LDX2  12,DU
      000055  012275 2360 00  030              LDQ   N
      000056  000001 1760 07  000              SBQ   1,DL
      000057  012276 7560 00  030              STQ   .DATA.+1214
      000060  000000 5330 00  000              NEGL  0
      000061  000000 0760 07  000              ADQ   0,DL
      000062  000002 6040 04  000              TMI   2,IC
      000063  000001 3360 07  000              LCQ   1,DL
      000064  012277 7560 00  030              STQ   .DATA.+1215
                  SOURCE LINE      16                            IF(A(I+1).GE.A(I)) GOTO 90
      000065  010000 6270 12  030              EAX7  A,2
      000066  410014 6210 12  030              EAX1  A-12,2
      000067  031640 5602 01  000              RPD   12,1,TNZ
      000070  000000 2350 17  000              LDA   0,7
      000071  000000 1150 11  000              CMPA  0,1
      000072  000074 6020 00  010              TNC   *+2
      000073  000120 7100 00  010              TRA   .S90
                  SOURCE LINE      17                            DIDSORT = .TRUE.
      000074  000001 2360 07  000              LDQ   1,DL
      000075  000000 7560 00  020              STQ   DIDSORT
                  SOURCE LINE      18                            B = A(I)
      000076  410014 6270 12  030              EAX7  A-12,2
      000077  030000 6210 00  030              EAX1  B
      000100  000000 0110 07  000              NOP   0,DL
      000101  031600 5602 01  000              RPD   12,1
      000102  000000 2350 17  000              LDA   0,7
      000103  000000 7550 11  000              STA   0,1
                  SOURCE LINE      19                            A(I) = A(I+1)
      000104  010000 6270 12  030              EAX7  A,2
      000105  410014 6210 12  030              EAX1  A-12,2
      000106  000000 0110 07  000              NOP   0,DL
      000107  031600 5602 01  000              RPD   12,1
      000110  000000 2350 17  000              LDA   0,7
      000111  000000 7550 11  000              STA   0,1
                  SOURCE LINE      20                            A(I+1) = B
      000112  030000 6270 00  030              EAX7  B
      000113  010000 6210 12  030              EAX1  A,2
      000114  000000 0110 07  000              NOP   0,DL
      000115  031600 5602 01  000              RPD   12,1
      000116  000000 2350 17  000              LDA   0,7
```

Object Program Listing (cont)

Figure 4-1 (cont). Compilation Listings and Reports

```
3349T 01   06-09-78   06.516                                                          LABEL   ......   PAGE   7

      000117  000000 7550 11  000           STA   0,1
                 SOURCE LINE      21                 90     CONTINUE
                        000120    .S90      NULL
      000120  000014 0220 03  000           ADLX2 12,DU
      000121  012277 0540 00  030           AOS   .DATA.+1215
      000122  000065 6010 00  010           TNZ   *-29
                 SOURCE LINE      22                        IF(DIOSORT) GOTO 13
      000123  000000 2340 00  020           SZN   DIOSORT
      000124  000352 6010 00  010           TNZ   .S13
                 SOURCE LINE      23                 77     WRITE(6,12) J,(A(I),I=1,N)
      000125  130000 7010 00  030           TSX1  .FWRD.
      000126  000132 7100 00  010           TRA   *+4
      000127  012261 000027   030           ZERO  .E.L..,23
      000130  012300 0000 00  030           ARG   .DATA.+1216
      000131  012301 0000 00  030           ARG   .S12
      000132  012263 2350 00  030           LDA   J
      000133  120000 7010 00  030           TSX1  .FCNVI
      000134  000014 2220 03  000           LDX2  12,DU
      000135  012275 2360 00  030           LDQ   N
      000136  000002 6040 04  000           TMI   2,IC
      000137  000001 3360 07  000           LCQ   1,DL
      000140  012277 7560 00  030           STQ   .DATA.+1215
      000141  410014 6350 12  030           EAA   A-12,2
      000142  110000 7010 00  030           TSX1  .FCNVC
      000143  000110 0110 07  000           NOP   72,DL
      000144  000014 0220 03  000           ADLX2 12,DU
      000145  012277 0540 00  030           AOS   .DATA.+1215
      000146  000141 6010 00  010           TNZ   *-5
      000147  070000 7010 00  030           TSX1  .FFIL.
                 SOURCE LINE      24                        J=J+1
      000150  012263 0540 00  030           AOS   J
                 SOURCE LINE      25                 12     FORMAT("1 ALPHABETIC SORT - LIST";I5//(" ";A30))
                 SOURCE LINE      26                        GO TO EOF,(1,149)
      000151  000006 6210 04  000           EAX1  6,IC
      000152  012264 6350 51  030           EAA   EOF,I
      000153  004300 5202 01  000           RPT   2,1,TZE
      000154  000000 1150 11  000           CMPA  0,1
      000155  777777 6000 31  000           TZE   -1,1*
      000156  000003 7100 04  000           TRA   3,IC
      000157  000002 0000 00  010           ARG   .S1
      000160  000164 0000 00  010           ARG   .S149
      000161  060000 7010 00  030           TSX1  .FGERR
      000162  000164 7100 00  010           TRA   *+2
      000163  012261 000032   030           ZERO  .E.L..,26
                 SOURCE LINE      27                 149    I=1
                        000164    .S149     NULL
      000164  000001 2360 07  000           LDQ   1,DL
      000165  012265 7560 00  030           STQ   I
                 SOURCE LINE      28                 150    IF(I .EQ. 1) STOP "END ALPHABETIC SORT"
                        000166    .S150     NULL
      000166  000001 2360 07  000           LDQ   1,DL
      000167  012265 1160 00  030           CMPQ  I
```

Object Program Listing (cont)

Figure 4-1 (cont). Compilation Listings and Reports

```
                                                                                   LABEL     ......    PAGE    8
33497 01  06-09-78   06.516

      000170  000176 6010 00  010           TNZ   *+6
      000171  050000 7010 00  030           TSX1  .FCXT.
      000172  000176 7100 00  010           TRA   *+4
      000173  012261 000034   030           ZERO  .E.L..,28
      000174  012310 0000 00  030           ARG   .DATA.+1224
      000175  000023 0000 07  000           ARG   19,DL
                  SOURCE LINE      29                    ASSIGN 149 TO EOF;   GO TO 7
      000176  000164 6200 00  010           EAX0  .S149
      000177  012264 7400 00  030           STX0  EOF
      000200  000040 7100 00  010           TRA   .S7
                  SOURCE LINE      30                    END



                         002261             ORG   .DATA.+1201
      002261  000000000000    000 .E.L..    OCT
      002262  333333333333    000           ETC
      002263  000000000001    000 J         DEC    1

                         002267             ORG   .DATA.+1207
      002267  000000000005    000           DEC    5
      002270  352107025520    000 .S11      BCI   (A72)

                         002272             ORG   .DATA.+1210
      002272  545454254524    000           BCI   ***END
      002273  545454202020    000           BCI   ***
      002274  202020202020    000           BCI

                         002300             ORG   .DATA.+1216
      002300  000000000006    000           DEC    6
      002301  357601202143    000 .S12      BCI   ("1 AL
      002302  473021222563    000           BCI   PHABET
      002303  312320624651    000           BCI   IC SOR
      002304  632052204331    000           BCI   T - LI
      002305  626376733105    000           BCI   ST",I5
      002306  616135762076    000           BCI   //(" "
      002307  732103005555    000           BCI   ,A30))
      002310  254524202143    000           BCI   END AL
      002311  473021222563    000           BCI   PHABET
      002312  312320624651    000           BCI   IC SOR
      002313  632020202020    000           BCI   T
```

Object Program Listing (cont)

Figure 4-1 (cont). Compilation Listings and Reports

```
                                             LABEL     ......    PAGE    9
 3349T 31  06-09-78   06.516

          DEBUG SYMBOL TABLE (.SYMT.)

     000000  332533433333   000          VTABF .E.L..,DOUBLE
     000001  012261000023   030
     000002  243124624651   000          VTABF DIDSORT,LOGICAL
     000003  000000300025   020
     000004  222020202020   000          VTABF G,CHARACTER
     000005  030000000020   030
     000006  212020202020   000          VTABF A,CHARACTER
     000007  010000000020   030
     000010  412020202020   000          VTABF J,INTEGER
     000011  012263000021   030
     000012  012020202020   000          LTABF .S1
     000013  000002000077   010
     000014  254626202020   000          VTABF EOF,CHARACTER
     000015  012264000020   030
     000016  312020202020   000          VTABF I,INTEGER
     000017  012265000021   030
     000020  112020202020   000          LTABF .S9
     000021  000044000077   010
     000022  010120202020   000          LTABF .S11
     000023  012270000077   030
     000024  010500202020   000          LTABF .S150
     000025  000166000077   010
     000026  072020202020   000          LTABF .S7
     000027  000040000077   010
     000030  452020202020   000          VTABF N,INTEGER
     000031  012275000021   030
     000032  010320202020   000          LTABF .S13
     000033  000052000077   010
     000034  110020202020   000          LTABF .S90
     000035  000120000077   010
     000036  010220202020   000          LTABF .S12
     000037  012301000077   030
     000040  010411202020   000          LTABF .S149
     000041  000164000077   010
```

Debug Symbol Table

Figure 4-1 (cont). Compilation Listings and Reports

```
                                                                              LABEL ......  PAGE    10
33497 01  06-09-78   08.516

ORIGIN SYMBOLIC   REFERENCES BY ALTER NUMBER

     0 ........  ........          3
    11 .FCNVG                      7      23
    12 .FCNVI                      23
     4 .FCOM.
     5 .FCXT.                      28
     7 .FFIL.                      23
     6 .FGERR                      26
    14 .FRDC.                      7
    10 .FRIN.                      7
    13 .FWRD.                      23
  2251 .E....    .DATA.            6      7      23     26     28
     0 A         .DATA.            7      8      16     18     19     20     23
     3 B         SPACE             18     20
     0 DIDSORT                     14     17     22
  2254 EOF       .DATA.            5      26     29
  2265 I         .DATA.            6      8      16     12     27     28
  2263 J         .DATA.            4      23     24
  2275 N         .DATA.            10     13     15     23

     0 .S0        FORMAT
     2 .S1                         5      6      26
    43 .S7                         10     29
    44 .S9                         8      12
  2273 .S11       FORMAT           7
  2311 .S12       FORMAT           23
    72 .S13                        11     14     22
     0 .S77
   120 .S90                        16     21
   164 .S145                       26     27     29
   166 .S150                       7      22
```

Cross Reference List

Figure 4-1 (cont). Compilation Listings and Reports

```
   EDIT DATE    06-09-78        **14.1 **

ELAPSED TIME   (SEC)      1.05          LINES/MINUTE     1704

  THERE WERE     1 DIAGNOSTICS  IN ABOVE COMPILATION
    30K WORDS WERE USED FOR THIS COMPILATION
```

Miscellaneous Data

Figure 4-1 (cont). Compilation Listings and Reports

SECTION V

INPUT AND OUTPUT

GENERAL DESCRIPTION

FORTRAN input/output (I/O) statements cause the transmission of information between internal storage and external input/output devices. Each I/O statement can specify an implicit (NAMELIST) or explicit list of scalars, arrays, and array elements; output statements can also specify constants and expressions of all types. The designated data items are assigned values on input and, on output, have their values transferred to the specified output device. The I/O statements used in FORTRAN (READ and DECODE for input; WRITE, PRINT, PUNCH, and ENCODE for output) are briefly described in Section III. This section contains a more detailed description of the following elements which make up the input/output statements.

- File reference (file code)
- FORMAT
- NAMELIST name reference
- Internal storage buffer reference for ENCODE and DECODE
- Optional transfer condition
- Input/output list specification

File reference can consist of an integer constant, an integer variable, or an integer expression that identifies the input/output unit. The value of the integer will be a two-digit file code, which must be in the range 01 $\leq$ file $\leq$ 63. A file is associated with a specific device by using the $ FILE and $ FFILE control cards or by using the 'fe' file descriptors of the RUN command described in Appendix B.

FORMAT reference can be an integer constant representing the statement label of a FORMAT statement, a character scalar, or an array name. If a statement label is represented, the identified FORMAT statement must appear in the same program unit as the input/output statement. If a character variable name is referenced, the variable must contain FORMAT information (see "Variable Format Specifications" in this section).

NAMELIST input/output is indicated by the presence of a NAMELIST name in the format reference position of the READ, WRITE, and PRINT statements. The NAMELIST statement(s) and its associated list must appear before any input/output statements that reference the NAMELIST name.

Internal storage buffer applies only to the ENCODE and DECODE statements. While it is desirable to use character variables, variable names of any type can be used.

Optional transfer conditions (end-of-file and error) are designated as END= and ERR=, respectively. END= can appear in sequential or random file input statements; ERR= can appear in any input/output statement. A statement label or switch variable name can follow the equal sign (=), and the order of the transfer conditions is not important. Conditions that can cause an error return include transmission errors or any of the error conditions described in the File and Record Control manual.

I/O list specification information that is transmitted is collected into records that can be formatted or unformatted. A formatted record consists of a string of permissible characters in the character set. The transfer of such a record requires that FORMAT information be referenced or implied, to supply the necessary positioning and conversion specifications. The number of records transferred by the execution of a formatted I/O statement is determined by the list and the referenced FORMAT statement. A formatted record can be analogous to a print line or a card image, whereas, an unformatted record consists of a string of words.

There are two kinds of formatted input/output: format directed and list directed. List directed formatted input/output can be specified by a FORMAT statement of the form FORMAT(V) or it can be implied by the form and content of the input/output statement.

Input/output Statements

Formatted Read/Write Statements - These statements include a FORMAT reference, the file reference, possibly an end-of-file option, an error return option, and a list specification. List directed I/O is accomplished via the FORMAT (V). Namelist I/O is accomplished with a NAMELIST name as a format reference.

Unformatted Read/Write Statements - These statements refer to binary word oriented sequential and random files.

Manipulation Input/output Statements - These statements are for file operations relating to positioning and file demarcation, and can be used to operate on sequential access files only.

FORMAT and NAMELIST Statements - These two nonexecutable statements are used with the formatted input/output statements.

The FORMAT statement specifies the arrangement of data in the input/output record. If the FORMAT statement is referred to by a READ statement, the input data must meet the specifications described later in this section.

The NAMELIST statement specifies an input/output list of variables and/or arrays. Input/output of the values associated with the list is affected by reference to the NAMELIST name in a READ, PRINT, or WRITE statement. If the NAMELIST name is referred to by a READ statement, the input data must meet the specifications described later in this section.

FILE REFERENCE

In the source program, files can be designated by any integer expression, the value of which must be in the range of $1 \leq \text{file} \leq 63$. In batch mode, the equating of a numeric file designation with some actual device is accomplished via standard GCOS file allocation control cards using a two-digit file code of the same integer value as the corresponding file designator. Thus, WRITE (06,100) references file code 06 at run time.

Since the file reference can be any integer expression, the following statements also reference file code 06.

```
I = 5
WRITE (I+1, 100)
```

Five specific file designators are predefined for all FORTRAN programs and serve as the default assignments in a batch environment:

05 - standard input file (I*)

```
READ (05,f) list
READ (05,x)
```

06 - standard output file (P*)

```
WRITE (06,f) list
WRITE (06,x)
```

This output appears in report code 06 of the execution report.

41 - standard input file (I*)

```
READ, list
READ f, list
READ x
READ(41,f) list
READ (41,x)
```

42 - standard print output file (P*)

```
PRINT, list
PRINT f, list
PRINT x
WRITE (42,f) list
WRITE (42,x)
```

This output appears in report code 52_8 of the execution report.

43 - standard punch output file (P*)

```
PUNCH x
PUNCH, list
PUNCH f, list
WRITE(43,f) list
WRITE(43,x)
```

This output is directed to the card punch. Its report code is 53_8.

f = FORMAT REFERENCE
x = namelist name

NOTE: These file designators can be overridden by the programmer.

FORMAT SPECIFICATIONS

The FORMAT statement in FORTRAN specifies the physical description of the input/output data items. This description can be designated in several different forms, as described in the following paragraphs.

Field Separators

The field separator, which is used to separate the field descriptors of a FORMAT statement, may be a slash, a comma, or a series of slashes. When the slash is used to separate field descriptors it specifies a demarcation of formatted records.

Repeat Specification

A field descriptor can be repeated by placing the repetition number before the field descriptor with the exception of quoted strings, tabulation controls, nH, and nX.

Example

```
FORMAT (3E12.4)
```

is the repeat specification for

```
FORMAT (E12.4, E12.4, E12.4)
```

A group of field descriptors can be repeated by enclosing the group in parentheses and placing the repetition number before the parentheses. This enables two levels of grouping to be permitted, using the same rules for representation.

Example

```
FORMAT (2(F10.6, E10.2)
FORMAT (2(I3,2(F8.4, E8.2)), A10)
```

are the repeat specifications for

```
FORMAT (F10.6, E10.2, F10.6, E10.2)
FORMAT (I3, F8.4, E8.2, F8.4, E8.2, I3, F8.4, E8.2, F8.4, E8.2, A10)
```

Scale Factors

To permit more general use of D-, E-, F-, and G-descriptors a signed integer constant scale factor followed by the letter P can precede the specification. The magnitude of the scale factor must be between -8 and +8, inclusive. The scale factor is defined for input as follows:

$$10^{-(\text{scale factor})} \times \text{external quantity} = \text{internal quantity}$$

For an F-type output, the scale factor is defined as follows:

$$\text{external quantity} = \text{internal quantity} \times 10^{+(\text{scale factor})}$$

For D- and E-type output conversion, the mantissa part of the output is multiplied by 10**(scale factor) and the exponent is reduced by the scale factor. A scale factor of 1P causes a nonzero numeric to print to the left of the decimal point, thus providing an extra digit of useful numeric output data with no net increase in field width as compared to a scale factor of zero.

For G output conversion, if the range of the value is such that the effective use is an F-conversion, the effect of the scale factor is suspended. If the effective use of E-conversion is required, the effect is the same as for E-output.

If input data is in the form xx.xxxx and it is intended for use internally in the form .xxxxxx, then the FORMAT specification to affect this change is 2PF7.4. For output data, scale factors can be used with D-, E-, F-, and G-conversion.

Example

The statement FORMAT (I2,3F11.3) might output the following printed line:

```
27ƀƀƀƀ-93.209ƀƀƀƀƀ-0.008ƀƀƀƀƀƀ0.554
```

But the statement FORMAT (I2,1P3F11.3) used with the same data would output the following line:

```
27ƀƀƀ-932.094ƀƀƀƀƀ-0.076ƀƀƀƀƀƀ5.536
```

whereas, the statement FORMAT (I2,-1P3F11.3) would output the following line:

```
27ƀƀƀƀƀ-9.321ƀƀƀƀƀ-0.001ƀƀƀƀƀƀ0.055
```

A scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it holds for all D-, E-, F-, and G-conversions following the scale factor within the same FORMAT statement. This applies to both single-record formats, multiple-record formats, and to repeated portions of formats . Once the scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. For F-type conversion, the output of numbers with an absolute value greater than or equal to 2^{35} after scaling, is output in E-conversion. Scale factors have no effect on I- and O-conversion.

Multiple Record Formats

When the list of an input or output statement is used to transmit more than one record with different formats, a slash (/) is used to separate the format specifications for the different lines. For example, if two records are to be read with a single READ statement and the first has a five-digit integer and the second has five real numbers, the FORMAT statement could be:

```
FORMAT (I5/5E10.3)
```

It is also possible to specify a special format for the first (one or more) records and a different format for subsequent records. This is done by enclosing the last record specifications in parentheses. For example, if the first card in a deck has an integer and a real number and all the following cards contain two integers and a real number, the FORMAT statement might be:

```
FORMAT (I6,E10.3/(2I6,E12.3))
```

If a multiple-line format is desired in which the first two lines are to be printed according to a special format, and all remaining lines according to another format, the last line specification should be enclosed in a second pair of parentheses.

Example

```
FORMAT (I2,3E12.4/2F10.3,3F9.4/(10F12.4))
```

If data items remain to be output after the format specification has been completely "used", the format repeats from the last previous left parenthesis that is at level 0 or 1. The various levels of parentheses are illustrated below. The parentheses labeled 0 are zero level parentheses; those labeled 1 are first level parentheses; and those labeled 2 are second level parentheses.

Example

```
FORMAT (3E10.3,(I2,2(F12.4,F10.3)),D28.17)
       0       1   2             21       0
```

If more items in the list are to be transmitted after the format statement has been completely used, the FORMAT repeats from the last first-level left parenthesis (i.e., the parenthesis preceding I2).

NOTE: In the examples above, both the slash and the final right parenthesis of the FORMAT statement are used to indicate the termination of a record.

Slashes have the following affect in a format statement:

Location of Slashes in Format	Blank Lines Printed	Input Records Skipped
Beginning	n	n
Middle	n-1	n-1
End	n-1	n
(Format has slashes only)	n	n

Carriage Control

The WRITE (file, form), PRINT, and PRINT form, statements prepare fields in edited format for the printer. The first character of each record is examined to see if it is a control character to regulate the spacing of the printer. If the first character is recognized as a control character, it is replaced by a blank in the printed line and the line printed after the proper spacing has been affected. This control is usually obtained by beginning a FORMAT specification with 1Hb̸ followed by the desired control character.

Output Device Control

In the absence of a NOSLEW option on a $ FFILE control card (batch mode only), the spacing of the printing on the output device is controlled by the first character of the line of output. The first character of the print line is examined to determine if it is a control character to regulate the spacing of the output device. If the first character is recognized as a control character, the line is printed after the proper spacing has been affected. The control character is blank when the line is printed. This control affects printers, terminals, and displays. When FORMAT (V) is used, either explicitly or implicitly, a blank character is inserted to advance the printer to the next line.

The control characters produce the following effects:

First Character	Effect
0	Causes one blank line to be inserted to provide double spacing.
+	Causes an overprint. In batch, no advance to the next line occurs. In time sharing, a carriage return is obtained but no line feed occurs.
1	Causes a slew to the top of the next page before printing (batch mode only).
&	Suppresses carriage return and line feed. No fill characters are inserted (time sharing mode only).
Any other	Causes single line spacing.

NOTE: If a single question mark character or single exclamation point character is encountered in any position on the print line, these characters will be interpreted as special printer control characters (refer to the File and Record Control manual for additional information).

Input Data

When data to be input to the object program is under format control, the following specifications are required:

- The data must correspond in order, type, and field designation with the field specifications in the FORMAT statements.
- The data field can be shortened by using commas for delimiters (i.e., the input record can contain 1,ɃɃɃɃ2ɃɃ3, for the format specification 3I6; the input values will be 1, 2, 3).
- If a negative number is to be indicated, the minus sign must be used; the plus sign is optional for a positive number.
- Blanks in a numeric field are interpreted as zero.
- Numbers for E- and F- conversion can designate any number of digits; however, only the high-order eight digits of precision are retained, and the number is rounded to eight digits of accuracy.
- Numbers for D- conversion can designate any number of digits; however, only the high-order 18 digits are retained, and the number is rounded to 18 digits of accuracy.
- Numeric data must be right-justified in the field.

The following procedures are permitted in the preparation of input data:

- Numbers for D- and E-conversion do not need to have four columns allocated to the exponent field. The beginning of the exponent field may be marked by a D or an E; if that is omitted, by a plus or minus sign (but not a blank). For example, E2, E+2, +2, +02, and D+02 are all permissible exponent fields.
- Numbers for D-, E-, and F-conversion do not need to contain a decimal point; the format specification is sufficient. For example, the number -09321+1 with the specification E12.4 is treated as though the decimal point had been placed between the 0 and the 9. If the decimal point is included in the field, its position overrides the position indicated in the format specification.

Numeric Field Descriptors

Six field descriptors are available for numeric data:

Internal	Conversion Code	External
Floating-point (double precision)	D	Real with D exponent
Floating-point	E	Real with E exponent
Floating-point	F	Real without exponent
Floating-point	G	Appropriate type
Integer	I	Decimal Integer
Integer or Floating-point	O	Octal Integer

These numeric field descriptors are specified in the forms

PrDw.d, PrEw.d, PrFw.d, PrGw.d, rIw, rOw, where: D, E, F, G, I, and O represent the type of conversion.

w is an unsigned integer constant representing the field width for converted data; this field width can be greater than required to provide spacing between numbers.

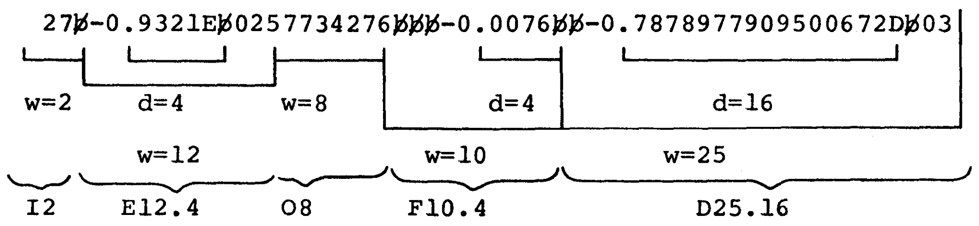
d is an unsigned integer or zero representing the number of digits of the field that appear to the right of the decimal point. For double precision numbers d is limited to 18 and for real numbers, d is limited to 8; d is right-justified in the field for both double precision and real numbers.

P is optional and represents a scale factor designator.

r is the repeat specification; it is an optional nonzero integer constant indicating the number of occurrences of the numeric field descriptor that follows.

Example

The statement FORMAT (I2,E12.4,O8,F10.4,D25.16) might cause the following line to be printed



```
 27ƀ-0.9321Eƀ0257734276ƀƀƀ-0.0076ƀƀ-0.7878977909500672Dƀ03
```

where: ƀ indicates a blank space.

D-, E-, F-, G-, I-, and O-format conversions must follow these rules:

1. No format specification should be designated if it provides for more characters (including blanks) than the number permitted for a particular input/output record, or the capabilities of the relevant device.

2. Information transmitted with the

   - O- conversion must have real or integer names
   - G- conversion must have real, double precision, or complex names
   - E- conversion must have real, double precision, or complex names
   - F- conversion must have real, double precision, or complex names
   - I- conversion must have integer names
   - D- conversion must have real, double precision, or complex names

3. The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real datum.

   Input processing is the same as the F-conversion with the exception of scale processing.

   The method of representation in the external output string is a function of the magnitude of the real datum being converted. If N is the magnitude of the internal datum the following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	Equivalent Output Conversion Effected
$0.1 \leq N \leq 1$	F(w-4).d,4X
$1 \leq N \leq 10$	F(w-4).(d-1),4X
. .	.
. .	.
. .	.
$10^{d-2} \leq N < 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leq N < 10^{d}$	F(w-4).0,4X
Otherwise,	nPEw.d

   NOTE: The effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of the F-conversion.

4. The field width w, for D-, E-, F-, and G-conversions, must include a space for a decimal point and a space for the sign. The D-, E-, and G-conversions also require space for the exponent. For example, for D- and E- and G-conversions on output, $w \geq d+6$, and for F-conversion, $w \geq d+2$.

5. The exponent, which can be used with D- and E-conversions, is the power of 10 to which the number must be raised to obtain its true value. The exponent is written with an E (for E-conversion) or D (for D-conversion) followed by a minus sign if the exponent is negative or a plus sign or a blank if the exponent is positive, and then followed by two numbers that are the exponent.

   Example

   .002 is equivalent to the number .2E-02.

6. For D-conversion input, up to 19 decimal digits are converted and the result is stored in a double word. For D-conversion output, the two storage words representing the double precision quantity are considered one piece of data and converted as such.

7. If a number to be output requires more spaces than are allowed by the field width w, the field is filled with asterisks, unless subroutine NASTRK is invoked (refer to Table 6-4). If the number requires fewer than w spaces, the leftmost spaces are filled with blanks.

   If the field width is exceeded solely because the presence of a nonfunctional leading zero to the left of the decimal point, that zero will be suppressed and the number will be printed. (For a negative number, the minus sign will occupy the former position of the suppressed zero.)

8. The output field is filled with blanks if the output number is $+377777777777_8$ (noise word), unless octal conversion is used.

9. Specifications for successive fields are separated by commas and/or slashes (refer to "Multiple Record Formats" in this section).

Complex Number Fields

Since a complex quantity consists of two separate and independent real numbers, a complex number is transmitted either by two successive real number specifications or by one real number specification that is repeated (e.g., 2E10.2 = E10.2,E10.2). The first specification supplies the real part; the second specification supplies the imaginary part. The following FORMAT statement transmits an array of six complex numbers.

Example

```
FORMAT (2E10.2, E8.3, E9.4, E10.2, F8.4, 3(E10.2, F8.2))
```

Alphanumeric Field Descriptors

Alphanumeric information can be transmitted in two ways that result in the storing of BCD or ASCII characters (as determined by the compilation option).

1. The specifications rAw and rRw cause character data to be read into or written from a variable.

2. Alphanumeric information (i.e., character constants) is introduced into a FORMAT statement by specifying nH, enclosing the string in quotation marks, or enclosing the string in apostrophes.

INPUT

If w is equal to or greater than s, the rightmost s characters are taken from the input field. The I/O pointer is advanced in accordance with the field width of the format specifier. If w is less than s, then w characters are taken from the input field. With A conversion, the data appears left-justified with s-w trailing blanks in the internal representation. For R conversion, the internal representation is right-justified with s-w leading zeros.

where: w is the field width from A or R specification
s is the size specification of a character variable as specified in the CHARACTER statement.

OUTPUT

If w is greater than s, then s characters are transmitted to the output field preceded by w-s blanks for R conversion, or followed by w-s blanks for A conversion. If w is less than or equal to s, the output field consists of w characters from the internal representation. With A conversion the w leftmost characters are transmitted; with R conversion, the w rightmost characters are transmitted.

where: w is the field width for A or R specification
s is the size specification of a character variable as specified in the CHARACTER statement

The R code is equivalent to the A code; however, the characters are right-justified with leading alphanumeric zeros in the internal representation of the R code on input.

When the variable associated with an A or R format is not specified as type CHARACTER, the variable is treated as a character variable with a size of one word of storage (i.e., 6 characters for BCD; 4 for ASCII).

Logical Field Descriptor

Logical variables can be read or written using the specification Lw, where L represents the logical type of conversion and w is an integer constant that represents the data field width.

On input, a value representing either true or false is stored if the first nonblank character in the field of w characters is a T or an F, respectively. If all the w characters are blank, a value representing false is stored. On output, a value of .TRUE. or .FALSE. in storage causes w-1 blanks to be written followed by a T or an F, respectively.

Character Positioning Field Descriptors

The X and T field descriptors enable a specified number of characters in the record to be skipped. On output, the X descriptor causes a specified number of spaces to be inserted in the external output record.

X FORMAT CODE

The field descriptor for space characters is nX. On input, n characters of the external input record are skipped. On output, n space characters are inserted in the external output record. If n = 0, a value of one is assumed.

T FORMAT CODE

The field descriptor for tabulation is Tt where t is the position in a FORTRAN record where the transfer of data is to begin. The t is an unsigned integer constant, which specifies that tabbing can proceed backward as well as forward. This format code permits input or output to begin at any specified position.

Variable Format Specifications

Any of the formatted input/output statements (including ENCODE and DECODE) can contain a character scalar or an array name in place of the reference to a format statement label. At the time a variable is referenced in such a manner, the first part of the information must be character data that constitutes a valid format specification, (e.g., (I4)). There is no requirement on the information following the right parenthesis that ends the format specification.

The format specification (the value of the variable referenced) must have the same form as that defined for a FORMAT statement, without the word FORMAT. Thus the character text of the specification begins with a left parenthesis and ends with a matching right parenthesis.

The format specification can be defined by a data initialization statement, by a READ statement with an A format, by use of a character replacement statement, or by ENCODE.

In the following example, A, B, and part of the array C are converted and stored according to the FORMAT specifications read into the array FMT at execution time.

```
  DIMENSION FMT (12), C(10)
1 FORMAT (12A6)
  READ (5,1) FMT
  READ (5,FMT) A,B, (C(I), I=1,5)
```

A similar example follows, using a character scalar for the variable format.

```
  DIMENSION C(10)
  CHARACTER FMT*72
1 FORMAT (A72)
  READ (5,1)FMT
  READ (5,FMT) A,B,(C(I),I=1,5)
```

NAMELIST INPUT/OUTPUT STATEMENTS

NAMELIST input/output is indicated by the presence of a NAMELIST name in the format reference position of the READ, WRITE, and PRINT statements. The NAMELIST statement and its associated list must appear before any input/output statement referencing the NAMELIST name.

Input

When a READ statement refers to a NAMELIST name, the designated input device is made ready and input of data is begun. The first input data record is searched for a "$" immediately followed by the NAMELIST name, which is followed by a comma or one or more blank characters. If the search fails, additional records are examined consecutively until there is a successful match or an end-of-file. When a successful match is made of the NAMELIST name on a data record and the NAMELIST name referred to in a READ statement, data items are converted and placed in storage.

Format 1

```
READ (file,name-1,opt1,opt2)
```

Format 2

```
READ name-2
```

where: file is the file reference

name-1 and name-2 are namelist names

opt1 is the error condition transfer

opt2 is the end-of-file condition transfer

NOTE: Format 2 issues a read request to the standard system input device

Any combination of the four types of data items, which are described below, can be used in a data record. The data items must be separated by commas, and empty fields (=,), (b̸,), or (,,) cause an invalid word to be stored. If more than one physical record is needed for input data, the last item of each record must be followed by a comma. The end of a group of data is signaled by a $ following the last item either in the same data record as the NAMELIST name or anywhere in any succeeding records. The $ can replace the comma following the last data item. Data is restricted to columns 1 through 72 in card image (media code 2). The $ that indicates the end of a logical record of input data cannot appear in column 1 since GCOS input processing will retain it as a pseudo control card, and delete it from the input data file.

Data items can take the form

- Variable name = constant

  ```
  CON = 17.5
  X(6) = 26.4
  ```

  where the variable name can be an array element name or a simple variable name with a maximum of six characters; subscripts must be integer constants.

- Array name = set of constants (separated by commas)

  ```
  X = 1.,2.,3.,5*6.3
  ```

  where k* constant can be included to represent k constants (k must be an unsigned integer). The number of constants must not exceed the number of elements in the array.

- Subscripted variable = set of constants (separated by commas)

  ```
  Y(4) = 9.,6.,10*1.8
  ```

  where k* constant can be included to represent k constants (k must be an unsigned integer). A data item of this form results in the set of constants being placed in array elements, starting with the element designated by the subscripted variable.

  The number of constants given cannot exceed the number of elements in the array that are included between the given element and the last element in the array, inclusive.

- Variable 1/Variable 2 = constant(s)

  where Variable 1 is a counter that is set after the data has been input, indicating the number of constants that have been stored for Variable 2.

Constants used in the data items can be

- Integers
- Real numbers
- Double precision numbers
- Complex numbers
- Logical constants
- Character data where the character string does not exceed the space available on the card; this cannot be used with a repeat count.

Logical or complex constants should be associated only with logical or complex variables, respectively; character data can be associated with any type of variable. The other types of constants can be associated with integer, real, or double precision variables and are converted in accordance with the type of variable. With the exception of the character data, blanks must not be embedded in a constant or repeat count field, but they can be used freely elsewhere within a data record.

Character data may be delimited by a blank if it does not contain any embedded blanks. It can also always be delimited by nH, apostrophes, or quotation marks.

Any selected set of variable or array names belonging to the NAMELIST name that is referred to by the READ statement, can be used as specified in the preceding description of data items. Names that are made equivalent to these names cannot be used unless they also belong to the NAMELIST name.

In the following examples, the arrays A, I, and L, and the variables B and J, belong to the NAMELIST name, NAM1; the array A, and the variables C, J, and K, belong to the NAMELIST name, NAM2.

```
DIMENSION A(10),I(5,5),L(10)
NAMELIST /NAM1/A,B,I,J,L/NAM2/A,C,J,K
```

```
                     123456______________________________

First Data Card      $NAM1   I(2,3)=5,J=4.2,B=4,

Second Data Card     A(3)=7,6.4,L=2,3,8*4.3$
```

NOTE: The $ sign in the first data card is not in column one.

If this input data is used with the NAMELIST statement illustrated above and with a READ statement, the following actions take place.

- The input file designated in the READ statement is prepared and the next record is read.
- The record is scanned for a $ immediately followed by the NAMELIST name, NAM1.
- Because the search is successful, data items are converted and placed in storage.
- The integer constant 5 is placed in I(2,3).
- The real constant 4.2 is converted to an integer and placed in J.
- The integer constant 4 is converted to real and placed in B.
- Since no data items remain in the record, the next input record is read.
- The integer constant 7 is converted to real and placed in A(3).
- The real constant 6.4 is placed in the next consecutive location of the array, A(4).
- Since L is an array name not followed by a subscript, L(1) through L(10) are filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively.

- The real constant 4.3 is converted to an integer and placed in L(3) through L(10).
- The $ signals termination of the input for the READ operation.

Output

When data is output via NAMELIST (e.g., WRITE(6,NAM1)), all variables in the NAMELIST statement will be output and the output values are labeled with an appropriate variable name.

Format 1

```
WRITE (file,name-1,opt1,opt2)
```

Format 2

$\left\{\begin{matrix}\underline{\text{PRINT}}\\ \underline{\text{PUNCH}}\end{matrix}\right\}$ name-2

where: file is the file reference

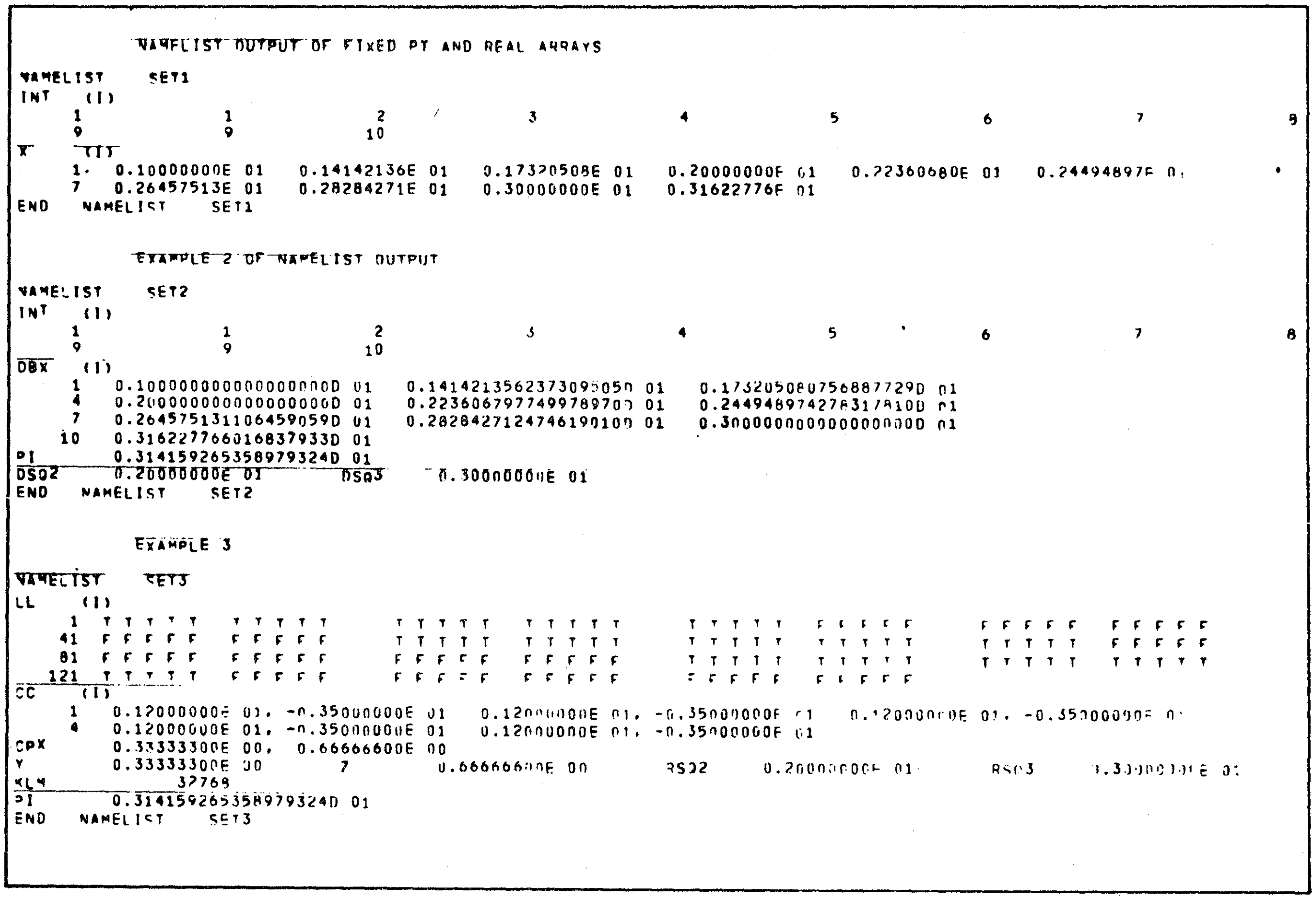
name-1 and name-2 are the NAMELIST names

opt1 is the error condition transfer

opt2 is the end-of-file condition transfer

NOTE: Format 2 directs output to the standard system print/punch output device.

The format of the output can appear with or without comma separators. Output directed to file 43 includes commas and, therefore, is in agreement with the NAMELIST input format. Output can be directed to file 43 by either the PUNCH statement or a WRITE statement referencing file 43. Output directed to a file other than 43 does not include comma separators and, therefore, cannot be processed by NAMELIST input. Figures 5-1 and 5-2 contain a sample program and sample output from that program in the latter format.



```
          NAMELIST OUTPUT OF FIXED PT AND REAL ARRAYS

NAMELIST    SET1
INT    (I)
     1            1            2            3            4            5            6            7            8
     9            9           10
X     (I)
     1   0.10000000E 01   0.14142136E 01   0.17320508E 01   0.20000000E 01   0.22360680E 01   0.24494897E 0
     7   0.26457513E 01   0.28284271E 01   0.30000000E 01   0.31622776E 01
END    NAMELIST    SET1

          EXAMPLE 2 OF NAMELIST OUTPUT

NAMELIST    SET2
INT    (I)
     1            1            2            3            4            5            6            7            8
     9            9           10
DBX    (I)
     1   0.10000000000000000000D 01   0.14142135623730950D 01   0.17320508075688772 9D 01
     4   0.20000000000000000000D 01   0.22360679774997897 0D 01   0.24494897427831781 0D 01
     7   0.26457513110645905 9D 01   0.28284271247461901 0D 01   0.30000000000000000000D 01
    10   0.31622776601683793 3D 01
PI       0.31415926535897932 4D 01
DSQ2     0.20000000E 01          DSQ3     0.30000000E 01
END    NAMELIST    SET2

          EXAMPLE 3

NAMELIST    SET3
LL     (I)
     1  T T T T T   T T T T T    T T T T T   T T T T T    T T T T T   F F F F F    F F F F F   F F F F F
    41  F F F F F   F F F F F    T T T T T   T T T T T    T T T T T   T T T T T    T T T T T   F F F F F
    81  F F F F F   F F F F F    F F F F F   F F F F F    T T T T T   T T T T T    T T T T T   T T T T T
   121  T T T T T   F F F F F    F F F F F   F F F F F    F F F F F   F F F F F
CC     (I)
     1   0.12000000E 01, -0.35000000E 01   0.12000000E 01, -0.35000000E 01   0.12000000E 01, -0.35000000E 0
     4   0.12000000E 01, -0.35000000E 01   0.12000000E 01, -0.35000000E 01
CPX      0.33333300E 00,  0.66666600E 00
Y        0.33333300E 00       7        0.66666600E 00       RSQ2     0.20000000E 01       RSQ3     0.33000000E 01
KLM              32768
PI       0.31415926535897932 4D 01
END    NAMELIST    SET3
```

Figure 5-1. Test Program for NAMELIST Output

```
          NAMELIST OUTPUT OF FIXED PT AND REAL ARRAYS

NAMELIST    SET1
INT    (I)
     1             1              2              3              4              5              6              7              8
     9             9             10
X      (I)
     1    0.10000000E 01    0.14142136E 01    0.17320508E 01    0.20000000E 01    0.22360680E 01    0.24494897E 01
     7    0.26457513E 01    0.28284271E 01    0.30000000E 01    0.31622776E 01
END    NAMELIST    SET1


          EXAMPLE 2 OF NAMELIST OUTPUT

NAMELIST    SET2
INT    (I)
     1             1              2              3              4              5              6              7              8
     9             9             10
DBX    (I)
     1    0.10000000000000000D 01    0.14142135623730950D 01    0.17320508075688772D 01
     4    0.20000000000000000D 01    0.22360679774997897D 01    0.24494897427831781D 01
     7    0.26457513110645905D 01    0.28284271247461901D 01    0.30000000000000000D 01
    10    0.31622776601683793D 01
PI        0.31415926535897932D 01
DSQ2      0.20000000E 01          DSQ3      0.30000000E 01
END    NAMELIST    SET2


          EXAMPLE 3

NAMELIST    SET3
LL     (I)
     1  T T T T T   T T T T T     T T T T T   T T T T T     T T T T T   F F F F F     F F F F F   F F F F F
    41  F F F F F   F F F F F     T T T T T   T T T T T     T T T T T   T T T T T     T T T T T   F F F F F
    81  F F F F F   F F F F F     F F F F F   F F F F F     T T T T T   T T T T T     T T T T T   T T T T T
   121  T T T T T   F F F F F     F F F F F   F F F F F     F F F F F   F F F F F
CC     (I)
     1    0.12000000E 01, -0.35000000E 01    0.12000000E 01, -0.35000000E 01    0.12000000E 01, -0.35000000E 01
     4    0.12000000E 01, -0.35000000E 01    0.12000000E 01, -0.35000000E 01
CPX       0.33333300E 00,  0.66666600E 00
Y         0.33333300E 00       7         0.66666600E 00        RSQ2      0.20000000E 01        RSQ3      0.30000000E 01
KLM                 32768
PI        0.31415926535897932D 01
END    NAMELIST    SET3
```

Figure 5-2. NAMELIST Output of Fixed Point And Real Arrays

INTERNAL DATA CONVERSION

The ENCODE and DECODE statements are similar to the formatted WRITE and READ statements, respectively, although the ENCODE/DECODE statements do not cause input/output to take place. They cause data conversion and transmission to take place between an internal buffer area and the elements specified by a LIST. The forms of the ENCODE and DECODE statements are:

```
ENCODE (a,t,opt2)list
DECODE (a,t,opt2)list
```

where: a is the internal buffer

t is the format reference

opt2 is the error condition transfer

list is the input/output specification

NOTE: The internal buffer area "a" is designated by the first operand within the parentheses and can be designated as

- A character scalar
- A character array element
- An array

When the buffer area is designated as a scalar, it is analogous to a print line for ENCODE where the print line is as long as the buffer area in characters. For DECODE, the buffer area is analogous to a card or record image, where the record size is equal to the size of the buffer in characters.

Multiple Record Processing

An analogy can be drawn between character array elements and records. Consider the following example that converts character data to integer type:

```
     CHARACTER TEXT*48(10)
     INTEGER DATA (50)
     DO 100 I=1,50,5
100  DECODE (TEXT(I/5+1),101)(DATA(J),J=I,I+4)
101  FORMAT (5I7)
```

Examination of the format and list reveals that 50 items are to be converted, 5 items per record; hence, 10 records are required. The character array TEXT has 10 elements that are treated as records, each element being 48 characters long. The format requires 35 characters of each element (5 x 7). Thus, the first 35 are processed.

The same result can be accomplished if the list and format specifications cover the full 10 records as follows:

```
          CHARACTER TEXT *48(10)
          INTEGER DATA (50)
          DECODE (TEXT,10) DATA
     10   FORMAT (5I7)
```

In a BCD mode program (six characters per word), the same result can also be accomplished with an internal buffer of type INTEGER as follows:

```
          INTEGER TEXT (8,10), DATA(50)
          DECODE (TEXT,10) DATA
     10   FORMAT (5I7)
```

If the same program is compiled in the ASCII mode, the format specification describes 35 character records, while the array has provisions for only 32 (8*4) characters per "record". This word size/byte size problem is eliminated by the character data type since

```
     CHARACTER  TEXT *48(10)
```

is valid for both modes. In BCD, the equivalent of an 8 x 10 array is allocated; in ASCII, the equivalent of a 12 x 10 array is allocated. The source program is character set independent. For this reason the preferred type of the internal buffer argument of the ENCODE and DECODE statements is CHARACTER. Warning diagnostics are posted when this is not the case, as in the third example.

Editing Strings with ENCODE

With ENCODE, characters not processed are left unchanged.

Example

```
          CHARACTER TEXT*20
          TEXT = "WOW IS THE TIME FOR "
          ENCODE (TEXT,10) "NOW"
     10   FORMAT (A3)
     20   PRINT, TEXT, "ALL GOOD MEN"
          STOP;END
```

The execution of statement 20 causes the following to be printed:

```
     NOW IS THE TIME FOR ALL GOOD MEN
```

If the editing is intended to be used to skip characters, the T format should be used rather than the X format (the X format would cause blanks to be inserted into the string).

Example

```
     10   CHARACTER TEXT*40
     20   TEXT = "NOW IS THE TIME FOR ALL GOOD MEN"
     30   ENCODE(TEXT,10) "PERSONS"
     40   10 FORMAT (T30,A7)
     50   PRINT, TEXT
     60   STOP;END
```

The execution of this program causes the following to be printed:

```
NOW IS THE TIME FOR ALL GOOD PERSONS
```

Conditional Format Selection

A problem common in FORTRAN programs arises when the format of the next record cannot be determined without first reading it. This problem can be overcome through the capability of the DECODE statement. As an example, consider that input to a program is in card form, and the cards come in one of three formats. When card column 1 contains a 0, the first format is to be applied; when it contains a 1 the second; and 2 the third. The following subroutine could be used:

```
          SUBROUTINE READ (A,I,Z)
          CHARACTER CARD*79
          READ 101,KOL1,CARD
    101   FORMAT(I1,A79)
          GO TO (200,300,400),KOL1+1
    200   DECODE (CARD,201) A,I,Z
    201   FORMAT (T11,F12.6,3X,I5,E12.6)
          RETURN
    300   DECODE (CARD,301) A,Z,I
    301   FORMAT (T11,2F12.6,3X,I5)
          RETURN
    400   DECODE (CARD,401)I,A,Z
    401   FORMAT (T51,I5,2E12.6)
          RETURN ; END
```

Another similar problem has to do with the building of format specifications at run time for subsequent use in input processing. As an example, consider that some data file is interspersed with control cards that specify the amount and format of ensuing data. The first field of the control card gives the number of data items that is read; the second gives the number of fields per card (up to 20) or is zero indicating "use the previously developed format"; the remaining fields on the control card come in pairs and provide "w" and "d" sizes for "F" Format specifications needed for correct conversion of each data item; the control card is in free-field format with comma separators. The following subroutine reads and verifies control cards, builds format specifications, and reads a set of data:

```
    SUBROUTINE READ (A,I)
    DIMENSION A(I)
    INTEGER WD(40)
    CHARACTER FORM*80/" "/
    READ,N,J,(WD(L),L=1, MIN0(2*J,40))
    IF (N.GT.I .OR. N.LT. 1) STOP "ITEM COUNT ERROR"
    IF (J.GT.20 .OR. J.LT.0) STOP "FIELD COUNT ERROR"
    IF (J.EQ.0 .AND.FORM.EQ." ")STOP "UNFORMED FORMAT ERROR"
    IF (J),200,
    NCOL = 0
    DO 50 L=1,2*J,2
    IF (WD(L+1).LT. 0 .OR. WD(L+1).GT.8)GO TO 300
    IF (WD(L).LT. WD(L+1)+2) GO TO 300
     50   NCOL =NCOL + WD(L)
          IF (NCOL .GT. 80)STOP "COLUMN COUNT ERROR"
          FORM=" "
          ENCODE(FORM,101) ("F",WD(L),WD(L+1),",",
        &L=1,2*J-2,2),"F",WD(2*J-1),WD(2*J),")"
    101   FORMAT("(",20(A1,I2,".",I2,A1))
200   READ(05,FORM)(A(L),L=1,N)
      RETURN
300   PRINT 301, (L+1)/2, WD(L),WD(L+1)
301   FORMAT ("1 FORMAT SPEC #",I3," IN ERROR.  W=",I5," D=",I5)
      STOP" FIELD DESCRIPTOR ERROR"
      END
```

The above examples also illustrate the use of a number of other FORTRAN language features, most notably:

1. Expressions used:
   a. as DO parameters
   b. in an output list
   c. as the index of a computed GO TO
2. The CHARACTER data type and A format specifiers for long strings
3. Adjustable dimensions
4. The T (tabulation) format specifier
5. Null label fields on an arithmetic IF
6. STOP with display

Note also that the use of CHARACTER scalars of arbitrary size eliminates program dependency on a character set. The above subroutine will run in ASCII or BCD mode, without change.

LIST SPECIFICATIONS

When variables are to be transmitted, an ordered list of the quantities to be transmitted must be included either in the input/output statements or the referenced NAMELIST statements. The order of the input/output list must be the same as the order in which the information exists or is to exist on the input/output medium.

An input/output list is a string of list items separated by commas that can be:

- An expression (output only)
- An implied DO
- An array name
- A scalar
- A constant (output only)
- An array element

and is processed from left to right. (Parenthesized sublists are permitted only with implied DO's; redundant parentheses result in a fatal diagnostic.)

Examples

The following input/output list utilizing nested implied DO's

```
A,B(3),(C(I),D(I,K),I=1,10), ((E(I,J), I=1,10,2),F(J,3),J=1,K)
```

implies that the information in the external input/output medium is arranged as follows:

```
A,B(3),C(1),D(1,K),C(2),D(2,K),....,C(10),D(10,K),
E(1,1),E(3,1),....,E(9,1),F(1,3),
E(1,2),E(3,2),....,E(9,2),F(2,3),E(1,3),...,F(K,3)
```

The result from the execution of an input/output implied DO list is a DO loop, as though each left parenthesis (except expression and subscripting parentheses) were a DO statement, with indexing given immediately before the matching right parenthesis, and the DO range extending up to that indexing information. The order of the input/output list above can be considered equivalent to the following:

```
  A
  B(3)
  DO 5 I=1,10
  C(I)
5 D(I,K)
  DO 9 J=1,K
  DO 8 I=1,10,2
8 E(I,J)
9 F(J,3)
```

Any number of quantities can appear in a single list. If more quantities are in some input record than in the list, only the number of quantities specified in the list are transmitted and the remaining quantities are ignored. Conversely, if a list contains more quantities than are given in one input record, more records are read and/or blanks are supplied, depending on the FORMAT statement. In this case, blanks are supplied until the FORMAT triggers the record advance. Thus, given a list of known length and a well defined FORMAT, it can be accurately predicted how many records will be read, regardless of the record lengths on the file. The following example

```
      CHARACTER A*1 (50)
      READ (5,100) (A(I),I=1,50)
100   FORMAT (50A1)
```

will read only one record. If less than 50 characters are present in that record, the remaining elements of A will be blank filled. By changing the format to 100 FORMAT(A1) the effect will be to read 50 records using the first character of each record to fill the array. It is the right parenthesis that causes the record advance. Alternately, a slash can be used to trigger a record advance (refer to "Multiple Record Formats" in this section).

Short List I/O

By specifying an array name without subscripts in the list of an input/output statement or a NAMELIST, an entire array can be read or written. Only the name of the array is given and the indexing information is omitted.

Example

```
     DIMENSION A(5,5)
         .
         .
     READ,A
```

where: the READ statement shown reads the entire array A; the array is stored in column order in increasing storage locations, with the first subscript varying most rapidly, and the last subscript varying least rapidly.

List Directed Formatted Input/output Statements

The following input/output statements enable a user to transmit a list of quantities without reference to a NAMELIST name or a detailed FORMAT specification. This is implied FORMAT(V) and the type of each variable in the list determines the conversion to be used.

In all cases where a format reference is supplied, the format must be of the form FORMAT (V). The reference can be a FORMAT statement number, a character scalar, or an array name. Table 5-1 gives the implied format conversions that are used for list directed formatted input/output.

```
     READ  t, list
     PUNCH t, list
     PRINT t, list
     READ   , list
     PRINT  , list
     PUNCH  , list
     READ (f,t,opt1, opt2) list
     WRITE (f,t,opt2) list
```

where: t is the statement label of a FORMAT(V) statement, a character scalar, or an array name.

list is the input information

f is the file reference which is also the file code that can be a positive integer constant, an integer variable, or an integer expression of the range $01 \leq f \leq 63$.

opt1 is the statement label or switch variable to be executed when an end-of-file condition is encountered.

opt2 is the statement label or switch variable to be executed when any I/O error is encountered.

Table 5-1. Implies Format Conversion

TYPE OF VARIABLE	INPUT	OUTPUT
Real	E (or F) w.d	0PE 16.8
Integer	Iw	I16
Logical	Lw	L2
Double Precision	D w.d	0PD 26.18
Complex	2Fw.d	0P2E16.8
Character	Am	Am
m = maximum size		

With list directed formatted input, record control is determined solely by the list. If some record is terminated and the list is not satisfied, another record is read. This process continues until the list is satisfied.

The input information must satisfy the following rules:

1. Numeric and character input values are separated by commas or blanks.

2. Blanks following exponent indicators E, D, or G are not interpreted as separators.

3. Quotes (") or apostrophes (') can be used to bracket a character input value that contains embedded blanks or commas. In this case, the quotes are delimiters and should not be followed by a comma unless the intent is to define a null field after the data.

4. A given input value must be fully contained on one input line.

5. Consecutive commas, an empty line, or the appearance of a comma as the last character of a line imply null input fields. Conversion of a null field is a function of the corresponding list item type and is shown in Table 5-2.

Table 5-2. Conversion of a Null Field

TYPE	VALUE
Integer	0
Real	0.0
Double Precision	0.D0
Complex	(0,0)
Logical	F
Character	all blanks

With list directed formatted output, record control is determined by the list and the standard line lengths. With BCD files, the standard line length is 132 characters; with ASCII files, the standard length is 72 characters. A new line/record is started when the next list item to be transmitted will not fit entirely on the current line. For example, if information has been formatted to character position 60 of an ASCII output line and the next item in the output list is an integer (implied I16 format), a new line is started.

Terminal End-of-File

When the input device is a time sharing terminal, an end-of-file condition may be signaled by transmitting a file separator character (e.g., in Teletype Models 33 and 35 control shift, L) as the only character of a line.

Formatted Input/output Statements

The formatted input/output statements apply to character-oriented records. They are intended for use with the standard input/output devices but may also be used with sequential files and can be expressed in any of the following forms:

Format 1

$$\left\{\begin{array}{l}\underline{\text{READ}}\\ \underline{\text{PRINT}}\\ \underline{\text{PUNCH}}\end{array}\right\}\ \text{format, list}$$

Format 2

```
READ (file, format[,opt1, opt2] ) list
```

Format 3

```
WRITE (file, format, opt2) list
```

where: file is the file reference

format is the format reference

opt1 is the end-of-file condition transfer

opt2 is the error condition transfer

list is the input/output specification

The file reference must be an unsigned integer constant with 5 and 41 assigned to the standard system input device; 6, 42, and 43 assigned to the standard system output devices.

Unformatted Input/output Statement

The unformatted input/output statements apply to sequential files and random files. The major difference between the unformatted sequential file and the unformatted random file operation is in the mode of access to the file. To write a file with the random WRITE statement, the file must be accessed as random. Any attempt to apply a random READ/WRITE statement to a file accessed as sequential causes a program to terminate abnormally.

File Properties

Sequential Files	- A sequential file can contain zero, one or more records accessed in a sequential manner.
Random Files	- A random file consists of records, each of which is addressable (i.e., each record can be accessed without repositioning the file). Each record in the random file must be of the same length.
File Updating	- Input-output routines with random files permit replacement of individual records in a file. The execution of all random file WRITE statements is considered a record replacement.
Record Sizes	- Random files have records, all of the same length.

SEQUENTIAL FILES

The unformatted sequential file input/output statements have the following formats:

Format 1

```
READ (file,opt1,opt2) list
```

Format 2

```
WRITE (file,opt2) list
```

Format 3

$$\left\{\begin{array}{l}\underline{\text{READ}}\\ \underline{\text{WRITE}}\end{array}\right\}\ \text{file}$$

where: file is the file reference

opt1 is the error condition transfer

opt2 is the end-of-file condition transfer

list is the input/output specification

NOTE: These statements apply to word-oriented serial access files (i.e., binary sequential files).

RANDOM FILES

The unformatted random files created by FORTRAN are normally recorded in standard system format. The unformatted random file input/output statements have the following form:

Format

$\left\{\begin{matrix}\underline{\text{READ}}\\ \underline{\text{WRITE}}\end{matrix}\right\}$ (file'n,opt2) list

where: file is the file reference

n is an integer constant, a variable, or an expression that specifies the sequence number of the logical record to be accessed.

opt2 is the error condition transfer

list is the input/output specification

It is a requirement that FORTRAN random files have a constant record size. Furthermore, before any random I/O can be performed on any given file, its record size must be defined. This is accomplished with either a $ FFILE control card, in batch mode, or with a CALL to the (library) subroutine RANSIZ. Three arguments may be supplied: the first is a file reference, the second provides the record size. Each of these arguments can be any integer expression and are required. The third argument is zero or not supplied when the file is in standard system format. A nonzero value specifies a pure data file.

Example

```
CALL RANSIZ(08,50,0)
```

This statement specifies that file code 08 has a constant record size of 50 words and is in standard system format.

Linked files can be accessed in a random mode by using a CALL ATTACH and specifying random mode. Random files can also be written in a "pure data" format, without block serial numbers or record control words. This can be accomplished by one of the following:

```
$     FFILE    U,NOSRLS,FIXLNG/N  (batch mode only)
       or

      CALL     RANSIZ(U,N,1)
```

where: U and N are the file reference and logical record size parameters.

FILE HANDLING STATEMENTS

File handling statements provide for the manipulation of input/output devices for positioning of sequential files and demarcation of sequential files. The following file handling statements are described in Section III:

REWIND
BACKSPACE
ENDFILE

SECTION VI

SUBROUTINE AND FUNCTION STATEMENTS

The three basic elements of scientific programming languages -- arithmetic, control, and input/output -- are given added flexibility through subroutines. Subroutines are program segments executed under the control of another program and are usually tailored to perform some often-repeated set of operations. A subroutine is written only once, but can be used again and again; it avoids a duplication of effort by eliminating the need for rewriting program segments for use in common operations.

There are four classes of subroutines in FORTRAN:

- Arithmetic statement functions (ASF)
- Built-In intrinsic functions
- FUNCTION subprograms
- SUBROUTINE subprograms

The major differences among the four classes are

1. The first three classes can be grouped as functions
2. In the first three classes
   - A function has a single value in an expression
   - A function is referred to by an expression containing its name; a subroutine is referenced by a CALL statement
3. The first two classes are open subroutines (i.e, incorporated into the object program each time there is a reference in the source program). The latter two classes are closed (i.e., they appear only once in object form).

NAMING SUBROUTINES

All four classes of subroutines are named in the same manner as a FORTRAN variable. External subroutine names (i.e., FUNCTION and SUBROUTINE subprograms) have the additional requirement that they be unique within the first six characters. The following rules are applicable for all four classes:

1. A subroutine name consists of one to eight alphanumeric characters, the first of which must be alphabetic.

2. The type of a function, which determines the type of a result, is defined as follows:

   a. The type of a FUNCTION subprogram can be indicated by the name of the function or by writing the type (REAL, INTEGER, COMPLEX, DOUBLE PRECISION, LOGICAL, CHARACTER) preceding the word FUNCTION. In the latter case, the type implied by its name is overridden. The type of the FUNCTION subprograms in the Subroutine Library (the mathematical subroutines) is defined. Therefore, they do not need to be typed elsewhere.

   b. The type of a built-in intrinsic function is indicated within the FORTRAN compiler and does not need to appear in a type statement.

   c. Arithmetic statement functions have no type.

3. The name of a SUBROUTINE subprogram has no type and should not be defined, since the type of results returned is only dependent upon the type of the arguments returned by that subroutine.

ARITHMETIC STATEMENT FUNCTIONS

An arithmetic statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to the arithmetic assignment statement.

NOTE: In a given program unit, all statement function definitions must precede the first executable statement of that program unit. The name of a statement function must not appear in EXTERNAL, COMMON, EQUIVALENCE, NAMELIST, or ABNORMAL statements as a scalar name, or appears as an array name, in the same program unit.

An arithmetic statement function is defined by the format

function (arg[,...]) = exp

where: _function_ is the function name

arg is a symbolic name (referred to as a dummy argument)

exp is an expression

The purpose of the dummy argument is to indicate the order and the number of arguments. Arg can be actual variable names that appear elsewhere in the program unit with the following exceptions:

- EXTERNAL names
- ABNORMAL names
- PARAMETER names
- NAMELIST names
- SUBROUTINE, FUNCTION, or ENTRY names
- Arithmetic statement function names

Exp can be specified with expressions which may include

- Constants
- Scalar references
- Intrinsic function references
- References to other arithmetic statement functions
- External function references
- Array element references
- Indeterminate references

The last item in the above list, indeterminate references, covers the case where a dummy argument symbol appears in exp as the reference arg (exp). This syntax can imply a function reference or an array element reference. The decision is made each time the arithmetic statement function is referenced, and is determined by the actual argument in the ASF itself.

Example

```
1  DIMENSION  P(10)
2  F(A,B)=A(K)+B(K)
3  X=F(P,SIN)
```

Expansion of line 3 produces an equivalent assignment statement

```
3  X = P(K)+SIN(K)
```

NOTE: The first expression term is an array element reference while the second is a function reference.

Arithmetic Statement Function Left of Equals

An arithmetic statement function can be referenced on the left hand side of the equal sign in an assignment statement; however, it must expand into a scalar variable or an array element.

Example

```
AA (I,J) = J(I)
DIMENSION K(10)
    •
    •
AA (3,K) = 4*X (This expands to K(3) = 4*X)
```

Referencing Arithmetic Statement Functions

A statement function is referenced by using its name with a list of actual arguments in standard function notation in an expression. The actual arguments, which constitute the argument list, must agree in number with the dummy arguments in the function definition. An actual argument in a statement function reference can be any expression if the corresponding dummy argument appeared as a scalar reference. If the corresponding dummy argument appears as an indeterminate reference, then the actual argument must be an array or function name.

Execution of a statement function reference results in the association of actual argument values with the corresponding dummy arguments in the function definition, and an evaluation of the expression. The resulting value is then made available to the expression that contained the function reference.

Arithmetic statement functions have no type at the time of definition unless they have been explicitly typed. Type is introduced at the time of reference when the actual arguments are substituted for the dummy arguments. The arithmetic statement function is typed according to its actual arguments. If the arithmetic statement function expansion contains a combination of types, the respective types are examined according to the stated order of type dominance. The type of the recessive primary is converted to that of the dominant primary (if necessary) and the operation is performed.

NOTE: An explicitly typed arithmetic statement function retains that type regardless of its argument type.

Examples

```
     D(I,J) = I + J
     PRINT, D(1,2),D(1,2.0), D(1.0, 2.0)
     STOP; END
```

results would be: 3, 3.0E01, 3.0E01

```
     INTEGER D
     D(I,J) = I + J
     PRINT, D(1,2), D(1,2.0), D(1.0, 2.0)
     STOP; END
```

results would be: 3,3,3

At time of reference, the actual arguments are substituted for the dummy argument symbols. Type is introduced at this time and any ambiguities (such as the indeterminate reference described above) are resolved. References to other functions are classified as intrinsic, external, or other arithmetic statement function, at this time also. Thus, to reference another arithmetic statement function, the definition of that function may follow the definition of, but must precede any references to, this referencing function.

Examples

Defined

```
     ROOT (A,B,C)=(-B+SQRT(B**2-4*A*C))/(2*A)
```

Referenced

```
     ANS = ROOT(16.9,20.5,T+30)
```

BUILT-IN INTRINSIC FUNCTIONS

All functions in Table 6-1, except FLD, AND, OR, XOR, BOOL, and COMPL, are the standard FORTRAN intrinsic functions. The forty functions listed in the table are the built-in intrinsic functions supplied with FORTRAN. These intrinsic functions (with the exception of two functions) require only a few machine instructions and are inserted each time the function is used. To use these functions, it is necessary to write their names where needed and enter the desired expression(s) for argument(s). The names of the functions are established in advance and must be written exactly as specified.

Table 6-1. Built-in Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name for Automatic Typing	Specific Name	Type of	
					Argument	Function
Type Conversion	Conversion to Integer int(a) See Note 1	1		INT IFIX IDINT	Real Real Double	Integer Integer Integer
	Conversion to Real See Note 2	1		FLOAT SNGL REAL	Integer Double Complex	Real Real Real
	Conversion to Double See Note 3	1		DBLE	Real	Double
	Conversion to Complex See Note 4	2		CMPLX	Real	Complex
Truncation	int(a) See Note 1	1		AINT	Real	Real
Obtaining Absolute Value	|a| See Note 5 (ar**2+ai**2)**.5	1	ABS	IABS ABS DABS[1] CABS[1]	Integer Real Double Complex	Integer Real Double Real
Remaindering	a1-int(a1/a2)*a2 See Note 1	2	MOD	MOD AMOD DMOD[1]	Integer Real Double	Integer Real Double

[1] external function

Table 6-1 (cont). Built-in Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name for Automatic Typing	Specific Name	Type of Argument	Type of Function
Transferring Sign	$\lvert a1 \rvert$ if $a2 \geq 0$ $-\lvert a1 \rvert$ if $a2 < 0$	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Obtaining Positive Difference	$a1-a2$ if $a1 > a2$ 0 if $a1 < a2$	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Choosing Largest Value	max(a1,a2,...)	2 or more	MAX	MAX0 MAX1 AMAX1 AMAX0 DMAX1	Integer Real Real Integer Double	Integer Integer Real Real Double
Choosing Smallest Value	min(a1,a2,...)	2 or more	MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double
			- -	AMIN0 MIN1	Integer Real	Real Integer
Obtaining Imaginary Part of Complex Argument	ai See Note 6	1	-	AIMAG	Complex	Real
Conjugating Complex Argument to Real	(ar,-ai) See Note 6	1	-	CONJG	Complex	Complex

Table 6-1 (cont). Built-in Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name for Automatic Typing	Specific Name	Type of	
					Argument	Function
Logical "and"	$a_1 * a_2 * \ldots$	2 or more		AND	Real Integer	Typeless
Logical "or"	$a_1 + a_2 + \ldots$	2 or more		OR	Real Integer Typeless	Typeless
Logical "exclusive or"	$a_1 \oplus a_2 \oplus \ldots$	2 or more		XOR	Real Integer Typeless	Typeless
Ignore Type		1		BOOL	Any except Logical	Typeless
Extracting/ Inserting Bit Field	Beginning with bit a_1 of word a_3 extract a_2 bits	3		FLD	#1, #2 Integer #3 Any except Logical	Typeless
Logical One's Complement	-a	1		COMPL	Real Integer or Typeless	Typeless

Notes for Table 6-1:

1. For a of type integer, int(a)=a. For a of type real or double precision, there are two cases: if a < 1, int(a)=0; if a ≥ 1, int(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example,

   int(-3.7) = -3

   For a of type complex, int(a) is the value obtained by applying the above rule to the real part of a.

   For a of type real, IFIX(a) is the same as INT(a).

2. For a type real, REAL(a) is a. For a of type integer or double precision, REAL (a) is as much precision of the significant part of a as a real datum can contain. For a of type complex, REAL (a) is the real part of a.

   For a of type integer, FLOAT (a) is the same as REAL (a).

3. For a of type double precision, DBLE (a) is a. For a of type integer or real, DBLE (a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex, DBLE (a) is as much precision of the significant part of the real part of a as a double precision datum can contain.

4. CMPLX (a1,a2) is the complex value whose real part is REAL (a1) and whose imaginary part is REAL (a2).

5. A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

   CABS is defined as the absolute value of (ar**2+ai**2)**.5.

Argument Checking and Conversion for Intrinsic Functions

A number of checks on arguments used in the intrinsic functions are made by the compiler. Due to the inline code expansion, the number of arguments specified must agree with the number shown in Table 6-1. The argument type must also agree with the type of the function with the exception of the typeless intrinsic functions described below. Argument checking and/or conversion is carried out by the compiler using the following general rules:

1. The hierarchy of argument types considered for conversion is integer, real, double precision, and complex.

2. A generic intrinsic function call is transformed to the function type that supports the highest level argument type supplied to it.

3. Arguments to a non-generic form of intrinsic function are converted to conform with the function type specified. This is within the constraints of the argument types integer through complex.

Automatic Typing of Intrinsic Functions

Use of the generic forms of the mathematical intrinsic functions allows for the type of the function's value to be determined automatically by the type of the actual arguments supplied (refer to Table 6-1). The six generic intrinsic functions are

- Absolute value - ABS
- Remaindering - MOD
- Maximum value - MAX
- Minimum value - MIN
- Positive difference - DIM
- Transfer of sign - SIGN

This means that the inline code generated for DABS(D) and ABS(D) would be the same assuming that the type of the variable D is double precision.

When arguments of different types are specified (i.e., functions allowing more than one argument), the type of the function itself is determined by the same rules that govern mixed mode expressions (refer to Table 4-1).

Typeless Intrinsic Functions

FLD

FLD is a typeless function that is used for bit string manipulation.

Format

FLD (i,k,e)

where: <u>i</u> is an integer expression in the range $0 \leq i \leq 35$

<u>k</u> is an integer expression in the range $0 \leq k \leq 36$

<u>e</u> is any integer, real, or typeless expression; a word of character data, or any of the typeless functions

This function extracts a field of k bits from a 36-bit string having the value of e beginning with bit i (counted from left to right where the 0th bit is the leftmost bit of e). The resulting field is right-justified and the remaining bits are set to zero.

Example

```
I = 64
J = FLD (29,1,I)
PRINT, "I = ",I
PRINT, "J = ",J
```

would result in the printing of

```
I = 64
J = 1
```

This intrinsic function can also appear on the left-hand side of the equal sign in an assignment statement. When the FLD function is used in this manner, it must not be the first executable statement of the program or it will be interpreted as an arithmetic statement function.

Format

FLD (i,j,a) = b

where: i is an integer expression in the range $0 \leq i \leq 35$

j is an integer expression in the range $1 \leq j \leq 36$

a is a scalar variable or a subscripted variable

b is an expression

The j rightmost bits of b will be inserted into a beginning at bit position i.

Example

```
CHARACTER*4 A,B
A = "ABCD"
B = "1234"
FLD (9,9,A) = B
PRINT, A
PRINT, B
```

Assuming ASCII characters, this would result in the printing of

```
A4CD
1234
```

Additional Typeless Functions

The other five typeless functions are

Function	Usage
AND (el,e2)	Bit by bit logical product of el and e2.
OR (el,e2)	Bit by bit logical sum of el and e2.
XOR (el,e2)	Bit by bit "exclusive or" of el and e2.
BOOL (e)	The type of e is disregarded.
COMPL (e)	The one's complement of all bits in e are taken. The type of e is disregarded.

The expressions of e can be of type integer, type real, or typeless; e can also be a Hollerith word, the FLD word, or any of the typeless functions.

Examples:

```
M1 = AND(1,K)

M2 = OR(1,K)

M3 = XOR(1,K)

M4 = BOOL(K)

M5 = COMPL(K)
```

If all variables were of type integer, and the values of K were positive and odd, the following statements would have the same effect as the preceding examples:

```
M1 = 1; M2 = K; M3 = K -1; M4 = K; M5 = -K -1
```

If the receiving variables, M1 through M5, were of type LOGICAL, the values of the variables would be as follows:

K	M1	M2	M3	M4	M5
0	F	T	T	F	T
1	T	T	F	T	T
2	F	T	T	T	T
3	T	T	T	T	T
4	F	T	T	T	T
5	T	T	T	T	T
6	F	T	T	T	T
7	T	T	T	T	T
8	F	T	T	T	T
9	T	T	T	T	T

where: T is true
F is false

If the receiving variables were of type REAL, the values are stored in the locations of the receiving variables without conversion.

Character data type and integer data type operations can be mixed in the time sharing mode by using the BOOL function. In the next example, two-element array is employed to bypass the requirement of separating integer and character variables.

Example:

```
010  902 FORMAT (1X,I6,1X,A4)
020      INTEGER IX(2)
030      IX(1)=63
040      IX(2)=BOOL("ABCD")
050      IF(IX(2).EQ.BOOL("ABCD")) PRINT, "OK"
060      WRITE(6,902) IX
070      STOP;END

*RUN
OK
    63 ABCD
```

FUNCTION SUBPROGRAMS

Defining FUNCTION Subprograms

A FUNCTION subprogram is defined external to the program unit th references it.

Format

t FUNCTION function (arg[,...])

where: t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICA CHARACTER, or null

function is the symbolic name of the function to be defined

arg is referred to as a dummy argument and is a variable nan an array name, or an external procedure name

The symbolic name of the function must appear at least once in t subprogram as a variable name in some defining context (e.g., left of the equ sign). The value of the variable at the time of execution of any RETU statement in this subprogram is returned as the value of the function. Howeve the symbolic name of the function must not appear in any nonexecutable stateme in this program unit, unless it is the symbolic name of the function in t FUNCTION statement or in a Type statement.

An abnormal FUNCTION subprogram can define or redefine one or more of i arguments to effectively return results in addition to the value of t function.

The FUNCTION subprogram can contain any statements except BLOCK DA SUBROUTINE, another FUNCTION statement, or any statement that directly indirectly references the function being defined; but it must contain at lea one RETURN statement.

If the function name appears in any of the following contexts, redefinit of the function result is affected.

- Left of the equal sign in an assignment statement
- In the list of a READ statement
- In the list of a DECODE statement
- As the buffer name in an ENCODE statement
- As the induction variable of a DO loop

Redefinition can also occur if the function name appears in the argum list of a CALL statement or a reference to some abnormal external function.

NOTE: A function cannot be referenced in an input/output list if such reference causes any input/output operation to be executed.

Supplied FUNCTION Subprograms

The functions listed in Tables 6-2 and 6-3 are the external FUNCTION subprograms supplied with the FORTRAN compiler. Table 6-2 contains the supplied mathematical library functions. Table 6-3 contains the supplied nonmathematical external functions. To use these functions it is necessary to write the name where it is needed and enter the desired expression(s) for argument(s).

The type of a mathematical library function cannot be changed by implicit typing. However, implicit typing will affect the type of an external nonmathematical function. If the type of the function is affected by implicit typing, the function name must be included in an explicit type statement to obtain the desired results.

Example

```
IMPLICIT INTEGER (P-Z)
REAL RANDT
A=RANDT(10.0)
```

Argument Checking and Conversion for Supplied External Functions

A number of checks are made on the arguments used in the mathematical library functions. The compiler checks the type of the arguments supplied and makes conversions according to the following rules:

1. The hierarchy of argument types is integer, real, double precision, complex.
2. If the arguments in a generic function call do not conform as to type, the function call is transformed to the function type that supports the highest level of the argument supplied to it.
3. Integer arguments are converted to the type of the function being called.
4. The arguments of a non-generic form of external function are converted to conform to the function type specified. This is within the constraints of the argument hierarchy.

No tests or conversions are performed for the external nonmathematical functions outlined in Table 6-3. The number and type of arguments in the function call must agree with the number and type of arguments specified in Table 6-3.

Automatic Typing of Supplied Mathematical External Functions

When the mathematical library functions in Table 6-2 are referenced by their generic names, the function type is determined by the type of the argument(s). This means that the use of ARCOS(D) would generate a call to DARCOS(D) if the type of the variable D is double precision. The one exception to this rule is when an integer argument is specified for a generic function. In this case, the argument is converted from integer to real and the real form of the function is called.

The mathematical functions, listed by their generic names, which are automatically typed are:

ACOS	COS	ALOG2
ARCOS	COSH	ALOG10
ACOSH	CBRT	ALOG
ASIN	EXP	POW
ARSIN	EXP10	SIN
ASINH	EXP2	SINH
ATAN	EXPC	SQRT
ATAN2	EXPC2	TAN
ATANH	EXPC10	TANH

Note that the type of ATAN2 is double precision if at least one of its arguments is double precision.

Table 6-2. Supplied External FUNCTION Mathematical Subprograms

Function	Definition	No. of Arg.	Name	Type of Arg.	Type of Function
Arccosine[1]	cos (a)	1	ACOS[2]	Real	Real
		1	ARCOS	Real	Real
			DACOS	Double	Double
			DARCOS	Double	Double
Arccosine, Hyperbolic[1]	cosh (a)	1	ACOSH	Real	Real
		1	DACOSH	Double	Double
Arcsine[1]	sin (a)	1	ASIN[3]	Real	Real
			ARSIN	Real	Real
			DASIN	Double	Double
		1	DARSIN	Double	Double
Arcsine, Hyperbolic[1]	sinh (a)	1	ASINH	Real	Real
		1	DASINH	Double	Double
Arctangent[1]	tan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
	tan (a/b)	2	ATAN2[4]	Real	Real
		2	DATAN2	Double	Double
Arctangent, Hyperbolic[1]	tanh (a)	1	ATANH	Real	Real
		1	DATANH	Double	Double
Cosine[1]	cos(a)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Cosine, Hyperbolic	cosh(a)	1	COSH	Real	Real
		1	DCOSH	Double	Double
Cube Root	(a)	1	CBRT	Real	Real
		1	DCBRT	Double	Double
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Exponential	10^a	1	EXP10	Real	Real
		1	DEXP10	Double	Double
Exponential	2^a	1	EXP2	Real	Real
		1	DEXP2	Double	Double

[1]Arguments expressed in radians.
[2]ACOS and ARCOS are the same function. Either name can be used. DACOS and DARCOS are the same function. Either name can be used.
[3]ASIN and ARSIN are the same function. Either name can be used. DASIN and DARSIN are the same function. Either name can be used.
[4]The y-axis must be the first argument specified.

Table 6-2 (cont). Supplied External FUNCTION Mathematical Subprograms

Function	Definition	No. of Arg.	Name	Type of Arg.	Type of Function
Exponential Complement	$e^a-1.0$	1 1	EXPC DEXPC	Real Double	Real Double
Exponential Complement	$2^a-1.0$	1 1	EXPC2 DEXPC2	Real Double	Real Double
Exponential Complement	$10^a-1.0$	1 1	EXPC10 DXPC10	Real Double	Real Double
Logarithm, Base 2	$\log_2(a)$	1 1	ALOG2 DLOG2	Real Double	Real Double
Logarithm, Common	$\log_{10}(a)$	1 1	ALOG10 DLOG10	Real Double	Real Double
Logarithm, Natural	$\log_e(a)$	1 1 1	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Power	a^b	2 2	POW DPOW	Real Double	Real Double
Sine[1]	sin(a)	1 1 1	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Sine, Hyperbolic[1]	sinh(a)	1 1	SINH DSINH	Real Double	Real Double
Square Root	$(a)^{1/2}$	1 1 1	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Tangent[1]	tan(a)	1 1	TAN DTAN	Real Double	Real Double
Tangent, Hyperbolic[1]	tanh(a)	1 1	TANH DTANH	Real Double	Real Double

[1]Arguments expressed in radians.

Table 6-3. Supplied External FUNCTION Nonmathematical Subprograms

Function	Usage	No. of Args.	Type of Arg.	Type of Function
Left Shift	ILS (i,j)	2	Integer	Integer
Right Shift	IRS (i,j)	2	Integer	Integer
Left Rotate	ILR (i,j)	2	Integer	Integer
Right Logical	IRL (i,j)	2	Integer	Integer
Set Switch Word	ISETSW (i)	1	Typeless	Integer
Reset Switch Word	IRETSW (i)	1	Typeless	Integer
Mode	MODE (i)	1	Integer	Integer
Compare	KOMPCH (a,n,b,m,f)	5	Character, Integer	Integer
Random Number Generator	RAND (range)	1	Real	Real
Random Number Generator	RANDT (range)	1	Real	Real
Random Number Generator	FLAT (seed)	1	Real	Real
Random Number Generator	UNIFM2 (seed), mean,width)	3	Real	Real

Shift Functions

The shift functions shift the contents of the memory location specified by the integer variable i by j bit positions. (Refer to the Macro Assembler Program (GMAP) manual for a description of shifting functions.)

Function	Usage
ILS(i,j)	Left shift i by j bit positions.
IRS(i,j)	Right shift i by j bit positions.
ILR(i,j)	Left rotate i by j bit positions.
IRL(i,j)	Right logical i by j bit positions.

where: i and j are integer arguments

Set/Reset Program Switch Word

(Refer to the General Comprehensive Operating Supervisor (GCOS) manual for a description of the program switch word.)

<u>Function</u>	<u>Usage</u>
ISETSW(i)	Set program switch word.
IRETSW(i)	Reset program switch word.

The binary equivalent of the value of i determines the bit positions to be set/reset in the program switch word; the function returns the new program switch word configuration.

NOTE: Bits 0-17 of the program switch word cannot be changed when operating in the time sharing mode.

<u>Execution Mode Determination</u>

The mode determination is specified by the format

MODE(i)

If i = 1, the function value = 0 for batch execution; 1 for time sharing execution.

If i = 2, the function value = 0 for BCD character mode; 1 for ASCII character mode.

If i ≠ 1 or 2, the function value is always = -1.

<u>Character String Compare</u>

The character string comparison is specified by the format

KOMPCH (a,n,b,m,f)

where: <u>a</u> and <u>b</u> are character constants or variables
<u>f</u>, <u>m</u>, and <u>n</u> are integer expressions

String b, which begins at position m, is compared to string a, which begins at position n; f characters are compared.

If b = a, the function value = 0
If b is greater than a, the function value = +1
If b is less than a, the function value = -1

(Refer to Appendix A for BCD and ASCII character collation (sort) values.)

<u>Random Number Generators</u>

There are four separate functions provided for producing a sequence of random numbers. Each function provides a sequence of random numbers from a uniform (rectangular) distribution, which means that the probability of any number in the range occurring in the sequence is the same as any other number.

Calling Sequence

A = RAND (range)

where: $0 < A < \text{range}$

The range must be a real constant or variable; the seed = 1. The same set of random numbers is generated each time the program unit is executed.

Calling Sequence

A = RANDT (range)

where: $0 < A < \text{range}$

The range must be a real constant or variable; the seed is based on the time of day. A different set of random numbers is generated each time the program unit is executed.

Calling Sequence

A = FLAT (seed)

where: $0 < A < 1$

This version has a constant range but allows the seed to be varied. The seed must be a real constant or variable.

Calling Sequence

A = UNIFM2 (seed,mean,width)

where: $[\text{mean}-\text{width}/2] < A < [\text{mean}+\text{width}/2]$

This version allows the seed and the range to be varied.

Example

A = UNIFM2 (9.9,1.5,2.0)

generates a set of random numbers between 0.5 and 2.5 using the value 9.9 for the seed.

NOTE: The value of the initial argument (seed) passed to the function at the time of the first call initializes the algorithm for the generation of the sequence of random numbers. For all subsequent calls to the function during the execution of the same program unit, the value of the argument is ignored. All arguments must be real constants or real variables.

Referencing FUNCTION Subprograms

A FUNCTION subprogram is referenced by using its symbolic name with a list of actual arguments in standard function notation as a primary in an expression. The actual arguments must agree in order, number, and type with the corresponding dummy arguments in the FUNCTION subprogram definition. The actual arguments in the function reference can be

- A variable name
- An array element name
- An array name
- An expression
- The name of an external procedure
- A constant

If an actual argument is an external function name or a subroutine name, then the corresponding dummy arguments must be used as an external function name or a subroutine name, respectively. The type of the external function in the calling routine must match the type specified in the called function. If the dummy argument is defined or redefined in the referenced subprogram, the corresponding actual argument must be a scalar name, an array name, or an array element name.

Execution of a FUNCTION reference results in an association of the actual arguments with all the dummy arguments in the defining subprogram. Following these associations, execution of the first executable statement of the defining subprogram begins.

- If the actual argument is an expression or a constant, the association is by value rather than by name.
- If the actual argument is an array element name with variables in the subscript, it can be replaced by the same argument with a constant subscript that contains the value(s) that would result from computing the variable subscript just before the function takes place.
- If the dummy argument is an array name, the corresponding actual argument must be an array name or an array element name.
- Unless it is a dummy argument, a FUNCTION subprogram is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

NOTE: If a user FUNCTION subprogram is written in a language other than FORTRAN, it is the user's responsibility to insure that the correct indicators, as well as valid numeric values, are returned to the calling program.

Example

Definition of a FUNCTION subprogram

```
        FUNCTION DIAG (A,N)
        DIMENSION   A (N,N)
        DIAG = A(1,1)
        IF (N .LE. 1) RETURN
        DO 6 I = 2, N
  6     DIAG = DIAG * A(I,I)
        RETURN
        END
```

Reference to a FUNCTION subprogram

```
        DIMENSION X (8,8)
        DET = DIAG (X,8)
```

SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram differs from a FUNCTION subprogram in three ways:

1. A SUBROUTINE has no value associated with its name. All results are defined in terms of arguments or common, and there may be any number of results.

2. A SUBROUTINE is not called into action simply by writing its name, because a value is not associated with the name. A CALL statement which specifies the arguments and stores all output values is used to bring the SUBROUTINE into operation.

3. The naming of a SUBROUTINE is similar to the FUNCTION subprogram without any type association.

NOTE: It is the user's responsibility to insure that the number and type of arguments in the calling program statement corresponds with the number and type of arguments in the called routine. This applies to all subroutines and functions (library or other).

Defining SUBROUTINE Subprograms

A SUBROUTINE statement is specified by the format

SUBROUTINE sub [(arg[,...])]

where: sub is the symbolic name of the SUBROUTINE to be defined

arg is a dummy argument and can be a variable name, an external procedure name, or an alternate return.

The variable names in the dummy argument list cannot appear in an EQUIVALENCE, COMMON, NAMELIST or DATA statement.

The SUBROUTINE subprogram can define or redefine one or more of its arguments to effectively return results; it can contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined, and must contain at least one RETURN statement.

Referencing SUBROUTINE Subprograms

A SUBROUTINE is referenced by a CALL statement. The actual arguments which constitute the argument list must agree in order, number, and type with the corresponding dummy arguments in the defining subprogram. The actual arguments in the subroutine call can be

- A constant
- A variable name
- An array element name
- An array name
- An expression
- The name of an external procedure
- An alternate return

Execution of a subroutine reference results in an association of the actual arguments with all the dummy arguments in the defining subprogram. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

- An actual argument that is an array element name with variables in the subscript can be replaced by the same argument with a constant subscript that contains the same values that would be derived by computing the variable subscript just before the association of arguments takes place.
- If a dummy argument is an array name, the corresponding actual argument must be an array name or an array element name.
- If a SUBROUTINE reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in COMMON, a definition of either entity within the subprogram is prohibited.
- If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element, or an array name (e.g., it should not redefine a constant).
- Unless it is a dummy argument, a SUBROUTINE is also referenced by its appearance in an EXTERNAL statement.

Examples

Definition of a SUBROUTINE Subprogram

```
      SUBROUTINE LARGE (ARRAY,I,BIG,J)
      DIMENSION ARRAY (50,50)
      BIG=ABS (ARRAY(I,1))
      J=1
      DO 6 K=2,50
      IF (ABS (ARRAY(I,K)) .LE. BIG) GO TO 6
      BIG=ABS (ARRAY(I,K))
      J=K
    6 CONTINUE
      RETURN
      END
```

Reference of a SUBROUTINE Subprogram

```
      CALL LARGE (ZETA,N,VAL,NCOL)
```

RETURN STATEMENT

The RETURN statement is used to terminate all subprograms and causes control to be returned to the calling program. There may be multiple RETURN statements in a subprogram.

Format

```
      RETURN [i]
```

where: i is an integer, constant, or variable whose value denotes the nth alternate return (* or $) in the argument list of the SUBROUTINE statement, reading from left to right.

Following the RETURN statement of a subprogram, control passes to the next executable statement which follows the CALL in the calling program. It is possible to return to any numbered executable statement in the calling program by using one of the following formats from the called subprogram. This return must not violate the transfer rules for DO loops.

NOTE: The function reference may be part of an expression, and when control returns from the function, evaluation of the expression continues.

Alternate Return Formats

Format 1

CALL subr ($alt[,...])

where: $ is used to designate the argument for the alternate return

alt is a statement label or a switch variable that specifies the alternate return in a subroutine

NOTE: Alternate returns cannot be used with functions.

Format 2

SUBROUTINE subr $\left(\begin{Bmatrix} * \\ \$ \end{Bmatrix} \text{alt}[,...]\right)$

where: * and $ are used to designate the argument in an alternate return

Example

Calling Program	Called Subroutine

```
       .
       .
       .
10  CALL SUB(A,B,C,$30,$40)
20  .....
       .
       .
       .
30  SUM = A+C
       .
       .
       .
40  PROD = SUM**B
       .
       .
       .
    END
```
	```
	    SUBROUTINE SUB(X,Y,Z,*,*)
	       .
	       .
	       .
	100 RETURN  N
	       .
	       .
	       .
	    END
	```

Execution of statement 10 in the calling program causes entry into the subprogram SUB. If statement 100 in subprogram SUB is executed, the return to the calling program will be to statement 20, 30, or 40 if N is zero, one, or two, respectively.

Alternate returns may best be understood by considering that a CALL statement that uses the alternate return is equivalent to a CALL and a computed GO TO statement in sequence.

Example

```
     CALL NAME (P,$20,Q,$35,R,$22)
```

is equivalent to

```
     CALL NAME (P,Q,R,I) IF (I.NE.O) GO TO (20,35,22), I
```

where: I is set to the value of the integer in the RETURN statement executed in the called subprogram.

NOTE: Calling arguments for alternate returns are not coded by the compiler in the same manner as the standard arguments. Therefore, this will need to be considered for the coding of any GMAP subroutines.

If the RETURN index is not specified or is zero, a normal (rather than alternate) return is made to the statement immediately following the CALL. The intermingling of arguments and alternate returns can be done freely in both the CALL and SUBROUTINE statements. The compiler separates the combined list into two separate lists, such that argument n is the nth actual or dummy argument, and alternate return n is the nth statement number or * or $, reading left to right. Thus, the following statements are equivalent:

Example

```
     CALL NAME (P,$20,Q,$35,R,$22)
     CALL NAME (P,Q,R,$20,$35,$22)
     CALL NAME ($20,$35,$22,P,Q,R)

     SUBROUTINE NAME (S,*,T,*,U,*)
     SUBROUTINE NAME (S,T,U,*,*,*)
     SUBROUTINE NAME (*,*,*,S,T,U)
```

Multiple Entry Points Into a Subprogram

The normal entry into a SUBROUTINE subprogram from the calling program is made by a CALL statement that refers to the subprogram name. The normal entry to a FUNCTION subprogram is made by a function reference in an expression. In both cases, entry is made at the first executable statement following the FUNCTION or SUBROUTINE statement.

It is also possible to enter a subprogram at an alternate entry point by a CALL statement or a function reference that refers to an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement.

Because ENTRY statements are nonexecutable, they do not affect control sequencing during normal execution of a subprogram. The order, type, and number of arguments do not need to agree between the SUBROUTINE or FUNCTION statement and any ENTRY statement; nor do ENTRY statements have to agree among themselves in these respects. Each CALL or FUNCTION reference, however, must agree in order, type, and number of actual arguments with the dummy arguments of the SUBROUTINE, FUNCTION, or ENTRY statement that it references. No subprogram can refer to itself directly or through any of its entry points. In addition, it must not refer to any other subprogram whose RETURN statement has not been satisfied.

NOTE: A program loop may result if this condition occurs.

Example

```
     Calling Program                Called Program

       .                            SUBROUTINE SUB1(U,V,W,X,Y,Z)
       .                                .
       .                                .
     1 CALL SUB1 (A,B,C,D,E,F)          .
       .                         10 U = V
       .                                .
       .                                .
     2 CALL SUB2(G,H,P)                 .
       .                            GO TO 60
       .                                .
       .                                .
                                    ENTRY SUB2(T,U,V)
                                    GO TO 10
     3 CALL SUB3                 60
       .                                .
       .                            GO TO 90
       .                            ENTRY SUB3
       END                              .
                                 90 RETURN
                                    END
```

The execution of statement 1 causes entry into SUB1 at the first executable statement of the subroutine. Execution of statements 2 and 3 also cause entry into the called program at the first executable statement following the ENTRY SUB2(T,U,V) and ENTRY SUB3 statements, respectively.

Example

The coding for a multiple-entry FUNCTION subprogram that will execute properly:

```
     Calling Program                Called Program

       .                            FUNCTION ADD1(N)
       .                            ADD1 = N + 1
       .                            GO TO 30
     I = ADD1(1)                    ENTRY ADD2 (N,M)
     J = ADD2(1,1)                  ADD1 = N + M + 1
     STOP; END                   30 CONTINUE
                                    RETURN; END
```

The coding for a multiple-entry FUNCTION subprogram that will not execute properly:

```
Calling Program              Called Program

    .                        FUNCTION ADD1(N)
    .                        ADD1 = N + 1
    .                        GO TO 30
I = ADD1(1)                  ENTRY ADD2(N,M)
J = ADD2(1,1)                ADD2 = N + M + 1    (must be ADD1)
STOP; END                 30 CONTINUE
                             RETURN; END
```

Within the calling program, one must refer to the entry name specified in the ENTRY statement at which he wants to enter the function subprogram. Within the function subprogram, however, with the exception of the entry statements, the function name must be used as it was specified in the FUNCTION statement.

Dummy Argument

A dummy argument is used to make entities in a calling program available to the called subprogram, and can be used in the subprogram as a scalar variable, an array, a subroutine, or a function name.

The dummy argument of a subprogram can be associated with an actual argument that is

- a scalar variable
- an array
- an array element
- a subroutine
- an external function
- a constant
- an expression
- a statement number to which a special return can be made from a subroutine program

When dummy arguments are used, they must adhere to the following rules:

1. When a statement number is specified, the use of the * or $ in a dummy argument position is required if a statement number is associated with that dummy argument.

2. When an external function name is specified, the use of a dummy argument is permissible if an external function name is associated with that dummy argument.

2. When a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type is made available through the argument association.

4. When a variable, array, or array element name is specified, the use of a dummy argument is permissible if a proper association with an actual argument is made.

SUPPLIED SUBROUTINE SUBPROGRAMS

Table 6-4 contains an alphabetical list of FORTRAN supplied SUBROUTINE subprograms and descriptions.

Table 6-4. Supplied SUBROUTINE Subprograms

Subprogram	Use	Call
ATTACH	Access existing permanent file.	ATTACH (lgu,catfil,iprmis, mode, istat, buffer)
CALLSS	Call a time sharing subsystem	CALLSS (string,name)
CNSLIO	Console communications.	CNSLIO (console,message, nwords,nreply,nrepws)
CONCAT	Move character substring.	CONCAT (a,n,b,m,f)
CORFL	Move data from/to 10-word file	CORFL (loc,i,j,k)
CORSEC	Memory allocation x processor time.	CORSEC (a)
CREATE	Create temporary mass storage or terminal file.	CREATE (lgu,isize,mode, istat)
DATIM	Get current date and time.	DATIM (d,t)
DEFIL	Create temporary file.	DEFIL (name,links,mode, istat)
DETACH	Deaccess current file.	DETACH (lgu,istat,buffer)
DUMP (BCD) DUMPA (ASCII)	Dump designated area of memory in specified format, terminate execution.	DUMP [DUMPA] $(a_1,b_1,i_1 \ldots)$
DVCHK	Divide check test.	DVCHK(j)
EXIT	Flush buffers to an external device and terminate current activity.	EXIT
FCLOSE	Close file, flush, and release buffers.	FCLOSE(i)
FILBSP	Backspace files on multi-file tape.	FILBSP (lgu,n)
FILFSP	Forward space files on multifile tape.	FILFSP (lgu,n)
FLGEOF	End of file processing.	FLGEOF (i,j)
FLGERR	Data error processing.	FLGERR (i,j)
FLGFRC	File and Record Control I/O error recovery.	FLGFRC (lgu,return)
FMEDIA	Output transliteration.	FMEDIA (fc, media code)
FPARAM	Set or reset I/O parameter.	FPARAM (i,j)
FXDVCK	Divide and check fault test.	FXDVCK (r,m)

Table 6-4 (cont). Supplied SUBROUTINE Subprograms

Subprogram	Use	Call
FXEM	Placement of error code.	ANYERR (v)
	Display of error trace.	FXEM (code, message,n)
	Alter FXEM switch word.	FXOPT (code, i_1,i_2,i_3)
	Set alternate error procedure location.	FXALT (SR)
	Alternate error return.	FXALT ($n)
LINK	Restore link and transfer to its entry point.	LINK (name)
LLINK	Restore link and return to next statement in calling subroutine.	LLINK (name)
MEMSIZ	Memory allocated.	MEMSIZ (j)
NASTRK	Disable asterisk-fill for field overflow on formatted output.	NASTRK
OVERFL	Exponent register overflow or underflow test.	OVERFL (j)
PDUMP (BCD) PDUMPA (ASCII)	Dump designated area of memory in specified format, return.	PDUMP [PDUMPA] ($a_1,b_1,i_1,..$)
PTIME	Processor time used for this activity.	PTIME (a)
RANSIZ	Specify record size of random file.	RANSIZ (u,m,n)
SETBUF	Define buffer for file I/O.	SETBUF (i,a,b)
SETFCB	Define file control block.	SETFCB (a,i,j)
SETLGT	Define logical file table.	SETLGT (a,i)
SLITE	Clear all sense lights. Turn on sense light.	SLITE (0) SLITE (i)
SLITET	Test and turn off sense lights.	SLITET (i,j)
SORT ISORT	Sort in ascending order.	SORT [ISORT] (array,nrec,lrs,key ,...)
SORTD ISORTD	Sort in descending order.	SORTD [ISORTD] (array,nrec,lrs,key ,...)
SSWTCH	Test sense switch.	SSWTCH (i,j)
TERMNO	Station code.	TERMNO (a)
TERMTM	Hours of log-on.	TERMTM (a)
TRACE	Trace and debug.	TRACE
USRCOD	User identification.	USRCOD (s)

Table 6-4 (cont). Supplied SUBROUTINE Subprograms

Subprogram	Use	Call
YASTRK	Re-establish asterisk-fill for field overflow on formatted output.	YASTRK
ATCALL	Callable portions of the FORTRAN Debugging System (See Appendix F)	ATCALL (subr)
FDEBUG		FDEBUG (di,do)
FDUMP		FDUMP
FTERM		FTERM
FTIMER		FTIMER
NOCALL		NOCALL (subr)
NTCALL		NTCALL (subr)

ATTACH

This subroutine is used to access an existing permanent file in batch or TSS mode.

Calling Sequence

```
CALL ATTACH (lgu,catfil,iprmis,mode,istat,buffer)
```

where: <u>lgu</u> is an integer variable or constant and is the usual FORTRAN file code.

<u>catfil</u> a character constant or variable is the catalog/file descriptor containing the catalog/file string and must be terminated by a semicolon. Embedded blanks are ignored. The master catalog password is not needed; however, subsequent passwords are required if they are part of the file description. Alternate names may be required if the file name is longer than eight characters or when non-FORTRAN subroutines are reading the same file.

<u>iprmis</u> is an integer variable or constant that gives the desired permission. Those are ORed with any permission in catfil.

= 1 READ ONLY
= 2 WRITE ONLY
= 3 READ and WRITE
= Any other (This is undefined and subject to change.)

<u>mode</u> is an integer variable or constant
= 0 Get file as defined
= 1 Get file as random
= 2 Get terminal

<u>istat</u> an integer variable, contains the octal value of the status word returned by the File Management Supervisor or contains:

0 = successful (batch mode only)
1 = file is currently open
2 = terminal requested in batch mode (illegal)
3 = additional memory needed, request denied
(time sharing user is terminated)
4 = <u>catfil</u> all blanks
others = refer to the <u>TSS System Programmer's Reference Manual</u>

<u>buffer</u> = Null: Get a work area for the File System.
= Not null: Use this array as a work area (at least 380 words).

Example of a Null Argument

```
     CALL ATTACH (lgu, "catfil;", iprmis,mode,istat,)
```

Upon successful return from ATTACH in time sharing, an FCB will have been created, and the file name (or alternate name) is in the FCB -10, -9 (in ASCII). If the file was in the AFT with a subset of the desired permissions, it is deaccessed and reaccessed with the new permissions.

CALLSS

This subroutine calls a time sharing subsystem and returns to the calling program.

Calling Sequence

```
CALL CALLSS (string)

      or (for some time sharing subsystems)

CALL CALLSS (string,name)
```

where: <u>name</u> is the four-character constant or variable that is the internal TSS name of the subsystem to be called. If name is not supplied, the first four characters in string are used for name. The name used as the argument may be different than the name used as a system command.

<u>string</u> is the command to invoke the subsystem and is a constant or variable containing a carriage return or a reverse slant as the terminating character.

Example

CALL CALLSS ("FRN P3\")	(the FORTRAN subsystem is invoked and program P3 is executed)
CALLSS ("BRNbP2\")	(the BASIC subsystem is invoked and program P2 is executed)
CALLSS ("CATALOG FILENAME\")	(the specific attributes of the file FILENAME are printed)
CALLSS ("ABCb\")	(the ABACUS subsystem is invoked)
CALLSS ("FDUMP\")	(the FDUMP subsystem is invoked)
CALLSS ("RUN P4\",\"CDIN")	(the CARDIN subsystem is invoked and file P4 is executed)

Nesting to more than two levels using CALLSS may produce unpredictable results. If the called time sharing system is SYSTEM, control is not returned to the calling program.

CNSLIO

This subroutine permits operator-program communication via the console typewriter. Return is made to the next executable statement in the calling program. This subroutine is restricted to batch execution; it may not be called by a FORTRAN program executing in the time sharing mode. It is suggested that limited use be made of this subroutine since it tends to distract the attention of the console operator.

Calling Sequence

```
CALL CNSLIO(console, message,nwords[,nreply,nrepws ])
```

where: console is defined as CHARACTER*6, or as integer, and is initialized with

"0000T/" for master console
"0000T*" for tape console
"0000*T" for unit record console
"0000/T" for special purposes

If no initialization is given, "0000T/" is assumed.

message is an array containing the CHARACTER message to be printed on the console

nwords is an integer variable or constant, representing the number of words to be printed on the console. Any value greater than 11 is set to 11 (66 characters).

nreply is an optional integer variable and is used when a reply is desired. When present, the operator reply (in BCD) is stored at location nreply.

nrepws is an optional integer variable or constant that is used when a reply of more than six characters is desired (maximum of 11 words). When omitted, a one-word reply is stored in nreply. When provided, nrepws words (nrepws *6 characters are stored at location nreply). If nreply and nrepws are not provided, the delimiting commas must also be omitted from the argument list.

CONCAT

This subroutine is used to provide the user with the ability to move a character substring of arbitrary length and position within a string.

Calling Sequence

```
CALL CONCAT (a,n,b,m,l)
```

where: a is a character variable whose character string is to be replaced

n is the first replaced character of a (n = 1 implies first character)

b is a character constant or variable which is the replacement character string

m is the leftmost replacement character of b (m = 1 implies first character)

l is the number of characters to be replaced; if l is not given, 1 is assumed

String a, beginning at position n, is replaced by string b, beginning at position m; l characters are replaced. m, n, and l are integer variables or constants. n through (n + l-1) of a are replaced with characters (m+l-1) of b.

Example

```
0010  CHARACTER A*20/"FIFTEEN WERE THERE  "/
0020  CHARACTER B*20/"SIXTEEN WERE ABSENT "/
0030  PRINT A,B
0040  CALL CONCAT (A,1,B,1,3)
0050  PRINT,A,B
0060  STOP;END

READY

*RUN
FIFTEEN WERE THERE   SIXTEEN WERE ABSENT
SIXTEEN WERE THERE   SIXTEEN WERE ABSENT
```

CORFL

This subroutine enables the time sharing user to move data from or to the ten-word memory file (see "DRL CORFIL", _TSS System Programmer's Reference Manual_).

NOTE: This call is ignored in batch.

Calling Sequence

CALL CORFL (loc,i,j,k)

where: _loc_ is the integer array into which or from which the data is to be moved

_i_ is the number of words to be moved ($1 \leq I \leq 10$)

_j_ is the relative location in the ten-word file at which the transfer is to begin.

_k_ = 0, data is transferred into the ten-word file
= 1, data is transferred from the ten-word file

i,j,k are integer variables or constants.

CORSEC

This subroutine provides the means of obtaining the product of a memory allocation and processor time.

Calling Sequence

```
CALL CORSEC (a)
```

where: _a_ is a real variable whose returned value is the product of 1024-word blocks currently allocated and processor time in seconds. This subprogram can also be used as a function.

Example

```
IF (CORSEC(a).GT.b)....
```

CREATE

This subroutine is used to create and access a temporary mass storage or terminal file.

Calling Sequence

```
CALL CREATE (lgu,isize,mode,istat)
```

where: <u>lgu</u> (integer variable or constant) is the usual FORTRAN file code.

<u>isize</u> (integer variable or constant) is the size, in words, of the desired file and must be present (not used if mode = 2)

<u>mode</u> is an integer variable or constant
= 0 for a linked mass storage file
= 1 for a random mass storage file
= 2 for a terminal file

<u>istat</u> is an integer variable status return word. The following codes apply.

= 0, successful
= 1, mode is invalid
= 2, file is currently open
= 3, no room in AFT
= 4, temporary file not available
= 5, duplicate file name
= 6, no room in PAT
= 7, illegal device specified

If the CREATE is successful, a FCB is created and the file code (in ASCII), is placed in FCB-10, -9.

If a $ FFILE card is used to create a FCB for a specific file code, a CALL CREATE which specifies that same file code destroys the FCB setup by the $ FFILE card and creates a new FCB.

DATIM

This subroutine is used to obtain the current date and time.

Calling Sequence

CALL DATIM (d,t)

where: *d* is an eight-character variable that gives the date as mm/dd/yy (with trailing blanks in BCD mode; e.g., MM/DD/YYb̸b̸b̸b̸).

t is a real variable that gives the time of day in hours as a floating-point number.

DEFIL

This subroutine creates a named temporary file and accesses it in the user's available file table. The call is applicable only for time sharing activities.

Calling Sequence

```
CALL DEFIL (name,links,mode,istat)
```

where: name is a character constant or a variable containing the ASCII name of the temporary file to be created; name must consist of a minimum of five characters and a maximum of eight characters.

links is the size of the file to be created (in links)
1 link = 3840 words

mode = 0, sequential file is created
≠ 0, random file is created

istat is the status indication returned as follows:

0 = successful
3 = no room in AFT
4 = temporary file not available
5 = duplicate file name
6 = no room in PAT

DETACH

This subroutine is used to close the file, release its buffer, and deaccess the file. If in TSS mode, the file is removed from the AFT.

Calling Sequence

```
CALL DETACH (lgu,istat,buffer)
```

where: lgu is an integer variable or constant and is the FORTRAN file code.

buffer = null argument: get space for FILSYS
= not null: use this variable array as buffer space (at least 380 words)

istat is an integer variable that is used as a status return word.
= 0: successful
= 1: could not get FILSYS buffer (batch only); time sharing user is terminated.

NOTE: If more memory is required (to deaccess the file) and the request is denied, the time sharing user is terminated.

DUMP DUMPA, PDUMP PDUMPA

This subroutine subprogram dumps all or designated areas of memory that have been allocated to selected variables in a specified format. If DUMP is called, execution is terminated by a call to EXIT. If PDUMP is called, control is returned to the calling program. (DUMPA and PDUMPA are used for ASCII).

Calling Sequence

CALL $\left\{\begin{matrix}\text{DUMP}\\\text{DUMPA}\end{matrix}\right\}(a_1,b_1,j_1,\ldots,a_n,b_n,j_n)$

CALL $\left\{\begin{matrix}\text{PDUMP}\\\text{PDUMPA}\end{matrix}\right\}(a_2,b_2,j_2,\ldots,a_m,b_m,j_m)$

where: a is the beginning variable of the area to be dumped.

b is the ending variable of the area to be dumped.

j is an integer specifying the dump format

= null argument or 0, octal
= 1, Integer
= 2, Real
= 3, Double Precision
= 4, Complex
= 5, Logical
= 6, Character

NOTE: If no arguments are supplied, (e.g., CALL PDUMP), all of memory is dumped in octal.

DVCHK,OVERFL,FXDVCK

These subroutine subprograms check logical fault vector locations in the slave program prefix (refer to the General Comprehensive Operating Supervisor (GCOS) Manual).

- DVCHK determines if a divide check has occurred.
- OVERFL determines if an exponent register overflow or underflow has occurred.
- FXDVCK allows another value to be returned after a divide check fault.

  NOTE: This subroutine must be called prior to the statement that might cause the fault. FXDVCK is incompatible with MOD. FXDVCK allows a user-supplied value to be returned as the quotient, whereas, MOD returns the remainder, not the quotient.

Calling Sequence

```
CALL  DVCHK(j)
CALL  OVERFL (j)
CALL  FXDVCK (r[,m])
```

where: *j* is an integer variable

= 1 : divide check,
exponent register overflow
exponent register underflow
= 2 : no fault vector

r is the double precision number that is to be used after a floating-point divide check.

m is the integer that is to be used after an integer divide check.

The FORTRAN fault processor processes integer and floating-point divide check faults, and exponent register overflow/underflow faults. A message is printed on file 06 stating the type of fault and the location at which the fault occurred. Execution continues with one of the following values returned

Fault	Value Returned	
Divide check (integer)	No change	Unless CALL FXDVCK is used.
Divide check (floating-point)	A large floating-point value[1]	Unless CALL FXDVCK is used.
Overflow (integer)	No change	
Exponent overflow	A large positive or large negative floating-point value	
Exponent underflow	Floating-point zero	

[1]Allows further computation without another immediate fault. This value is set to approximately ± 10**36.

EXIT

This subroutine subprogram flushes all buffers to output files and terminates the current activity. Control is returned to the General Comprehensive Operating Supervisor.

Calling Sequence

```
CALL EXIT
```

FCLOSE

This subroutine subprogram closes a file without rewinding it and releases the buffer(s) assigned if it is the standard size (320 words). Return is to the next executable statement in the calling program.

Calling Sequence

```
CALL FCLOSE(i)
```

where: i is an integer variable or constant logical file designator.

FILBSP,FILFSP

These subroutines can only be used with tape files. They allow multifile tapes to space from one file to another.

Calling Sequence

```
CALL FILBSP (lgu,n) backspace n files

CALL FILFSP (lgu,n) forwardspace n files
```

where: lgu is an integer variable or constant file code

n is the number of files to be skipped (integer variable or constant)

To ensure proper positioning, the current file, if output, should be closed with an ENDFILE statement and counted as one of the files to be backspaced. Declare the files to be multifile and unlabeled by use of a $ FFILE card.

Example

```
$ FFILE  xx,MLTFIL,NSTDLB
```

NOTE: On an EOF condition, an additional backspace must be executed to get past the end-of-file condition.

FLGEOF

This subroutine subprogram provides a signal to request return to the calling subprogram if an end-of-file condition occurs. Return is to the next executable statement in the calling program.

Calling Sequence

CALL FLGEOF(i,j)

where: i is the logical file designator, an integer variable, or a constant.

j is an integer variable used to indicate an end-of-file condition and should not be used for any other purpose.

If an end-of-file condition could have occurred, j must be tested for zero. If $j \neq 0$, an end-of-file condition did occur; if $j = 0$, an end-of-file was not encountered.

NOTE: Use of the END= option in an I/O or ENCODE/DECODE statement is preferable to calling FLGEOF.

FLGERR

This subroutine subprogram provides a means of checking data errors. Return is to the next executable statement in the calling program.

Calling Sequence

```
CALL FLGERR(i,j)
```

where: _i_ is the logical file designator, an integer variable, or a constant.

_j_ is an integer variable used to indicate an input data error (GFRC error) and should not generally be used for any other purpose.

If an error condition could have occurred, j must be tested for zero. If j ≠ 0, an error condition did occur; if j = 0, an error condition was not encountered.

NOTE: Use of the ERR= option in an I/O or ENCODE/DECODE statement is preferable to calling FLGERR.

FLGFRC

This subroutine provides some control of the File and Record Control errors by setting an error routine address into the file control block (refer to the File And Record Control manual). This subroutine should be called prior to the first I/O for this file.

Calling Sequence

CALL FLGFRC (lgu,ptr)

where: lgu is an integer variable or constant representing the numeric file code

ptr is the name of the recovery subroutine or an alternate return to a label in the same program

Any File and Record Control error that would take the "user-supplied routine" exit will cause transfer of control to the ptr recovery subroutine or label after the printing of a message and status code (refer to the File and Record Control manual for details of the user-supplied routine).

NOTE: Essentially, a GMAP CALL to the routine ptr is generated so that a FORTRAN subroutine could obtain the status code.

Example

```
     SUBROUTINE RECV(IFC,IERR)
     PRINT 200, IERR,IFC
200  FORMAT ("ERROR#",I4, "b̸OCCURRED ON FILE #", I3)
     RETURN
     END
```

Return is to the GFRC routine which detected the error.

FMEDIA

This subroutine allows the user to cause transliteration to occur on files directed to mass storage or tape.

Calling Sequence

```
CALL FMEDIA (fc,media)
```

where:	fc	is the logical file code (integer variable or constant) = 0 for BCD no printer slew control = 2 for BCD card images = 3 for BCD with printer slew control = 6 for standard system ASCII format (no slew) Other codes are ignored
	media	code (integer variable or constant) specifies the desired media code

The legal combinations are as follows:

0 to 2	3 to 0
0 to 3	3 to 2
0 to 6	3 to 6
2 to 0	6 to 0
2 to 3	6 to 2
2 to 6	6 to 3

NOTE: CALL FMEDIA should not be used if the batch transliteration routine (.GXLIT) is being loaded. (Refer to the File and Record Control manual.)

FPARAM

This subroutine permits the user to set or reset some of the I/O parameters of the run-time library. Specifically, it can be used to

1. Set the line length (multiple of four) for formatted output directed to a terminal; the default setting for this parameter is 72. The maximum line length is 160 characters.
2. Set the media code for unformatted file output; the default setting of this parameter is 1.
3. Set the reflexive read characters that are sent to a terminal to request input; the default setting of this parameter is a string of four ASCII characters, carriage return, line feed, equal sign, or X-ON.

Calling Sequence

```
CALL FPARAM (i,j)
```

where: _i_ is an integer variable or constant, with a value of 1, 2, or 3 corresponding to one of the three functions above.

_j_ is an integer variable or constant, providing the line length or media code for i values of 1 and 2, or providing the octal value of four ASCII characters for an i value of 3.

Examples

```
DATA J/O015012077077/
CALL FPARAM (3,J)
```

Reflexive read signature changed to "??"; in which 015 is a carriage return, 012 is line feed, and the two 077s are question marks.

```
CALL FPARAM (1,160)
```

Terminal line length setting to 160 characters.

FXDVCK (see DVCHK)

FXEM (FORTRAN EXECUTION ERROR MONITOR)

This subroutine performs the following functions:

- Prints a trace of subroutine calls.
- Prints execution error messages.
- Terminates execution with a Q6 abort, continues with execution of the program, or transfers to an alternate error routine.
- Allows the user to determine if an error has been processed by the Execution Error Monitor.

These functions are accomplished by the setting/resetting of bits in switch word groups that control termination, message printing and trace, and alternate error return for the errors described in Table 6-5 (refer to Appendix E for FXEM examples).

Calling Sequences

CALL ANYERR(v)

where: v is an integer variable into which the FORTRAN Execution Error Monitor places the error code (see Table 6-5) if an error occurs; v should not be used for any other purpose.

NOTE: If an ERR=CLAUSE is included in an I/O operation, the error monitor is not called. Control is returned to the statement number specified.

Calling Sequence

```
CALL FXEM(ncode [,msg,n ] )
```

where: ncode is the error code expressed as an integer in the range $1 \leq n \leq 143$ (refer to Table 6-5).

msg is the message displayed on file 06 following the error trace; msg must be a character constant or a variable.

n is the number of words to be printed and must be in the range $0 \leq n \leq 20$.

NOTE: If only the ncode is specified, only the trace is printed.

Calling Sequence

```
CALL FXOPT(ncode,i1,i2,i3)
```

where: ncode is the error code generated for which i1, i2, i3 are to be set (refer to Table 6-5).

i1 is the switch word setting for termination
=0: abort with a Q6 abort
=1: execution continues

i2 is the switch word setting for message printing and trace
=0: print
=1: suppress printing

i3 is the switch word setting for alternate error procedure
=0: use normal return
=1: use alternate error procedure

NOTE: If i3 = 1, i1 is ignored

Examples

CALL FXOPT(32,0,1,0)	Abort; no message printed; no alternate return
CALL FXOPT(32,1,0,0)	Continue; print message; no alternate return
CALL FXOPT(32,0,0,1)	Print message; go to alternate return if error occurs

Calling Sequence

```
CALL FXALT(SR)
```

where: SR is the alternate error procedure subroutine that is used as the transfer address for the error monitor.

FUNCTION subprograms and parameters are not allowed for SR. If the alternate procedure option for an error code is indicated but no call to FXALT has been made, a Q5 abort occurs if an error condition occurs. A RETURN statement in the alternate routine causes execution to be continued at the next executable statement following the statement that caused the error.

The alternate error procedure should not invoke the routine in which the error was found (i.e., the alternate error procedure for a formatted input/output statement cannot perform formatted input/output operations). The statement CALL FXALT($n) designates statement n in the calling program as the alternate error return. Statement n must be in the same program unit in which the CALL FXALT appears but does not have to be in the same program unit in which the error occurs.

NOTE: If the same error occurs in the alternate error routine, a loop results.

The standard setting of bits in the FXSW1 switch word groups controls termination. The execution results are indicated in the second column of Table 6-5. The settings in the second and third switch word groups (trace and alternate return) are initially zero.

Table 6-5. Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
0	A	Not used				
1	C	I**J	I=0,J=0	$0 \rightarrow$ QR	EXPONENTIATION ERROR 0**0	SET RESULT=0
2	C	I**J	I=0,J<0	$(2**35)-2 \rightarrow$ QR	EXPONENTIATION ERROR 0**(-J)	SET RESULT=2**35-2
3	C	{ DA**J A**J }	{ DA=0,J=0 A=0,J=0 }	$0 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
4	C	{ A**J DA**J }	{ A=0,J<0 DA=0,J<0 }	$10**38 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**(-J)	SET RESULT=10**38
5	C	B**C	B<0	$0 \rightarrow$ EAQ	EXPONENTIATION ERROR (-B)**C	SET RESULT=0
6	C	A**B	A=0,B=0	$0 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
7	C	A**C	A=0,C<0	$10**38 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**(-C)	SET RESULT=10**38
8	C	e**B	B>88.028	$10**38 \rightarrow$ EAQ	EXP(B),B GRT THAN 88.028 NOT ALLOWED	SET RESULT=10**38
9	C	LOG(A)	A=0	$-(10**38) \rightarrow$ EAQ	LOG(0) NOT ALLOWED	SET RESULT-(10**38)
10	C	LOG(B)	B>0	$0 \rightarrow$ EAQ	LOG(-B) NOT ALLOWED	SET RESULT=0.0
11	C	ARCTAN(A/B)	A=0,B=0	$0 \rightarrow$ EAQ	ATAN2(0,0) NOT ALLOWED	SET RESULT=0
12	C	{ SIN(A) COS(A) }	$\|A\|>2^{27}$	$0 \rightarrow$ EAQ	SIN OR COS ARG GRT TH 2**27 NOT ALLOWED	SET RESULT=0
13	C	$\sqrt{B}$	B<0	$\sqrt{B}=\sqrt{\|B\|}$	SQRT(-B) NOT ALLOWED	EVALUATE FOR +B
14	C	CA**K	CA=0,K=0	$0 \rightarrow$ AQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
15	C	CA**J	CA=0,J≤0	$10**38 \rightarrow$ AR $0 \rightarrow$ QR	EXPONENTIATION ERROR 0**(-J)	SET RESULT=(10**38,0.0)
16	C	DA**DB	DB≠0,DA < 0	$0 \rightarrow$ EAQ	EXPONENTIATION ERROR (-DA)**DB	SET RESULT=0
17	C	DA**DB	DA=0,DB=0	$0 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**0	SET RESULT=0

Table 6-5 (cont). Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
18	C	DA**DB	DA=0,DB < 0	10**38 → EAQ	EXPONENTIATION ERROR 0**(-DB)	SET RESULT=10**38
19	C	e**DA	DA > 88.028	10**38 → EAQ	EXP(B),B GRT 88.028, NOT ALLOWED	SET RESULT=10**38
20	C	LOG(DA)	DA=0	-(10**38) → EAQ	DLOG(0) NOT ALLOWED	SET RESULT=-(10**38)
21	C	LOG(DA)	DA < 0	0 → EAQ	DLOG(-B) NOT ALLOWED	SET RESULT=0
22	C	$\sqrt{DA}$	DA < 0	$\sqrt{DA}=\sqrt{\|DA\|}$	SQRT(-B) NOT ALLOWED	EVALUATE FOR +B
23	C	$\left\{\begin{matrix}\text{SIN DA}\\\text{COS DA}\end{matrix}\right\}$	$\|DA\|>2^{54}$	0 → EAQ	DSIN OR DCOS ARG GRT 2**54 NOT ALLOWED	SET RESULT=0
24	C	ARCTAN(DA/DB)	DA=0,DB=0	0 → EAQ	DATAN2(0,0) NOT ALLOWED	SET RESULT=0
25	C	CA/CB	CB=(0,0)	10**38 → AR 10**38 → QR	COMPLEX Z/0 NOT ALLOWED	SET RESULT=(10**38, 10**38)
26	C	e**CA	REAL CA>88.028	10**38 → AR 10**38 → QR	EXP(Z),REAL PART GRT 88.028 NOT ALLOWED	SET RESULT=(10**38, 10**38)
27	C	e**CA	$\|IMAG\ (CA)\|>2^{27}$	0 → AR 0 → QR	EXP(Z),IMAG PART GRT 2**27 NOT ALLOWED	SET RESULT=(0,0)
28	C	LOG(CA)	CA=(0,0)	-(10**38) → AR 0 → QR	CLOG(0) NOT ALLOWED	SET RESULT (-(10**38),0.0)
29	C	$\left\{\begin{matrix}\text{SIN(CA)}\\\text{COS(CA)}\end{matrix}\right\}$	$\|REAL(CA)\|>2^{27}$	0 → AQ	CSIN OR CCOS ARG WITH REAL PART GRT 2**27 NOT ALLOWED	SET RESULT=0
30	C	$\left\{\begin{matrix}\text{COS(CA)}\\\text{SIN(CA)}\end{matrix}\right\}$	IMAG(CA) > 88.028	10**38 → AR 10**38 → QR	CSIN OR CCOS ARG WITH IM PART GRT 88.028 NOT ALLOWED	SET RESULT=(10**38, 10**38)
31	C	BCD I/O	ILLEGAL FORMAT STATEMENT	-----	FORMAT AT LLLLLL,FIRST WORD HHHHHH IS ILLEGAL	TREAT AS END OF FORMAT
32	C	BCD I/O	ILLEGAL CHARACTER IN DATA OR BAD FORMAT	-----	ILLEGAL CHAR IN DATA OR BAD FORMAT	TREAT ILLEGAL CHAR AS ZERO
33	A	BCD I/O	ATTEMPT TO READ OUTPUT FILE	-----	READ AFTER WRITE IS ILLEGAL	FC = XX
34	A	BCD I/O	END-OF-FILE	-----	END OF FILE READING FILE CODE FC	OPTIONAL RETURN NOT REQUESTED

Table 6-5 (cont). Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
35	C	REWIND AND END FILE PROCESSOR	ILLEGAL REQUEST	-----	REQUEST TO XXXXXX ON FC WAS IGNORED	-----
36	C	FFFB	BACKSPACE ERROR	-----	TAPE POSITIONED AT 1ST FILE	BACKSPACE REQ. LARGER THAN FILE COUNT
37	A	FILE OPENING	FILE NOT DEFINED	-----	LOG. FILE CODE FC DOES NOT EXIST	NO OPTIONAL EXIT EXECUTION TERMINATED
38	A	FILE OPENING	NO SPACE FOR I/O BUFFERS	-----	INSUFFICIENT CORE AVAIL- ABLE FOR BUFFERS	NO OPTIONAL EXIT EXECUTION TERMINATED
39	A	BINARY I/O	ILLEGAL END- OF-FILE	-----	UNEXPECTED EOF	OR BAD FORMAT; FILE # XX
40	C	BINARY I/O	LIST EXCEEDS LOGICAL RECORD LENGTH	-----	LIST EXCEEDS LOGICAL RECORD LENGTH	STORE ZEROS IN REMAINING LIST ITEMS;FC
41	A	BINARY I/O	SYSOUT/FIXED LENGTH RECORDS	-----	SYSOUT OR FIXED LENGTH RECORDS MUST	BE SMALLER THAN BLOCK SIZE; FILE # XX
42	C	NAMELIST INPUT	ILLEGAL HEADING CARD	-----	ILLEGAL HEADING CARD BELOW	SCAN TERMINATED
43	C	NAMELIST INPUT	ILLEGAL VARIABLE NAME	-----	ILLEGAL VARIABLE NAME BELOW	SKIPPING TO NEXT VARIABLE NAME
44	C	NAMELIST INPUT	ILLEGAL SUBSCRIPT OR ARRAY SIZE EXCEEDED	-----	ILLEGAL SUBSCRIPT BELOW, OR DATA EXCEEDS VARIABLE	SKIPPING TO NEXT VARIABLE NAME
45	C	NAMELIST INPUT	ILLEGAL CHARACTER AFTER RIGHT PARENTHESIS	-----	ILLEGAL CHAR IN DATA BELOW	ASSUME COMMA PRECEDES CHAR
46	C	NAMELIST INPUT	ILLEGAL CHAR IN DATA	-----	ILLEGAL CHAR IN DATA BELOW	TREAT CHAR AS ZERO
47	A	BACKSPACE RECORD	FILE CANNOT BE BACKSPACED	-----	FILE CODE NN, BACKSPACE REFUSED	FILE IS SYSOUT OR IS NOT MAGNETIC TAPE, DISK OR DRUM
48	C	NAMELIST INPUT	ILLEGAL LOGICAL CONSTANT	-----	ILLEGAL LOGICAL CONSTANT APPEARS BELOW (OR AT END OF PRECEDING RECORD)	TREAT ILLEGAL LOGICAL CONSTANT AS F
49	A	BACKSPACE FILE	ERRONEOUS END-OF-FILE	-----	ERRONEOUS END OF FILE ON BACKSPACE	

Table 6-5 (cont). Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
50	C	BACKSPACE FILE	BLOCK COUNT OF ZERO	-----	BLOCK COUNT IN FCB EQUALS ZERO	
51	C	SENSE LIGHT SIMULATOR	INDEX NOT $0 \leq n \leq 35$	-----	REFERENCE TO NON-EXISTENT SENSE LIGHT	DECLARED OFF IF TESTING, IGNORED IF SETTING
52	C	NAMELIST INPUT	ILLEGAL HOLLERITH FIELD	-----	ILLEGAL HOLLERITH FIELD BELOW	SKIPPING TO NEXT VARIABLE NAME
53	C	SENSE SWITCH TEST	INDEX NOT $1 \leq n \leq 6$	-----	NON-EXISTENT SENSE SWITCH TESTED	SWITCH DECLARED OFF
54	A	FILE OPENING	ATTEMPT TO WRITE I*	-----	ILLEGAL WRITE REQUEST ON SYSIN1	NO OPTIONAL EXIT EXECUTION TERMINATED
55	A	FXEM	NAMELIST INPUT	-----	ILLEGAL COMPUTED GO TO	
56	A	FILE OPENING	ATTEMPT TO READ P*	-----	ILLEGAL READ REQUEST ON SYSOU1 OR SYSPP1	
57	C	BCD I/O	ILLEGAL CHAR FOR L CONVERSION	-----	ILLEGAL CHAR FOR L CONVERSION IN DATA BELOW	TREAT ILLEGAL CHARACTER AS SPACE
58	C	BACKSPACE RECORD	-----	-----	FILE NN IS CLOSED	BACKSPACE REFUSED
59	C	NAMELIST INPUT	EMPTY HOLLERITH FIELD	-----	EMPTY HOLLERITH FIELD	
60	C	I**J	I**J > 2**35 |I|>1,J > 35 J IS EVEN I< -1,J > 35, J IS ODD	(2**35)-2 → QR (2**35)-2 → QR -((2**35)-2) → QR	EXPONENT OVERFLOW	SET RESULT=± ((2**35)-2
61 62 63 64 65 66		RESERVED FOR USERS				
67	C	FAULT	EXPONENT UNDERFLOW	-----	EXPONENT UNDERFLOW	AT LOCATION XXXXXX
68	C	FAULT	INTEGER OVERFLOW	-----	OVERFLOW	AT LOCATION XXXXXX
69	C	FAULT	EXPONENT OVERFLOW	-----	EXPONENT OVERFLOW	AT LOCATION XXXXXX
70	C	FAULT	INTEGER DIVIDE BY ZERO	-----	DIVIDE CHECK	AT LOCATION XXXXXX

Table 6-5 (cont). Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
71	C	FAULT	FLOATING POINT DIVIDE BY ZERO	-----	DIVIDE CHECK	AT LOCATION XXXXXX
72	C	RANDOM I/O	LIST EXCEEDS LOGICAL RECORD LENGTH	-----	LIST EXCEEDS LOGICAL RECORD LENGTH	STORE ZEROS IN REMAINING LIST ITEMS FC # XX
73	A	RANDOM I/O	FILE NOT STANDARD SYSTEM FORMAT, ZERO BLOCK COUNT; BSN ERROR; ZERO RECORD COUNT	-----	FILE NOT STANDARD SYSTEM FORMAT FILE # FC	
74	A	RANDOM I/O	NO DEVICE FOR FILE	-----	LOGICAL FILE CODE FC DOES NOT EXIST	NO OPTIONAL EXIT EXECUTION TERMINATED
75	A	RANDOM I/O	BAD RECORD REFERENCE	-----	ZERO OR NEGATIVE REC #	FC # XX
76	A	RANDOM I/O	RECORD SIZE NOT SPECIFIED IN FCB. GIVE VIA $ FFILE CARD OR CALL RANSIZ (FC,SIZE)	- -	REC SIZE NOT GIVEN FOR RANDOM FILE	FC # XX
77	A	RANDOM I/O	RANDOM I/O TO LINKED FILE ILLEGAL	-----	RANDOM I/O TO LINKED FILE ILLEGAL	FC # XX
78	A	RANDOM I/O	THE RECORD NO. GIVEN IN THE RANDOM READ OR WRITE STATEMENT IS OUTSIDE THE FILE LIMITS	---	REC # OUT-OF-BOUNDS-	FC # XX
79	A	RANDOM I/O	LIST EXCEEDS DECLARED RECORD LENGTH	-----	LIST EXCEEDS DECLARED RECORD LENGTH	FC # XX
80	A	RANDOM I/O	FILE IS NOT LARGE ENOUGH TO CONTAIN RECORD	-----	FILE SPACE EXHAUSTED-	FC # XX
81	C	FORMAT I/O ENCODE/DECODE	LINE EXCEEDS SIZE OF RECEIVING FIELD	-----	LINE EXCEEDS SIZE OF RECEIVING FIELD	TREAT AS END OF FORMAT
82	C	FORMAT I/O ENCODE/DECODE	FIRST NON-BLANK CHAR-ACTER IS NOT (	-----	FIRST NON-BLANK CHAR-ACTER IS NOT (	TREAT AS END OF FORMAT
83	C	ARCSINE	|ARG| > 1.0		|ARG|>1.0	EVALUATE FOR ARG=1.0
84	C	FORMAT I/O ENCODE/DECODE	|INTEGER|>2**35-1	-----	|INTEGER|>2**35-1	LIMIT TO 2**35-1

Table 6-5 (cont). Error Codes and Meanings

ERROR CODE	DEFAULT PROCEDURE ABORT/ CONTINUE	FUNCTION	ERROR	EXCEPTION RETURN	MESSAGE LINE 1	MESSAGE LINE 2
85	C	I/O	"GFRC" ERROR	-----	"GFRC" ERROR CODE XXX	FC #XX
86	A	FORMAT I/O ENCODE/DECODE	ENCODE/DECODE-I/O MAY NOT BE USED RECURSIVELY	-----	ENCODE/DECODE- I/O MAY	NOT BE USED RECURSIVELY
87	C	I/O	SPACE/CORE OBTAINED	-----	SPACE/CORE OBTAINED FOR	LOG. FILE CODE #XX
88	C	CALLSS	END OF STRING CHARACTER MISSING	-----		
89	C	EXP DEXP	UNDERFLOW	-----	EXP(TOO LARGE A NEGATIVE NUMBER)	SET RESULT =0.0
90	C	TAN DTAN	ARG TOO LARGE	-----	LARGE ARG(71E4) TO TAN	MAY CAUSE LOSS OF PRECISION
91	C	ACOSH DACOSH	ILLEGAL ARG	-----	ACOSH OF NUMBER .LT. 1.0 NOT ALLOWED	SET RESULT TO 0.0
92	C	ATANH	ILLEGAL ARG	-----	X .GE. 1.0 TO ATANH(X)	SET RESULT TO + OR -10**38
93	C	BACKSPACE FILE	BAD TAPE STATUS	-----	FILE CODE NN	BAD STATUS ON TAPE
94	C	BCD I/O	ILLEGAL FORMAT	-----	FORMAT AT XXXXXX HAS ILLEGAL	CONVERSION; IGNORE
95	C	INTERNAL CONVERSION	MODULE NOT SUPPORTED	-----	INTERNAL CONVERSION ROUTINE FINC IS NO LONGER SUPPORTED	USE ENCODE/DECODE
96-143		NOT PRESENTLY USED				

NOTATION: I,J,K are integers
A,B,C, are real numbers
DA,DB,DC are double-precision numbers
CA,CB,CC where CA=X,Y are complex numbers

NOTE: If it is desirable to set up additional error messages, begin with error code 143, and use the error codes in descending order. (i.e., 143, 142, 141, etc.)

LINK, LLINK

The LINK subroutine enables the programmer to call program overlays. The following call is used to load a link and transfer control to it without returning to the calling program/overlay.

Calling Sequence

CALL LINK(name)

where: name designates the variable name of the link as it appears on the $ LINK control card. (See the General Loader manual for "Link/Overlay Processing".) Name may be a variable which currently has a character type value, or it may be a character constant (e.g., "LINK1"). The link name must be 1-6 characters if using the BCD option, or it must be 5-8 characters if using the ASCII option. Explicit trailing blanks are included in the character count.

The following statement is used to load a link and return to the next sequential statement of the calling routine.

CALL LLINK(name)

NOTE: Due to TSS FORTRAN RUN subsystem limitations, it is necessary to force the loading of input-output library routines with the main link in a time sharing loadable H* file. This requires the presence of a PRINT statement or another form of input-output in the main program.

MEMSIZ

This subroutine provides the user with the means of obtaining the amount of memory allocated for execution of the program.

Calling Sequence

```
CALL MEMSIZ(j)
```

where: *j* an integer variable, is returned as the number of 1024-word blocks currently allocated for the program in execution.

NASTRK

This subroutine may be called to avoid filling an output field with asterisks when a formatted output value exceeds the field width specified. The most significant part of the number is truncated to fit the field (see subroutine YASTRK).

Calling Sequence

```
CALL NASTRK
```

OVERFL (see DVCHK)

PDUMP,PDUMPA (see DUMP)

PTIME

This subroutine provides the means of obtaining processor time used.

Calling Sequence

```
CALL PTIME(a)
```

where: _a_ a real variable, is the value returned with the processor time used in hours.

NOTE: This feature can also be used as a function. The value returned will be a cumulative time for this job or, if under time sharing, it will be cumulative since log-on for the current user.

RANSIZ

This subroutine is used to specify the record size for a random binary file. Normal return is to the next executable statement of the calling program. If the record size for a given random file is not provided at load time via the $ FFILE card, a call to this routine before opening the file is mandatory.

Calling Sequence

CALL RANSIZ (u,n[,m])

where: u is the logical file designator of type integer, and can be any legal arithmetic expression.

n is the record size of type integer and can be any legal arithmetic expression.

m is a file format indicator of type integer and can be any legal arithmetic expression.
=0 or null: standard system format
≠0: block and control records are not to be processed.

NOTE: A call to RANSIZ can also be used to override a $FFILE control card size specification. However, a call to RANSIZ is the preferred method of specification because it works in both batch and time sharing mode.

If m ≠ 0, all the designated file space is available for data. The records are not blocked, can begin anywhere in a sector, and may span sector boundaries.

SETBUF

This subroutine is used to assign space in storage for use as an input/output buffer(s) but does not change the buffer size. Because the size of the buffer(s) must be one greater than the actual record size, the system standard buffer size is 321 words. Normal return is to the next executable statement in the calling program.

Calling Sequence

```
CALL SETBUF(i,a)

CALL SETBUF(i,a [,b])
```

where: i, the logical file designator, is an integer variable or a constant.

a is the array name of the first buffer.

b is the array name of the second buffer for file i, if required.

SETFCB

This subroutine is used to define a file control block (FCB) for use by the I/O subprograms. Normal return is to the next executable statement of the calling program unless one of the following error conditions occur

1. Abort with a Q2 abort code if there is no logical file table.

2. Abort with a Q1 abort code if there is no space available in the logical file table for inserting a specified file control block.

Calling Sequence

```
CALL SETFCB(a,i,j[,...])
```

where:	a	is the location of LOCSYM in the user created file control block.
	i,j	are the logical file designators (integer variables or constants) that refer to the file control block.

SETLGT

This subroutine is used to define a logical unit table for use by the I/O library subprograms. If GLOAD has not set up the LGU, this subroutine must be called before any input/output is requested (e.g., $ OPTION NOFCB). It is called when the logical file table generated by the General Loader is to be replaced and the table is to be placed in the user area of memory. The NOFCB option must be specified in the $ OPTION control card.

NOTE: Normal return is to the next executable statement of the calling program.

Calling Sequence

```
CALL SETLGT(a,i)
```

where: _a_ is the array name of the logical unit table to be used.

_i_ is an integer variable or constant representing the number of words in a.

SLITE,SLITET

This subroutine subprogram simulates the setting and testing of sense lights. Normal return is to the next executable statement in the calling program.

Calling Sequence

CALL SLITE(0) to clear sense lights 1-35

CALL SLITE(i) to turn on sense light i ($1 \leq i \leq 35$)

CALL SLITET(i,j) to test and turn off sense light i, if it is currently set

where: i is an integer variable or constant.

j is an integer variable that cannot be the induction variable of a currently active DO loop.
=1: i is ON
=2: i is OFF

SORT

This subroutine is used to sort positive integer or character arrays in ascending order. All comparisons are based on 36-bit integer magnitudes. This means that data cannot be sorted into an integer algebraic sequence (refer to Appendix A for the proper collating sequence).

NOTE: Floating-point data cannot be sorted reliably.

ISORT

This subroutine is used to sort integer arrays in ascending order. The data is sorted into an integer algebraic sequence where negative values are allowed.

Calling Sequence

```
CALL SORT (array,nrec,lrs,key[,...])
CALL ISORT (array,nrec,lrs,key [,...] )
```

where: array is the name of the array to be sorted.

nrec is an integer variable or constant and is the number of elements, or logical records, in the array.

lrs is an integer variable or constant describing the logical record size, in words, of the records in the array to be sorted.

Example

```
DIMENSION I(5,20)          defines 20 integer records whose
CALL SORT(I,20,5,...)      logical record size = 5 elements * 1
                           word per element = 5
     or

CHARACTER *12 I(5,20)      in ASCII; each element is 12
CALL SORT(I,20,15,...)     characters = 3 words
                           15 words per logical record
                           5 elements * 3 words per element = 15
     or

CHARACTER *12 I(5,20)      in BCD, each element is 12
CALL SORT (I,20,10,...)    characters = 2 words
                           10 words per logical record
                           5 elements * 2 words per element = 10
```

key is the relative word number of the ith sort key in each logical record and must be in the range $0 \leq$ key $\leq$ lrs. Record comparisons are made starting with key_1, and either progress through key_n, or until a non-equal comparison is made. Any number of sort keys may be specified; however, at least one must always be specified. If key has a value of zero, the sort will occur on the first word of each array element.

The following example illustrates a two-dimensional array sort.

```
0010  CHARACTER*5 ARR(3,5)
0020  PRINT,"INPUT DATA"
C READ DATA ONE COLUMN AT A TIME
0030  READ(5,10)ARR
0040  10 FORMAT(3A5)
C THE ARRAY CONTAINS 5 LOGICAL RECORDS CONSISTING OF 3 WORDS EACH
C THAT IS, 5 COLUMNS AND 3 ROWS
0050  CALL SORT(ARR,5,3,0,1,2)
0060  PRINT,"SORTED DATA BY COLUMN"
0070  WRITE(6,10)ARR
0080  STOP;END

ready

INPUT DATA
=ELK FAST 1
=COW SLOW 2
=DOG FAST 3
=CAT FAST 4
=ELK FAST 5
SORTED DATA BY COLUMN
CAT FAST 4
COW SLOW 2
DOG FAST 3
ELK FAST 1
ELK FAST 5
```

SORTD

This subroutine is used to sort positive integer or character arrays in a descending order. All comparisons are based on 36-bit integer magnitudes. This means that data cannot be sorted into an integer algebraic sequence (refer to Appendix A for the proper collating sequence).

NOTE: Floating-point data (real values) cannot be sorted reliably.

ISORTD

This subroutine is used to sort integer arrays in descending order. The data is sorted into an integer algebraic sequence where negative values are allowed.

Calling Sequence

```
CALL SORTD (array,nrec,lrs,key[,...])
CALL ISORTD (array,nrec,lrs,key [,...] )
```

where: array is the name of the array to be sorted.

nrec is an integer variable or constant that specifies the number of items or logical records in the array.

lrs is an integer variable constant describing the logical record size, in words, of the records in the array to be sorted.

Example

```
DIMENSION I(5,20)          defines 20 integer records whose
CALL SORTD(I,20,5,...)     logical record size = 5 elements * 1
                           word per element = 5
     or

CHARACTER *12 I(5,20)      in ASCII;  each  element  is  12
CALL SORTD(I,20,15,...)    characters = 3 words
                           15 words per logical record
                           5 elements * 3 words per element = 15
     or

CHARACTER *12 I(5,20)      in   BCD,  each  element  is   12
CALL SORTD(I,20,10,...)    characters = 2 words
                           10 words per logical record
                           5 elements * 2 words per element = 10
```

<u>key</u> is an integer variable or constant and is the word number of the ith sort key in each logical record and must be in the range $0 \leq key \leq lrs$. Record comparisons are made starting with key_1 and either progress through to key_n , or until a non-equal comparison is made. Any number of sort keys may be specified; however, at least one must always be specified.

```
0005  INTEGER ARRAY(10)
0010  DO 10 I=1,10
0020  10 ARRAY(I)=RANDT(20.0)
0040  WRITE(6,200)ARRAY
0042  CALL SORTD(ARRAY,10,1,0)
0043  WRITE(6,200)ARRAY
0050  200 FORMAT(10(2X,I7))
0060  STOP;END

ready

*RUN
 9     3     0     4     17     11     19     17     12     8
19    17    17    12     11      9      8      4      3     0
```

SSWTCH

This subroutine subprogram tests the GCOS switch word for the status of a sense switch. Normal return is to the next executable statement in the calling program.

Calling Sequence

CALL SSWTCH(i,j) to test sense switch i

where: i is an integer variable or constant that must be from 1 to 6.

j is an integer variable that cannot be the induction variable of a currently active DO loop.
=1: i is ON
=2: i is OFF

Bits 6-11 of the Program Switch Word (described in the *General Comprehensive Operating Supervisor* manual), correspond to sense switches 1-6.

TERMNO

This subroutine is used as a means of obtaining station code.

Calling Sequence

CALL TERMNO (a)

where: a is a character variable. The value returned is a two-character station code.

In batch, the call returns two blank characters for the station code.

TERMTM

This subroutine is used to obtain the elapsed time since log-on. The call is applicable only for time sharing activities; it is ignored in the batch mode.

Calling Sequence

```
CALL TERMTM (a)
```

where: _a_ a real variable, is the value returned for the hours since log-on.

TRACE

This subroutine is called from a FORTRAN object program in the time sharing mode. It is useful in tracing and debugging an object module (refer to Debug and Trace Routines manual).

USRCOD

This subroutine is used as a means of obtaining user identification. The call is applicable only for time sharing activities; it is ignored in the batch mode.

Calling Sequence

```
     CALL USRCOD (s)
```

where: s a character variable, is returned as the value of the 12-character user identification.

YASTRK

This subroutine may be called to override the affect of the NASTRK subroutine and to re-establish the default action of filling an output field with asterisks when a formatted output value exceeds the field width specified (refer to subroutine NASTRK).

Calling Sequence

```
     CALL YASTRK
```

APPENDIX A

ASCII/BCD CHARACTER SET

ASCII CHAR	Octal Collating Sequence	BCD CHAR	Octal	MODEL 33/35 KEY	HOLLERITH CARD Punch	MEANING
NULL	000	---	---	'CS'P	---	Null or time fill char
SOH	001	---	---	'C'A	---	Start of heading
STX	002	---	---	'C'B	---	Start of text
ETX	003	---	---	'C'C (EOM)	---	End of text
EOT	004	---	---	'C'D (EOT)	---	End of transmission
ENQ	005	---	---	'C'E (WRU)	---	Enquiry (who are you)
ACK	006	---	---	'C'F (RU)	---	Acknowledge
BEL	007	---	---	'C'G (BELL)	---	Bell
BS	010	---	---	'C'H	---	Backspace
HT	011	---	---	'C'I (TAB)	---	Horizontal tabulation
LF	012	---	---	LINE FEED	---	Line Feed (New Line)
VT	013	---	---	'C'K (VT)	---	Vertical Tabulation
FF	014	---	---	'C'L (FORM)	---	Form Feed
CR	015	---	---	RETURN	---	Carriage Return
SO	016	---	---	'C'N	---	Shift Out
SI	017	---	---	'C'Ø	---	Shift In
DLE	020	---	---	'C'P	---	Data Link Escape
DC1	021	---	---	'C'Q (X-ON)	---	Device Control 1
DC2	022	---	---	'C'R (TAPE)	---	Device Control 2
DC3	023	---	---	'C'S (X-OFF)	---	Device Control 3
DC4	024	---	---	'C'T (TAPE)	---	Device Control 4
NAK	025	---	---	'C'U	---	Negative Acknowledge
SYN	026	---	---	'C'V	---	Synchronous Idle
ETB	027	---	---	'C'W	---	End of Transmission Blocks
CAN	030	---	---	'C'X	---	Cancel
EM	031	---	---	'C'Y	---	End of Medium
SS	032	---	---	'C'Z	---	Special Sequence
ESC	033	---	---	'CS'K	---	Escape
FS	034	---	---	'CS'L	---	File Separator
GS	035	---	---	'CS'M	---	Group Separator
RS	036	---	---	'CS'N	---	Record Separator
US	037	---	---	'CS'Ø	---	Unit Separator
SP	040	blank	20	SPACE BAR	blank	Space
!	041	!	77	'S'1	0-7-8	Exclamation Point
"	042	"	76	'S'2	0-6-8	Quotation Mark
#	043	#	13	'S'3	3-8	Number Sign
$	044	$	53	'S'4	11-3-8	Currency Symbol
%	045	%	74	'S'5	0-4-8	Percent
	046		32	'S'6	12	Ampersand
'	047	'	57	'S'7	11-7-8	Apostrophe
(	050	(	35	'S'8	12-5-8	Opening Parenthesis
)	051	)	55	'S'9	11-5-8	Closing Parenthesis
*	052	*	54	'S':	11-4-8	Asterisk
+	053	+	60	'S';	12-0	Plus
,	054	,	73	,	0-3-8	Comma
-	055	-	52	-	11	Hyphen or Minus
.	056	.	33	.	12-3-8	Period
/	057	/	61	/	0-1	Slant

ASCII CHAR	Octal Collating Sequence	BCD CHAR	Octal	MODEL 33/35 KEY	HOLLERITH CARD Punch	MEANING
0	060	0	00	0	0	Zero
1	061	1	01	1	1	One
2	062	2	02	2	2	Two
3	063	3	03	3	3	Three
4	064	4	04	4	4	Four
5	065	5	05	5	5	Five
6	066	6	06	6	6	Six
7	067	7	07	7	7	Seven
8	070	8	10	8	8	Eight
9	071	9	11	9	9	Nine
:	072	:	15	:	5-8	Colon
;	073	;	56	;	11-6-8	Semicolon
<	074	<	36	'S',	12-6-8	Less Than
=	075	=	75	'S'-	0-5-8	Equal
>	076	>	16	'S'.	6-8	Greater Than
?	077	?	17	'S'/	7-8	Question Mark
@	100	@	14	'S'P	4-8	Commercial At
A	101	A	21	A	12-1	Uppercase Letter
B	102	B	22	B	12-2	Uppercase Letter
C	103	C	23	C	12-3	Uppercase Letter
D	104	D	24	D	12-4	Uppercase Letter
E	105	E	25	E	12-5	Uppercase Letter
F	106	F	26	F	12-6	Uppercase Letter
G	107	G	27	G	12-7	Uppercase Letter
H	110	H	30	H	12-8	Uppercase Letter
I	111	I	31	I	12-9	Uppercase Letter
J	112	J	41	J	11-1	Uppercase Letter
K	113	K	42	K	11-2	Uppercase Letter
L	114	L	43	L	11-3	Uppercase Letter
M	115	M	44	M	11-4	Uppercase Letter
N	116	N	45	N	11-5	Uppercase Letter
O	117	Ø	46	Ø	11-6	Uppercase Letter
P	120	P	47	P	11-7	Uppercase Letter
Q	121	Q	50	Q	11-8	Uppercase Letter
R	122	R	51	R	11-9	Uppercase Letter
S	123	S	62	S	0-2	Uppercase Letter
T	124	T	63	T	0-3	Uppercase Letter
U	125	U	64	U	0-4	Uppercase Letter
V	126	V	65	V	0-5	Uppercase Letter
W	127	W	66	W	0-6	Uppercase Letter
X	130	X	67	X	0-7	Uppercase Letter
Y	131	Y	70	Y	0-8	Uppercase Letter
Z	132	Z	71	Z	0-9	Uppercase Letter
[	133	[	12	'S'K	2-8	Opening Bracket
\	134	\	37	'S'L	12-7-8	Reverse Slant
]	135	]	34	'S'M	12-4-8	Closing Bracket
^	136	^	40	'S'N	11-0	Circumflex
___	137	___	72	'S'Ø	0-2-8	Underline
`	140	---	---	---	---	Grave Accent
a	141	---	---	---	---	Lowercase Letter
b	142	---	---	---	---	Lowercase Letter
c	143	---	---	---	---	Lowercase Letter
d	144	---	---	---	---	Lowercase Letter
e	145	---	---	---	---	Lowercase Letter
f	146	---	---	---	---	Lowercase Letter
g	147	---	---	---	---	Lowercase Letter
h	150	---	---	---	---	Lowercase Letter
i	151	---	---	---	---	Lowercase Letter
j	152	---	---	---	---	Lowercase Letter
k	153	---	---	---	---	Lowercase Letter
l	154	---	---	---	---	Lowercase Letter

ASCII CHAR	Octal Collating Sequence	BCD CHAR	Octal	MODEL 33/35 KEY	HOLLERITH CARD Punch	MEANING
m	155	---	---	---	---	Lowercase Letter
n	156	---	---	---	---	Lowercase Letter
o	157	---	---	---	---	Lowercase Letter
p	160	---	---	---	---	Lowercase Letter
q	161	---	---	---	---	Lowercase Letter
r	162	---	---	---	---	Lowercase Letter
s	163	---	---	---	---	Lowercase Letter
t	164	---	---	---	---	Lowercase Letter
u	165	---	---	---	---	Lowercase Letter
v	166	---	---	---	---	Lowercase Letter
w	167	---	---	---	---	Lowercase Letter
x	170	---	---	---	---	Lowercase Letter
y	171	---	---	---	---	Lowercase Letter
z	172	---	---	---	---	Lowercase Letter
{	173	---	---	---	---	Opening Brace
|	174	---	---	---	---	Vertical Line
}	175	---	---	---	---	Closing Brace
~	176	---	---	---	---	Tilde
DEL	177	---	---	RUBOUT	12-7-9	Delete

Legend:

'C' = CTRL key
'CS' = CTRL and SHIFT keys
'S' = SHIFT key

APPENDIX B

TIME SHARING SYSTEM DESCRIPTION

*

The standard means of communication with the GCOS Time Sharing System (TSS) is via a CRT display terminal, a keyboard/printer terminal, a paper-tape terminal unit for input/output, or any combination. In any case, the information transmitted to and from the system is displayed on the terminal/printer. Keyboard input is used for purposes of description; instructions for the use of paper tape are given under "Paper Tape Input" in this section.

The time sharing system is utilized by means of a command language which is distinct from any of the specialized programming languages that are recognized by the individual time sharing compilers/processors (e.g., the time sharing FORTRAN language). The command language is basically the same for any component of the time sharing system (i.e., FORTRAN, BASIC, Text Editor, etc.). A few of the commands pertain to only one or another of the component time sharing systems, but the majority of them are common to all component systems.

The valid time sharing system commands relate to the generation, modification, and disposition of program and data files, as well as program compilation and execution requests. The complete time sharing command language is described in the _TSS General Information_ manual. However, the RUN command for the YFORTRAN and FORTRAN Time Sharing Systems is described in this appendix.

Once communication with the system has been established, any question or request from the system must be answered within ten minutes, with the exception of the initial requests for user identification (user-ID) and sign-on password, which must be given within one minute. When these time limits are exceeded, the terminal is disconnected.

*

Log-on Procedure

Communication with the time sharing system is initialized through the following steps:

- Activate the terminal unit
- Dial the site-designated phone number for the time sharing center
- Connect the receiver to the terminal coupler after a high-pitched tone is heard; if a busy signal is heard, hang up and try later
- Press the carriage return key
- Begin the log-on procedure

NOTE: A carriage return must be given following any complete response, command, or line of information typed by the user. (In the examples shown throughout this appendix, the user's response is underlined for illustration, and a carriage return terminating each separate response is understood.

Once the terminal has been connected to the computer, the time sharing system begins the log-on procedure by transmitting the message:

```
HIS SERIES 60 ON(date)AT(time)CHANNEL(nnnn)
```

where: time is given in hours and thousandths of hours (hh.hhh)

nnnn is the logical identifier of the line to which the user is connected.

Following the message, the system asks for the user's identification:

```
USER ID -
```

The user identification (user-ID) that has been assigned by the time sharing installation management must be typed on the same line. This user-ID uniquely identifies a particular user already known to the system for the purpose of locating user programs and files, and initiates accounting for usage of the time sharing resources allocated.

Example

```
USER ID - J.P.JONES
```

After the proper response, the system asks for the sign-on password that was assigned with the user-ID.

```
PASSWORD
XBGBEEGHXXXE
```

The password must be typed directly on the "strikeover" mask provided below the PASSWORD request. The password is used by the system as a check on the legitimacy of the identified user. The "strikeover" mask ensures that the password, when typed, cannot be read by another person. (In the event that either the user-ID or password is given twice incorrectly, the user's terminal is immediately disconnected from the system.) At this point, if the accumulated charges for the user's past time sharing usage equals or slightly exceeds 100 percent of current resource allocation, the user receives a warning message. If accumulated charges exceeds 110 percent of current resources, the message

```
RESOURCES EXHAUSTED - CANNOT ACCEPT YOU
```

is printed and the terminal is immediately disconnected. The following information message may be printed if more than 87% of the user's file space quota has been used.

```
n BLOCKS FILE SPACE AVAILABLE
```

NOTE: This condition does not affect the log-on procedure.

Assuming that the user-ID and password are legitimate, and resources have not been over-extended, an asterisk is issued indicating readiness to accept commands and/or build files. The RUN and RESEQUENCE commands are unacceptable at this point since it is not known what type of source is to be acted upon. For example, does RUN mean to compile a BASIC or FORTRAN program, or does it mean a batch job is to be submitted for processing? The user has two alternatives available to deal with this ambiquity:

1. The mode is established by simply entering the desired system selection (i.e., FORTRAN or YFORTRAN, which can be abbreviated as FORT or YFORT) accompanied by an OLD or NEW request.

   Once the system selection has been made, the system remains in effect until explicitly changed (or cancelled by means of the break key). The RUN command can be used once the mode is established.

2. The BRN, FRN, and JRN commands can be issued independent of previous system selection (if any) and imply RUN for BASIC, FORTRAN, and CARDIN, respectively. Note that JRN cannot be used as an execution command for a CARDIN program unless that program follows the CONVERT subsystem syntax.

The JRN command is not identical to the CARDIN RUN command. Refer to the TSS Terminal/Batch Interface manual for details concerning the use of JRN in conjunction with the CONVERT subsystem. *

The following is an example of a complete log-on procedure, up to the point where the user is ready to begin building a file or exercising commands:

```
HIS SERIES 6000 ON 05/26/77 AT 14.568 CHANNEL 0012

USER ID -J.P.JONES
PASSWORD
[illegible]
*            - (begin entering input on this line)
```

Program Statement Input

The system is currently in build-mode (as indicated by the initial asterisk) and is ready to accept FORTRAN program statement input or control commands. All lines of input other than control commands are accumulated on the user's current file as they are entered into the system.

Following each line of noncommand-language input and the terminating carriage return, the system supplies another initial asterisk when the carriage is returned, to indicate the system is ready to accept more input.

A line of FORTRAN input can contain:

1. One or more FORTRAN statements
2. A partial statement
3. A continuation of a statement left incomplete in the preceding line of input
4. A comment
5. A combination of 3 and 1, or 3 and 2, in that order
6. A combination of 1 and 2

A line input must begin with a line-sequence number from one to eight numeric characters. The line-sequence number enables the programmer to correct and modify the source program. (Hereinafter, the line-sequence number is referred to simply as the "line number".)

NOTE: A line number is distinct from a statement number in that a statement number is a part of the FORTRAN language statement itself.

The line number is always terminated with a single control character that can be a blank, an ampersand, a number sign, an asterisk, or the letter C. This control character merely serves to indicate what type of information follows (i.e., new statement, continuation, or comment) and is not compiled as part of the program. The semicolon can be used to indicate the end of one complete FORTRAN statement and the beginning of another on the same line of input. A carriage return must be used to terminate a complete line of input. This line format is suitable for direct processing by the FORTRAN compiler with the options NFORM and LNO.

The general format of a line of FORTRAN input is

```
nnnnnnnncstatement or continuation ;statement...;statement

                            or

nnnnnnnnc comment
```

where: nnn...n is a numeric line number, the magnitude of which is less than 2^{18} (262,144)

c is a control character that can be a blank, an ampersand, an asterisk, a number sign, or the letter C, and must immediately follow the last digit of the line number.

CONTROL CHARACTER

The control character identifies the type of information that follows it.

b̸ (blank)	- If the character position immediately following the last digit of the line number contains a blank, and the next nonblank character is not an ampersand, then that nonblank character is assumed to begin a new FORTRAN statement. In this case, the next nonblank character may begin a FORTRAN statement number (i.e., mm...m statement-text).
& (ampersand)	- If an ampersand is the first nonblank character following the line number, the next significant character is assumed to be a continuation of the previous statement in the previous line of input. (A blank character is significant only as a continuation of the character string from a preceding line.) The effect of "&" is to suppress the previous carriage return as an end-of-statement indicator.
* (asterisk) or C	- If the line number is terminated with an asterisk or the letter C, the following information is assumed to be a comment. The comment itself is terminated by a carriage return.
# (pound sign)	- If a numeric character is desired in column 1 of the card image and line numbers exist in the source file, a pound sign (#) immediately following the line number causes the character following it to be placed in column 1 .

A semicolon within a noncomment line indicates both the end of the preceding statement and the beginning of a new statement. The new statement can include the FORTRAN statement number, mm...m.

The format of a statement that follows a blank control character, is

...nnb̸ b̸...b̸ mm...m FORTRAN-language-text

(The statement format portion is underlined.)

where: b̸...b̸ are optional blanks

mm...m is an optional numeric statement number where mm < 99999

BLANKS (OR SPACING) WITHIN A LINE OF INPUT

Initial, embedded, or trailing blanks in a line of input have no significance in its interpretation; however, blanks are illegal within the line number and the nonnumeric character immediately following the line number is interpreted as a control character. Thus, spacing can be used quite freely within a line of input for legibility. Blanks within character constants and nH fields (i.e., alphanumeric information are meaningful and are retained in the object program coding.)

NOTE: The line/statement format is completely free-form, or position independent with the exception of the control character.

To this point, the discussion of line format has been oriented to the NFORM format described earlier in this document. This is generally the most convenient form to use in time sharing, although it is not mandatory. The source file can be built using the Text Editor and can be used without line numbers through the NLNO option. The source program can be in "fixed" format (i.e., without line numbers) through the FORM option. The full spectrum of line formats and source file recording modes is available to the time sharing user.

Source Program Modification

Keyboard input is sent to the computer and written onto the user's current file in units of complete lines. A line of terminal input is terminated by a carriage return and no part of the line is transmitted to the system until that carriage return is given. Therefore, corrections or modifications can be done at the terminal at two distinct levels:

1. Correction of a line-in-progress (i.e., a partial line not yet terminated).
2. Correction or modification of the source program (i.e., the contents of the current source file) by the replacement or deletion of current lines, or the insertion of new lines.

The correction of a typing error that is detected before the line is terminated can be done in one of two ways:

- Delete one or more characters from the end of the partial line
- Cancel the incomplete line and begin again

NOTE: Use of the delete control character deletes the character preceding the deletion character. (The delete control character used is dependent upon the make of terminal at the site.)

Example

If # is the deletion character,

JONS#

deletes S

JONS DAVEY#######

deletes S DAVEY

Correction or modification of the current source file is done on the basis of line numbers and proceeds accordingly.

Example

The source file contains

```
100    READ(5,16)HRS,RATE,NO
200    WRITE(6,16)HRS,RATE,NO
300 16 FORMAT(F3.2,F4.2,I6)
```

1. Replacement. A numbered line replaces any identically numbered line that was previously typed or contained on the current file.

   Example

   ```
   200    WRITE(6,12)PAY
   ```

   replaces the current line numbered 200.

2. Deletion. A "line" consisting of only a line number (i.e., 100) causes the deletion of any identically numbered line that was previously typed or contained on the current file.

   Example

   ```
   100
   ```

   deletes line 100 from the source file.

3. Insertion. A line with a line-number value that falls between the line-number values of two pre-existing lines is inserted in the file between those two lines. If the line number is less than the first line number, it is inserted at the beginning of the file; if greater than the largest line number, it is inserted at the end of the file.

   Example

   ```
   250 12 FORMAT(//16HPAY IS EQUAL TO ,F6.2)
   ```

   is inserted above line 300.

   The new source file now contains

   ```
   200    WRITE(6,12)PAY
   250 12 FORMAT(//16HPAY IS EQUAL TO ,F6.2)
   300 16 FORMAT(F3.2,F4.2,I6)
   ```

Input Error Recovery

The decimal input/output routine permits the time sharing user (BCD or ASCII) to correct a string of characters in an executing FORTRAN program that was entered from a terminal when a character is illegal for the current format conversion (e.g., a decimal point is illegal in an "I" field). When the current input line is printed on the terminal with a pointer to the illegal character, the correction can be made, and the input/output routine resumes with the new string. If the response is a carriage return, an error message is printed.

At any point in the process of entering file building input in line-numbered subsystems, the LIST command may be given, which results in a clean, up-to-date copy of the current file. In this way, the results of any previous corrections or modifications can be verified visually. (The several forms of the LIST command are described in detail in the *TSS General Information* manual). Following the command "OLD *filename*", the LIST command can be used initially to inspect the contents of the current source file (i.e., the "old" program).

Automatic Terminal Disconnections

Once communication with the Time Sharing System has been established, any question or request must be answered within ten minutes. If these time limits are exceeded, the terminal is disconnected.

Log-Off Procedure

To terminate the current session with the Time Sharing System and disconnect the terminal, the

```
*BYE

 or

*LOGOFF
```

command may be given.

A report of the user's time sharing usage charges is given, as illustrated below, and the terminal is disconnected:

```
**COST:  $    0.17 TO DATE:  $  206.11=21%
**ON AT 15.000 - OFF AT 15.016 ON  07/19/78
```

If the BYE command is used, prior to the issuance of the usage charges, the AFT is scanned, and the user is queried as to the disposition of any temporary files.

To terminate the current session without disconnecting the terminal, the command NEWUSER may be given in place of BYE. This procedure allows another user to log-on immediately, or it can be used to change the charge number without going through the log-off/log-on procedure. The current log-off report is then printed and a new log-on sequence is initiated.

CAUTION: Failure to follow log-off procedures as described above may result in unpredictable problems (lines or files remaining busy, etc.). Certain data sets do not automatically disconnect after log-off from the terminal. In such cases, it is necessary to manually disconnect the data set by lifting the handset, pressing the talk button, and hanging up the handset when the dial tone is heard.

I-D-S/II in a FORTRAN Time Sharing Environment

The use of I-D-S/II in the FORTRAN time sharing environment requires the ability to specify FORTRAN source files, I-D-S/II control files, and I-D-S/II data base area and key files as well as the desired options from the terminal. The YFORTRAN and FORTRAN time sharing systems provide this capability.

Files Required by I-D-S/II

I-D-S/II requires control files and data base area files. Data base key files and data base procedure files may also be required. The control files required are

- Schema File - the schema file, a random file produced by the schema translation, is the "1*" file unless it has been renamed in the Device Media Control Language (DMCL). It has the alternate name "1.". If 1* has been renamed in the DMCL, it must have that alternate name. The schema file is required in the AFT at execution time.

- Validated Subschema File - The validated subschema file, a random file produced by the subschema translation and validation, has the alternate name "6*" and is required in the AFT at compilation time.

- Subschema Control Structure - Unlike the other I-D-S/II files, the subschema control structure, a sequential file produced by the subschema validation, is not accessed from the AFT. This file, which was referenced by the filecode C* during validation, is bound instead with the FORTRAN object program at load time. It consists of two object modules, S.xxxx and D.xxxx, where xxxx are the first four letters of the subschema name.

Data base area files are required. Data base key files may also be required. Both type of files must be placed in the AFT under their alternate names (i.e., the file codes which were specified in the schema DMCL). The following types of data files can be specified:

- Integrated
- Integrated with Record Keys
- Indexed
- Indexed with Record Keys

If any required data base procedures were not included in the FORTRAN source program itself, files containing these procedures must be supplied. These files, produced during previous compilations, supply the procedures specified in the schema and subschema. These object units, like the control structure, are bound with the FORTRAN object program at load time.

When the DML option is specified, an INVOKE statement in the FORTRAN source program enables the FORTRAN compiler to read the 6* file and obtain the subschema. The subschema then becomes part of the FORTRAN program and defines the User Working Area (UWA).

At run time, the schema file (1*) and the data base area and key files must be in the AFT under the appropriate alternate names. The control structure is used at run time to describe the subset of the data base which is accessible to the program.

Comparison of the FORTRAN and YFORTRAN Time Sharing Systems

There are two time sharing versions of the FORTRAN compiler. Each version is invoked by the call specified below.

Compiler Version	Language Call
Batch based time sharing compiler	YFORTRAN
Time sharing based compiler	FORTRAN

The time sharing based FORTRAN compiler compiles under the time sharing system rather than being spawned as in the case of the batch based time sharing compiler. It differs from the batch based compiler because it

- Compiles under the GCOS time sharing system
- Eliminates the need for configuring batch memory; YFORTRAN compiles through DRL TASK (Refer to the <u>TSS System Programmer's Reference Manual</u>).
- Significantly reduces overhead in the FORTRAN time sharing system
- Does not require the "CORE=" clause for compilations
- Has identical compilers with the exception of the executive phase (YFXC vs YUEX)

THE YFORTRAN TIME SHARING SYSTEM RUN COMMAND

The YFORTRAN time sharing RUN command can be written as either RUN or RUNH. The RUNH form is used to display a heading line on the terminal that gives a date, a time, and a SNUMB. Any of the seven following options can be specified with the RUN (or RUNH) command:

```
RUN [H] [- nnn] [fs] [ = [fh] [; fc] [(opt [,...] )] [ulib]] [#fe]
```

-nnn nnn is the maximum processor time (in seconds) the program is allowed to run during execution.

fs is the set of file descriptors (separated by semicolons) for source files in the standard BCD card image format, in compressed card image format (COMDK), or in time sharing ASCII standard system format, and/or descriptors for binary card image object files. These files serve as inputs to the compiler and/or loader. Concatenation of source files is provided by using a separate semicolon between each file descriptor. Where a BCD or COMDK source file is supplied (media code 1 or 2), fs can also include a descriptor for a BCD alter file. The alter file must begin with a $ UPDATE card and must be in alter number sequence. If there are many BCD or COMDK source files in the list, the alter file updates the first source file. If the FORTRAN program contains I-D-S/II DML statements, fs should also contain the file descriptor for the subschema control structure file. If data base procedures are required and are not supplied as part of the FORTRAN source program, file descriptors for the procedure object files should also be listed here.

Alternatively, the list fs can consist of a single file descriptor that points to a previously generated system loadable (H*) file.

A file descriptor consisting of the single character "*" indicates the current file (*SRC). The fs list is optional, and when missing, indicates that only the current file (*SRC) is to be compiled.

fh is a single file descriptor of a random file into which the system loadable file (H*) produced by the General Loader is saved if the compilation is successful. This file is written if no fatal errors occur during compilation. If the named file does not exist, a permanent random file of 36 blocks (llinks) is created and added to the user's catalog. If the field is missing, the H* file is generated into a temporary file. The presence of this option is valid only when the program indicated by the list fs, the FORTRAN library, and the user library (if any) is bindable (i.e., no outstanding SYMREFs). If the General Loader indicates that outstanding SYMREFs exist, an executable H* file is created, but any reference to an unsatisfied SYMREF causes the program to be abnormally terminated. (The General Loader inserts a MME GEBORT at references to unsatisfied SYMREFs. When a MME is encountered during the execution of a time sharing subsystem, GCOS and the Time Sharing Executive simulate an illegal operation fault.)

;fc is a single file descriptor preceded by a semicolon of a sequential file into which the compiler is to place the binary (C*) result of any indicated compilation(s). One object module is written to this file for each source program in the file(s) given by fs. If a subschema control structure was specified in the fs list, two object modules, S.xxxx and D.xxxx, are written to this file. Any data base procedure object units specified in the fs list are also written to this file.

If the named file does not exist, a permanent linked file of three blocks (llinks) is created and added to the user's catalog. This file expands as necessary up to a maximum of 20 blocks (llinks), to hold the object deck(s). In this case, the field fs plus the libraries do not need to indicate a complete program (individual or collections of subroutines can be compiled and saved). When this optional field is missing, a C* file is not generated; when present, the DECK option is activated for the compilation process.

(opt) is a list of options available for time sharing which, when specified, must be separated by commas. Some of these options affect the compilation process and some affect the loading process (the default options are underlined).

DEBUG - The run time debug symbol table is generated.

NOTE: This debug symbol table is used for debugging in the batch mode only. Refer to the <u>General Loader</u> manual for use of the debug feature and the debug symbol table.

<u>NDEBUG</u> - The run time debug symbol table is not generated.

BCD - Object character set is BCD. If applicable, this option must be specified whenever the General Loader is to be called. This is required for compile, compile and load, and load activities; it is not required for execute only runs (run H* file). The BCD option cannot be specified if the DML option is selected.

<u>ASCII</u> - Object character set is ASCII.

FORM - Source is in "fixed" format (LNO option is not valid with FORM).

<u>NFORM</u> - Source is in "free" format.

<u>LNO</u> - Source is line numbered (default option if FORM is not specified).

NLNO - Source is not line numbered (default option if FORM is specified).

OPTZ - The object module is optimized.

<u>NOPTZ</u> - The object module is not optimized.

NWARN - No compilation warning messages are printed, although, fatal messages are printed.

CORE=nn - The compilation activity memory requirement is set to nnK+9K or 29K, whichever is larger. If not specified, nn is set to 20.

FDS - The FORTRAN Debugging System (FDS) is enabled. See Appendix F.

<u>NFDS</u> - The FORTRAN Debugging System is not invoked.

DML - The Data Manipulation Language (DML) facility of I-D-S/II is invoked. If DML is specified, the necessary I-D-S/II files must also be specified in the RUN command. The BCD option cannot be used with the DML option.

STAT - The I-D-S/II statistics are printed. If a sequential file with the alternate name "P." exists in the AFT, the I-D-S/II statistics and abort codes are written to that file. The file is written as a BCD file and can be converted to an ASCII file for examination from a terminal by the command "CONV file descriptor". If "P." does not exist in the AFT, the statistics and abort codes are specified, and written to the terminal. If the STATS option is not specified, the I-D-S/II statistics are not printed and the fatal abort codes are directed to the terminal. A FINISH statement must be included in the FORTRAN program in order to receive any statistics. STAT is valid only when the DML option is specified.

LDEL - Logical record delete is requested. The default is physical record deletion. LDEL is valid only when the DML option is specified.

The remaining options concern the loading process (the default option is underlined).

<u>GO</u> - The program is loaded and executed at the completion of compilation.

NOGO - The program is not executed at the completion of the compilation. If specified, the object program is loaded and saved. If no object (H*) save file is specified, only the compilation is performed (General Loader is not called).

ULIB - File descriptors exist at the end of the options field that allocate user libraries to be searched for missing routines prior to searching for them in the system library.

<u>NOLIB</u> - No user libraries are to be used.

TIME=nnn- The batch compilation and/or General Loader activity time limits are set to nnn seconds; where nnn $\leq$ 180. If not specified, nnn is set to 60.

URGC=nn - The urgency for the batch compilation and/or General Loader activity is set to nn, where nn $\leq$ 40. If not specified, nn is set to 40.

TEST - A test version of the compiler is to be used for the activity. There must be an accessed file (in the AFT) with the name FORTRANY. If these two conditions are met, then file FORTRANY is allocated as file code ** in the activity.

REMO - All temporary files that are created during compilation and loading are removed from the AFT as they are no longer needed. This option keeps the number of files in the AFT down to a minimum but causes more time to be spent processing each RUN command.

NAME=name - Provides a name for the main link of the saved H* file. It can be used at time of creation of this file and subsequently as it is reused. This name is placed in the SAVE/field of the $ OPTION card.

ulib A list of file descriptors (separated by semicolons) pointing to random files containing user libraries to be searched before the system library. This list must be provided by the user when the ULIB option is specified.

#fe A list of file descriptors (the first preceded by a number sign) for files required during execution. Each catalog/file description is separated by a semicolon (refer to the <u>TSS General Information</u> manual). The file description can be in any of the following formats:

1. <u>filename</u> in the form filename "nn", represents a logical file code referenced by the I/O statements in the program where 01 ≤ nn ≤ 63.

2. <u>filedescr</u> specifying a full description.

   a. <u>filename</u> "nn"

   b. <u>filename$password</u> "nn"

   c. <u>userid/catalog$password</u> "nn"

Filecodes 05, 06, 41, 42, and 43 are implicitly defined for terminal directed I/O and do not need to be mentioned in the RUN command unless I/O is to be directed to a file. Other logical file codes can be terminal-directed by specifying a descriptor of the form filename "nn", where "nn" is the desired filecode.

The I-D-S/II files required for compilation and execution should also be specified in the #fe list. #fe should contain the file descriptor for the 6* subschema file required for compilation with the alternate name "6*".

<u>Example</u>

```
FORTY/DML/6STAR"6*"
```

#fe should also contain the file descriptors for the I-D-S/II files required for execution that include:

1. <u>Schema File</u> - This file must have the alternate name "I.". If an alternate filecode was specified in the DMCL schema entry, it must have that alternate name.

2. <u>Data Base Area and Key Files</u> - These random files must have alternate names which are the same as the filecodes defined in the DMCL entry.

3. Statistics File - If the STAT option is specified and the output is to be written to a file, the desired file descriptor with the alternate name "P." should be entered in the #fe list.

   Example

```
FORTY/DML/SCHEMA"L."
FORTY/DML/AREA1"A1"
FORTY/DML/KEY1"K1"
FORTY/DML/STATUS"P."
```

FORTRAN TIME SHARING SYSTEM RUN COMMAND

The FORTRAN time sharing RUN command can be written as either RUN, RUNH, FRN, or FRNH. The RUNH form is used to display a heading line on the terminal giving date and time. Any of the seven following options can be specified with the RUN (or RUNH) command:

```
FRN [H]   [-nnn]  [fs] [= [fh] [; fc] [(opt [,...] )] [ulib]] [#fe]
```

-nnn is the maximum processor time (in seconds) the compiled object program is allowed to run during execution.

fs is the set of file descriptors (separated by semicolons) for source files in the standard BCD card image format, in compressed card image format (COMDK), or in time sharing ASCII standard system format, and/or descriptors for binary card image object files. These files serve as inputs to the compiler and/or the time sharing loader. Concatenation of source files is provided by using a separate semicolon between each file descriptor.

Where a BCD or COMDK source file is supplied (media code 1 or 2), fs may also include a descriptor for a BCD alter file. The alter file must begin with a $ UPDATE card and must be in alter number sequence. If there are many BCD or COMDK source files in the list, the alter file updates the first source file.

If the FORTRAN program contains I-D-S/II DML statements, fs should also contain the file descriptor for the subschema control structure file. If data base procedures are required and are not supplied as part of the FORTRAN source program, file descriptors for the procedure object files should also be listed here. The list fs can consist of a single file descriptor that points to a previously generated system loadable (H*) file.

A file descriptor consisting of the single character * indicates the current file (*SRC). The fs list is optional, and when missing, indicates that only the current file (*SRC) is to be compiled.

fh is a single file descriptor of a random file into which the system loadable file (H*) produced by the general loader is saved if the compilation is successful. This file is written if no fatal errors occur during compilation. If the named file does not exist, a permanent random file of 36 blocks (llinks) is created and added to the users' catalog. If the field is missing, no temporary H* file is created. If this is the case, the time sharing loader creates a complete bound memory-image of the object execution program, "releases" itself via DRL RELMEM, and enters the execution directly.

If the time sharing loader indicates that outstanding SYMREFs exist, any reference to them during object program execution causes abnormal termination via a DRL ABORT.

;fc is a single file descriptor (preceded by a semicolon) of a sequential file into which the compiler is to place the binary object (C*) result of any indicated compilation(s). One object module is written to this file for each source program in the file(s) given by fs. If a subschema control structure is specified in the fs list, two object modules, S.xxxx and D.xxxx, are written to this file. Any data base procedure object units specified in the fs list are also written to this file.

If the named file does not exist, a permanent linked file of three blocks (llinks) is created and added to the user's catalog. This file expands as necessary up to a maximum of 20 blocks (llinks), to hold the object deck(s). When C* is specified, a compiler temporary file (*1 scratch file) of 48 blocks (llinks) is defined and its name is placed into the AFT.

(opt) is a list of comma-separated compiler/loader options available in the time sharing based FORTRAN system. Those options available with the YFORTRAN RUN command but not specified here are not currently used with the FORTRAN RUN command. They are ignored if specified (default options are underlined).

BCD - The internal character set for object program execution is BCD. If applicable, this option must be specified whenever the loader is called. This is required for compile, compile and load, and load activities; it is not required for execute only runs (from the H* save file). The user should not load object deck files compiled under different options (i.e., one under BCD and another under ASCII) since execution results would be unpredictable. The BCD option cannot be specified if the DML option has also been selected.

ASCII - Internal character set for the object program execution is ASCII.

FORM - Source is in "fixed" format (LNO is not valid with FORM).

NFORM - Source is in "free" format.

LNO - Source is line numbered (default option if FORM is not specified).

NLNO - Source is not line numbered (default option if FORM is specified).

OPTZ - The object module is optimized.

NOPTZ - The object module is not optimized.

NWARN - No compilation warning messages are printed, although fatal messages are printed.

FDS - Enables the FORTRAN Debugging System (FDS). See Appendix F.

DML - The Data Manipulation Language (DML) facility of I-D-S/II is invoked. If DML is specified, the necessary I-D-S/II files must also be specified in the RUN command. The BCD option cannot be used with the DML option.

STAT - The I-D-S/II statistics are printed. If a sequential file with the alternate name "P." exists in the AFT, the I-D-S/II statistics and abort codes are written to that file. The file is written as a BCD file and can be converted to an ASCII file for examination from a terminal by the command "CONV file descriptor". If "P." does not exist in the AFT, the statistics and abort codes are written to the terminal. If the STATS option is not specified, the I-D-S/II statistics are not printed and the fatal abort codes are directed to the terminal. A FINISH statement must be included in the FORTRAN program in order to receive any statistics. This option is valid only if the DML option is specified.

LDEL - Logical record delete is requested. The default is physical record deletion. This option is valid only if the DML option is specified.

The following remaining options concern the loading process:

<u>GO</u> - The program is executed at the successful completion of the compile-load process.

NOGO - The program is not executed at the completion of the compilation. If specified, the object program is loaded and saved. If no object (H*) save file is specified, only the compilation is performed (the General Loader is not called).

ULIB - File descriptors (separated by semicolons) exist following the end of the options field that allocate user libraries to be searched for missing routines prior to searching for them in the system library.

*

<u>NOLIB</u> - No user libraries are to be used. Specification of user libraries in this case causes a RUN diagnostic.

CORE = nn where nn is additional memory (mod 1024) to be added to the standard time sharing loader allocation of 25K. This should be done if the message "<F> PROGRAM EXCEEDS STORE SIZE" appears. The compiler attempts to estimate the space requirements for the load process by accumulating the size of the generated memory, .DATA. region, labeled common and blank common for each subprogram compiled; then adding a constant (11K for the standard library) to this to arrive at the size of a load space requirement. If the message 'NOT ENOUGH CORE TO RUN JOB' appears, TSS allocation is too small to compile/load this program.

MAP - A memory map is produced after loading.

ulib - a list of file descriptors (separated by semicolons) pointing to random files containing user libraries to be searched before the system library. This list must be provided by the user when the ULIB option is specified. Up to nine user library files can be specified.

#fe - A list of file descriptors (the first preceded by a pound sign) for files required during execution. Each catalog/file description is separated by a semicolon (refer to the <u>TSS General Information</u> manual). The file description can be in any of the following formats:

1. <u>filename</u> in the form filename "nn", represents a logical file code referenced by the I/O statements in the program where 01 $\leq$ nn $\leq$ 63.

2. <u>filedescr</u> specifying a full description.

   "nn"
   <u>filename</u> "nn"
   <u>filename$password</u> "nn"
   <u>userid/catalog$password</u> "nn"

   Filecodes 05, 06, 41, 42, and 43 are implicitly defined for terminal directed I/O and need not be mentioned in the RUN command unless I/O is to be directed to a file. Other logical file codes can be terminal directed by specifying a descriptor of the form "nn", where "nn" is the desired filecode.

   The I-D-S/II files required for compilation and execution should also be specified in the #fe list. #fe should contain the file descriptor for the 6* subschema file required for compilation with the alternate name "6*".

   <u>Example</u>

   ```
   FORTY/DML/6STAR"6*"
   ```

   #fe should also contain the file descriptors for the I-D-S/II files required for execution that include:

   - <u>Schema File</u> - This file must have the alternate name "I.". If an alternate filecode was specified in the DMCL schema entry, it must have that alternate name.

   - <u>Data Base Area and Key Files</u> - These random files must have alternate names which are the same as the filecodes defined in the DMCL entry.

   - <u>Statistics File</u> - If the STAT option is specified and the output is to be written to a file, the desired file descriptor with the alternate name "P." should be entered in the #fe list.

     <u>Example</u>

     ```
     FORTY/DML/SCHEMA"L."
     FORTY/DML/AREA1"A1"
     FORTY/DML/KEY1"K1"
     FORTY/DML/STATUS"P."
     ```

Example

1. Create a random file of 50 llinks, with general read permissions to contain the user's library with the ACCESS subsystem. ACCESS CF,/ULIB1,B/50,50/,R,MODE/R/

2. Listing of a deck setup for creating and saving a user library file (through JRN or batch).

```
 1      8         16
 
 $      IDENT     .....
 $      USERID    UMC$PASSWD
A$      FILEDIT   NOSOURCE,OBJECT,INITIALIZE
 $      FILE      R*,F1S,10L
 $      DATA      *C,,COPY
 $      SELECTD   UMC/OBJDECK1
 $      SELECTD   UMC/OBJDECK2
 $      SELECTD   UMC/OBJDECK3
 $      ENDEDIT
 $      ENDCOPY
A$      PROGRAM   RANLIB
 $      PRMFL     A4,W,R,UMC/ULIB1
 $      FILE      R*,F1R,10L
 $      ENDJOB
```

*

Alternate Named Files

For files required during execution, the programmer can designate an alternate name by using the following format:

filedescr "altname"

where: altname = nn; attaching the logical file code nn to the specified file.

*

Example

RUN#"10"

If a given file descriptor consists of only a two-digit logical file code not enclosed within quotation marks, a temporary file is created unless a quick-access permanent file with the same name already exists. The PERM command can subsequently be used to make the temporary file permanent. Alternatively, such temporary files can be made permanent at the time the user logs off.

Example

RUN PROGRAM#10

If no file exists in the user's catalog with the name 10, a linked temporary file is created with that name and I/O that was directed to the logical file code 10 is routed to the temporary file.

The fe list of the RUN command serves two additional functions: the creation of a file control block, and the association of the logical file code with some specific file, or the terminal. When this association involves a catalog file descriptor, that file is accessed (or created) and added to the user's available file table (AFT); the file is then allocated to the process. This is analogous to the allocation by the $ PRMFL and $ FILE control cards in a batch operation.

When a file is first referenced by an executing program, a general file "open" function is invoked. At this time, the file control block comes into play as one of three possibilities:

1. There is no file control block for the referenced file.
2. The file control block indicates that the terminal is to be used.
3. The file control block indicates that a file is to be used.

If there is no file control block, one is automatically generated indicating that a file is to be used. When the file control block indicates that the terminal is to be used, the device attachment is completed and I/O proceeds. When the file control block indicates that a file is to be used (cases 1 and 3), the AFT is searched. If a match is found (i.e., an allocated file has a two-digit file code/name equivalent to the file description in the I/O statement), attachment is made to that file and I/O proceeds. If no match is found (i.e., there has been no file allocation for the current file designator), a comment is displayed on the terminal identifying the undefined file designator.

Example

```
FILE XX NOT IN AFT. ACCESS CALLED
```

where: XX is the two-digit file designator being referenced by the running program.

At this point, the ACCESS subsystem is called (as indicated by the above message) and displays:

```
FUNCTION?
```

Commands can now be given to ACCESS. When the dialog is finished, ACCESS returns to the user's program. The "open" routine then makes a fresh search of the AFT. If a match is now found (indicating some file has been accessed), attachment is made to that file and I/O proceeds. If a match is not found, the file control block is changed to indicate attachment to the terminal and I/O proceeds.

Example

Consider that PROGRAM contains I/O statements with a file designator of 10 and the following dialogue transpires:

```
*FORTRAN
*OLD PROGRAM
*RUN

FILE 10 NOT IN AFT. ACCESS CALLED

FUNCTION?
```

If the response is a carriage return, the terminal is used for file 10. If the response is

```
AF,/MYFILE"10",R,W
```

the ACCESS subsystem accesses the file MYFILE of the user's master catalog under the alternate name 10 with read and write permissions. ACCESS then repeats the query "FUNCTION?". If the user now responds with a carriage return, I/O for file 10 is directed to MYFILE.

One additional option exists for the purpose of collecting the results of a compiler abort. If at the time the RUN command is issued there exists a file in the AFT of name ABRT, that file is allocated to the compilation activity as file code *F. In the event of a compiler abort, a memory dump and symbolic display of the internal tables is written to this file in a form suitable for printing.

Accessing I-D-S/II Files Required for Execution

The I-D-S/II files necessary for execution can be accessed by listing them in the #fe list of the RUN command as specified above or by the time sharing GET command. Another alternative is to use calls to the supplied FORTRAN subroutine ATTACH.

Example

```
CALL ATTACH (1,"FORTY/DML/AREA1""A1"";",1,0,ISTAT,)
            .
```

The file is placed in the AFT under the alternate name "A1" which is the filecode specified in the schema DMCL. The schema file 1* cannot be accessed in this way because 1* is necessary for the execution of the INVOKE statement, and INVOKE must be the first executable statement.

First Line Run Command

The RUN command can be designated as the first line or lines of the source program. This is useful when running FORTRAN programs with DML statements because the RUN command may require several lines of input to specify all the I-D-S/II files. The following rules apply to the first line of the RUN command.

1. This feature is available on time sharing ASCII files only.

2. The line can be in the current file (*SRC) or a referenced permanent file; however, it must begin with the first line of the first source file.

3. The first two characters following the line number must be *# with no embedded blanks.

4. Multiple *# lines can appear in a source file, provided the total number of characters does not exceed 480 (six 80-character lines).

5. The lines must conform with the RUN syntax continuation (i.e., each line, except the last, must be terminated by one of the following field-separating delimiters: equal sign; left parenthesis; right parenthesis; semicolon; or pound sign).

6. The line(s) are treated as comment line(s) by the FORTRAN compiler.

7. The first line contained RUN command can be overridden by indicating save files, options, or concatenation on the RUN type-in.

Example

```
*FORTRAN
*NEW
*010*#RUN    *(20,30)=HSTAR(BCD,NOGO)
*020  PRINT, "HELLO DOLLY..."
*030  STOP; END
*RUN (Invokes first line syntax)
```

DML Example

```
*#RUNH*;FORTY/DML/CSTAR=HSTAR(DML)#FORTY/DML/6STAR"6*";
2*#FORTY/DML/SCHEMA"1.";
3*#FORTY/DML/AREA1"A1";FORTY/DML/KEY1"K1"
```

1. RUN

   The current *SRC FORTRAN source file is compiled and executed.

2. RUNH-20 FR001=HSTAR; CSTAR1 (ULIB) ABC; XYZ #

   INPUT "01" ; OUTPUT "02"

   FORTRAN program file FR001 is to be compiled and executed. The H* is saved on file HSTAR and C* on file CSTAR1. For the execution, the random user libraries ABC and XYZ are scanned for outstanding SYMREFs in FR001. Logical file codes 01 and 02 have been used as alternate names for the quick-access permanent files INPUT and OUTPUT. A heading line for the date and time is displayed and the object program is limited to 20 seconds of execution time.

3. RUN #"10"

   The current *SRC file is compiled and executed and I/O through logical file code 10 is directed to/from the terminal.

4. RUN BCDIOM = CSTAR2 (BCD,NOGO)

   FORTRAN file BCDIOM is compiled and the object deck is saved on file CSTAR2. The object file is to be executed in BCD mode.

5. RUN HSTAR #02

   Execute a previously bound and saved H* file. The quick-access file "02" is accessed by the RUN subsystem. If no such file exists, a temporary file is created.

6. RUN = HSTAR (TIME=60, CORE=22, ULIB) SEARCH

   Compile and execute the current *SRC file, saving the bound H* file on random file HSTAR. Limit the compile time to 60 seconds and increase the memory limits. The random user library 'SEARCH' is searched to satisfy outstanding SYMREFs prior to searching the standard system library.

7. RUNH *(10,190); SCRLIB(300,)

   Compile and execute the program by concatenating the current file lines 10 through 190 and file SCRLIB lines 300 through the last line of the file.

8. RUN *; CSTAR1; CSTAR2

   Compile and execute the current *SRC file and bind it with two previously saved C* files: CSTAR1 and CSTAR2.

DML TSS Example

```
RUN *;FORTY/DML/CSTAR=(DML,STAT)#FORTY/DML/6STAR"6*";
FORTY/DML/SCHEMA"1.";FORTY/DML/AREA1"A1";
FORTY/DML/KEY1"K1";FORTY/DML/STATUS"P."
```

The current *SRC file is compiled using the subschema file "6*" and bound with the subschema control structure. The resulting object code is executed using the schema file ("1."), one data base area file ("A1"), and one data base key file ("K1"). The I-D-S/II statistics and abort codes are written to the file "P.".

Batch Activity to Build Time Sharing H* File

The following example program illustrates a method of building a time sharing H* file in batch mode.

```
 1      8        16
-------------------------------------
 $      SNUMB    .....
 $      IDENT    .....
 $      LOWLOAD  100
 $      USE      .GRBG./36/
 $      OPTION   NOFCB,NOGO,SAVE/object
 $      USE      .GTLIT,.TSGF.,.FTSU.,.FXEMA
A$      FORTY    NFORM,NLNO,ASCII
 $      SELECTA  source program file
A$      EXECUTE  DUMP
 $      PRMFL    H*,W,R,Hstar file
 $      ENDJOB
```

Time Sharing System RUNL Command for Link/Overlay

When a bound object program is too large for execution under time sharing, segmentation is achieved by using a special form of the RUN command (RUNL) to link/overlay H* files that are to be constructed. When the RUNL command is used, a PSTR printout can be obtained with the YFORTRAN system but not with the FORTRAN system.

Before the RUNL command can be used, a separate RUN command with the NOGO option must have been specified to create each of the C* files that will be needed in the RUNL command. This command can be written as RUNL or RUNLH where the latter form displays a heading line with the current date and time (and SNUMB if YFORTRAN), with the format

```
RUNL[H] C*file list = H*file[(options)][ulib files]; link list
```

C* file list	- The set of file descriptors for the binary object files for the nonoverlayed main program link.
H* file	- A single file descriptor of a random file into which the system loadable file produced by the loader is saved if the load process is successful. If the named file does not exist, a file of 216 llinks (random temporary) is created.

(options):

ULIB - File descriptors exist at the end of the options field that locate user libraries to be searched prior to searching the system library. The load process for each link involves searching the same set of user libraries first.

CORE = nn - The YFORTRAN memory requirements are set to nn+9K or 29K, whichever is larger. If not specified, nn is set to 20K.

The FORTRAN link loader memory requirement is nnK if nn < 23K or 23K+ nnK if nn > 23.

NAME = name - Provides a name for the main link of the saved H* file; when not provided, the name "//////" is used.

MAP - If the user has previously defined a file with the name PSTR, a load map of the link/overlay save file is written to that file. Otherwise, a temporary file is created by that name and the output is written to that file. This feature is currently available only under the YFORTRAN system.

GO - Allows a user to enter execution directly from the RUNL command (the default is NOGO). The user must provide for run time file definition and dynamic attaching through "CALL ATTACH", etc. If it is necessary to specify through RUN the necessary object time files, the user must explicitly use the RUN command after creating the link/overlay H* file.

Example

RUN HSTAR#INPUT"01";OUTPUT"02"

link list - A sequence of link phrases wherein each link phrase is used to specify the position at which segmentation is to take place. When the link phrase is encountered in the RUNL command, all object deck files for the link being terminated have been copied to the loader input file R*. The link phrase is parsed, resulting in the generation of a $ LINK card image and possibly a $ ENTRY card image being written to R*.

Formats

LINK(name1[,name2]) C*file list for name1

LINK(name1[,name2,entry]) C*file list for name1

LINK(name1[,,entry]) C*file list for name1

where: name1 (a five- or six-character constant or variable) is a unique identifier for the new link

name2, if present, is the identifier of the previously loaded link to be overlayed. The new link assumes the origin of the old link. All links to be overlayed are written in system loadable format

<u>entry</u>, if specified, is the name of the desired primary or secondary SYMDEF entry point of a subprogram in the current link

Subprograms contained in any other link can always reference subprograms in the main link. Only links that reside in memory at the same time can reference each other. For example, if link B is loaded as an overlay of link A (LINK (B,A)), the subprograms of link B cannot reference subprograms of link A.

NOTES: 1. To ascertain the size required to allocate a permanent H* save file, create a temporary file by means of RUNL. Then use the PERM command to create a permanent file. The size of the permanent file will automatically be chosen just large enough to contain the "used" llinks in the temporary file.

2. Under YFORT, "PSTR" load map generated by the General Loader in batch can be sent to a remote station or central site printer, if it is a permanent file.

<u>Example</u>

```
PERM PSTR;PS        Make file permanent if temp used
SCAN PS
FORM? LOAD          Print number of errors
000 ERRORS
EDIT? YES           For multiple-blank suppression
?BATCH
STATION CODE        Reply XX or carriage return

                      XX = remote station code
                      carriage return = central site printer

$ IDENT             Input batch $ IDENT card
```

Alternatively, a BMC run in batch can print the file.

3. A temporary H* save file cannot be command-loaded; use the LODT command (not LODX). The YFORTRAN or FORTRAN RUN command should be used, since run time files can then be specified.

4. The name of the main link is //////, unless NAME=name is used as an option. The user must specify the name when loading the H* save file.

5. Creating a multiple-line embedded RUNL command is the best way to deal with a long, complex command.

<u>Example</u>

```
1*#RUNLH MAIN; SUB1;SUB2=HSTAR (ULIB,MAP)
2*#FY/SDL7LIB,R;
3*#LINK (A)SUB3;SUB4;
4*#LINK (B,A,ENTRY5)SUB5;SUB6;
5*#LINK (C,B)SUB#;SUB8
```

Observe rules for line termination.

6. After the loader builds the H* save file containing the links, it is necessary to reload these links in the order required to achieve the program function. Reloading is done by means of a time sharing library routine (FTLK) that has two entries, LINK and LLINK. LINK is callable from the FORTRAN source to load a particular link and transfer control to a predesignated entry within that link. This SYMDEF must be specified in the "entry" field of the link phrase. LLINK can be called to load a particular link and return control to the place in the program at which LLINK has been called. The two calls are as follows:

```
CALL LINK ("A    ")
CALL LLINK ("B    ")
```

   The link names must be either five or six characters in length and blank-filled as needed.

7. When using FORTRAN random I/O, the CALL RANSIZ statement must be placed in the main link. This assures proper file wrapup by forcing the random I/O subroutine FRRD to reside with the main link in memory at all times.

8. The main link in a link/overlay run must contain some input/output when the Hstar file is to be executed in the time sharing mode.

9. The RUNL command cannot be used to process octal patch corrections under the FORT system.

Example of RUNL Inputs and Link H* Creation

Ten subroutines plus a main program are to be executed under time sharing. The first overlay (link A), is to have three subroutines; the second overlay (link B), four subroutines; and the third overlay (link C), three subroutines.

1. Compile and save the C* object deck files (CSTAR) for each program.

```
RUN MAIN =;CSTAR1(NOGO)
RUN SUBA;SUBB;SUBC =;CSTAR2(NOGO)
RUN SUBD;SUBE;SUBF;SUBG =;CSTAR3(NOGO)
RUN SUBH;SUBI;SUBJ =;CSTAR4(NOGO)
```

2. Create a link overlay H* file (HSTAR) using RUNL.

```
RUNL CSTAR1 = HSTAR(ULIB,MAP) ULIB1;
LINK(A) CSTAR2; LINK(B,A,ENTRYB)CSTAR3;LINK(C,B) CSTAR4
```

3. Load and execute the H* save file specifying core limits and run-time input/output files.

```
RUN HSTAR=(CORE=35K)#INPUT"41";OUTPUT"13"
```

Example of LINK/LLINK Usage

1. Compile and save the C* object deck files for the main program and the two subroutines.

```
010 PRINT,"MAIN EXECUTING"
020 CALL LLINK ("A     ")
030 CALL SUBA
040 CALL LINK ("B     ")
050 STOP;END

RUN =;MAIN(NOGO)
NEW


010 SUBROUTINE SUBA
020 PRINT,"LINKA EXECUTING"
030 RETURN; END

RUN=;ALINK(NOGO)


010 SUBROUTINE SUBB
020 PRINT, "LINKB EXECUTING"
030 RETURN; END

RUN=;BLINK(NOGO)
```

2. Create a link overlay H* file using RUNL.

```
RUNL MAIN=HSTAR;LINK(A) ALINK;LINK(B,A,SUBB)BLINK
```

3. Load and execute the H* file.

```
RUN HSTAR
   or
FRN HSTAR=(CORE=32K)
```

Example of Loader Input File

The following control card setup would appear on R* for the example above illustrating the use of LINK/LLINK.

```
 $         LOWLOAD
 $         USE         .GRGB./36/
 $         USE         .GTLIT,.TSGF.,.FTSU.,.FXEMA,.FTLK
 $         OPTION      NOMAP
 $         OPTION      NOGO
 $         OBJECT
 $         DKEND
 $         LINK        A
 $         OBJECT      SUBA
 $         DKEND       SUBA
 $         LINK        B,A
 $         ENTRY       SUBB
 $         OBJECT      SUBB
 $         DKEND       SUBB
A$         EXECUTE
```

Example of a Time Sharing Session

A comprehensive example of program creation, testing, correction and modification follows. Replies to the user from the system are underlined. Explanations are enclosed in parentheses and are not part of the printout.

```
USER ID - J.P.JONES
PASSWORD--
ABDBEKGH
*FORTRAN
*NEW
*AUTØX - (enter automatic-line-number mode)
*0010      READ,A,B,C
*0020      X1=A*B/C
*0030      X2=A**2;B**2
*0040      ANS=X2/X1
*0050      PRINT 10,X1,X2, ASN###ANS (typing error correction)
*0060 10   FØRMAT(1X,"X1=",F6.S#2,"X2=",F7.2,"ANS=",
*0070      F6.2)
*0080      STØP
*0090      END
*0100      (end automatic mode by carriage return)
*0030      X2=A**2+B**2-C (replacement of line 30)
*SAVE FØRT01
DATA SAVED--FØRT01

*LIST      (display corrected program)
0010       READ,A,B,C
0020       X1=A*B/C
0030       X2=A**2+B**2-C
0040       ANS=X2/X1
0050       PRINT 10,X1,X2, ANS
0060  10   FØRMAT(1X,"X1=",F6.2,"X2=",F7.2,"ANS=",
0070       F6.2)
0080       STØP
0090       END

*RUN      (run program)

= 3.2,10.5,2.2 (type input data)
X1= 15.27X2= 118.29ANS= 7.75    (output - correct,
                                  but poor format)

*0060 10 FØRMAT(1X,"X1="  ,F6.2," X2=",F7.2,"  ANS=",
                               (correct format statement)
*RUN

= 3.2,10.5,2.2
X1= 15.27  X2= 118.29  ANS= 7.75 (improved output format)
*RESAVE FØRT01
DATA SAVED--FØRT01

*BYE (finished)
**RESØURCES USED $  2.08, USED TØ DATE $  263.85= 27%
**TIME SHARING OFF AT 15.421 ON 07/10/79
```

Supplying Direct-Mode Program Input

During program execution, keyboard input may need to be supplied to satisfy one or more READ statements in the program. Each time input is required, the equal-sign character, "=", is printed at the terminal. The user begins typing the input immediately following the equal sign.

It is also possible to input data from a paper tape. The actual characters transmitted to the terminal from a READ statement are

- carriage return (CR)
- line feed (LF)
- equal sign (=)
- sign-on (X-ON)

The sign-on character activates the paper tape reader if the reader is in the ready state which is achieved by having the paper tape "loaded" and the reader switch set on. Paper tapes which are to be used in this way should end each line with the characters

- carriage return (CR)
- line feed (LF)
- rubout (RO)
- sign-off (X-OFF)

NOTE: The sign-off character, X-OFF, turns off the reader but leaves it in a ready state for any subsequent READs.

Terminal output from the PUNCH statement automatically appends this control information to the end of each line to facilitate the preparation of the tapes. In any event, the user must manually begin such tapes with an appropriate leader of RO characters.

Emergency Termination of Execution

The use of the BREAK key terminates program execution and the terminal buffer is flushed. Control returns to the readiness status for entering commands or building files after the use of the break key.

Paper Tape Input

In order to supply build-mode input from paper tape, the user gives the command TAPE. The system responds with READY. At this point, the user should position the tape in the reader and start the device. Input is terminated when one of the following conditions occurs:

- The end-of-tape occurs
- The reader is turned off
- An X-OFF character is read by the paper tape reader
- A jammed tape causes a delay of more than one second between the transmission of characters

At present a maximum of 80 characters are permitted per line of paper tape input. Longer lines are truncated at 80 characters with the remaining data placed in the next line. A maximum of two disk links (7680 words) of paper tape input is collected during a single input procedure. All data in excess of two disk links is lost.

LIMITATIONS IMPOSED BY THE AFT

The AFT allows a maximum of 20 files. This may restrict the running of FORTRAN DML programs in time sharing since a compile-and-execute run requires a source file, subschema files (6* and C*), a schema file (1*), and data base area and key files. If the number of data base areas and key files is large, the run may require more files than allowed in the AFT. Note that the collector file sy** is always present in the AFT.

One way to avoid this difficulty is to use a system-loadable file (H*). The source program can be compiled with the subschema file (6*) and bound with the control file (C*) to produce the H* file. The AFT can then be cleared. The files required for execution can be accessed under their alternate names by the time sharing GET command. Data base area and key files can also be accessed by calls to ATTACH in the FORTRAN source program. The H* can then be run.

Example

```
RUN DMLTEST;FORTY/DML/CSTAR=HSTAR(DML,NOGO)#FORTY/DML/6STAR"6*"
*REMC
*GET FORTY/DML/SCHEMA"1."
*GET FORTY/DML/AREA1"A1"
*GET FORTY/DML/KEY1"K1"
*RUN HSTAR=(STAT)
```

MEMORY CONSIDERATIONS

Under the FORT or FRN system, the maximum memory allowed for compilation is the initial memory plus a maximum of 75K. The amount of memory available may be limited to less by time sharing itself. If the program is too large to run within these limits, a Y1 (X2) compiler abort occurs. The only way to avoid this situation is to reduce the size of the program.

Under the YFORTRAN system, the maximum memory allowed for compilation is the initial memory plus 3K. If this is not enough memory, the "CORE=" option should be used.

RESTRICTIONS ON LOAD USAGE

It is not possible to ready an area for LOAD in time sharing. The FORTRAN DML statement:

```
READY(ALL|REALM=<realm list>,LOAD)
```

is illegal in time sharing. LOAD usage requires special JCL and must be run in batch. This special JCL is described in Appendix E of the <u>DM-IV (FORTRAN) Program's Reference Manual</u>.

TIME SHARING SYSTEM DEFINITIONS AND FILE DESCRIPTION

Definitions

Line Numbers	- Line numbers are required for line sequencing purposes. A line number consists of one to eight numeric characters. (There can be leading blanks, but no embedded blanks.)
Manual Mode	- In manual mode, the line numbers for each line must be entered.
Automatic Mode	- In automatic mode, the system provides the line numbers. They are printed as the build-mode request for input (i.e., the asterisk) is issued. The number is written onto the collector file as a part of the statement.
New File	- A new file is a temporary file created when the command or the response NEW is used. It is assumed the user will build a file which then may be saved, thus creating an old file. A new file is created by a reinitialization of the current file.
Old File	- An old file is a previously built and saved file which is selected with the OLD command and the name of the desired file. The old file is copied onto the current file where it is available for processing or modification.
Current File	- The current file is an assigned temporary file on which a new file is built or the selected old file is copied. Regardless of the intervening commands or subsystem selections, the current file contains the last NEW or OLD selection, with whatever modifications that may have been entered. The modifications are, therefore, temporary until the file is saved by means of the command SAVE. The original old file is not altered until a RESAVE command naming the old file is executed.
Collector File	- The collector file is a transparent temporary file assigned for each log on. All input which is not a recognizable command is gathered onto this file (e.g., numbered statements). Then, depending upon the subsystem, when the file becomes full or a command is typed, the collector file is merged with the current file and the entire current file is edited and sorted if necessary. For example, when the commands RUN, LIST, or SAVE are encountered, and data exists in the collector file, it is merged with the current file in sort order.
Available File Table	- An available file table (AFT) is provided for each time sharing system user at log-on, but ceases to exist after the user is disconnected from the system. This table holds a limited number of file names (currently set at 20) which are entered in the AFT when the files are initially accessed (opened). The AFT is an advantage because ● Files requiring passwords or long catalog/file descriptions may be referenced by file name alone once they have been entered in the table. ● Files used repeatedly remain readily available, thus reducing the overhead time and cost of accessing the file each time.

The following commands place the named permanent files in the AFT:

LIST	filename(s)
OLD	filename(s)
SAVE/RESAVE	filename(s)
GET	filename(s)
PRINT	filename(s)
PERM	tempfile, filename

Because the AFT is of limited length, it can become full. If this happens and a command is given which requires a new filename to be placed in the AFT, the command subsystem will print an error message indicating that the AFT is full. At this point, any files that are no longer needed must be removed from the AFT in order to continue. The STATUS FILES command produces a listing of all of the files in the AFT, and the REMOVE command can be used to remove specified files from the AFT. The files are not purged or altered in any way; only the name is removed from the AFT and the file is set non-busy.

NOTE: When compiling and executing programs that contain FORTRAN DML statements, the following I-D-S/II files can be specified in the #fe list, but must be in the AFT.

- At compile time:
  - The validated subschema file (the 6* file from subschema validation and translation) under the alternate name "6*"
- At execute time:
  - The schema file (the 1* file from the schema translation) under the alternate name "1.", or under the alternate file code specified in the schema DMCL
  - The data base area and key files under the alternate names specified in the schema DMCL
  - If the STAT option is selected, the file code specified for the statistics must be under the alternate name "P."

Description of Files

FILE SPECIFICATION

When a permanent file is used in the time sharing system, reference to it must be specified as one of the following formats.

1. filename — where the filename only is required
2. filedescr — where the full file description may be used, in any of the following formats:
   a. filename
   b. filename$password
   c. userid/catalog$password...
      /catalog$password/filename$password

If a required password is stored incorrectly or not given, the system will explicitly ask for the proper password.

If the file was previously opened (e.g., with a GET), only the filename needs to be given regardless of its full description. If the requested file is not already open, it must emanate directly from the master catalog (quick-access type file) in order for Formats 1 and 2 to be applicable.

Where desired-permissions and/or an alternate-name are applicable, a specific format must be used.

Format

```
filedescr["altname"],permissions
```

where: altname — may be a valid file name (one to eight characters), enclosed in quotation marks.

permissions — may be any one or combination of the following verbs separated by commas:

READ (or R)

WRITE (or W)

EXECUTE (or E)

APPEND (or A)

Where a desired permissions specification is required, a null permissions field implies READ and WRITE permissions (i.e., the default interpretation for desired permissions is R,W).

If a file segment specification of the form (i,j), where i and j are line numbers, is given in addition to desired permissions and/or an alternate name, it must appear last in the specification string.

Format

```
filedescr["altname"],permissions(i,j)
```

Examples

OLD	FIL1$GOGO,R
SAVE	/CAT1CAT2$MAYI/FIL0$HERE
LIST	FILE2$HOHO
PURGE	FIL3$ARIZ;FIL4;FIL5$SUN
GET	JJONES/DATACAT/BATCHWRLDFIL"INFILE"

Categories of Files

In the time sharing environment, distinctions are made between permanent files in two separate categories that apply to all files.

- File-access type is a general time sharing, file-system-usage distinction and is not exclusive to FORTRAN
- File mode deals primarily with the kinds of files produced under the FORTRAN system.

FILE-ACCESS TYPES

There are three types of files which are based on the method of creation and subsequent accessing of the file:

1. Quick-access files are permanent files that were automatically created by the system as a result of using the SAVE filename or PERM tempfile command. Quick-access files can also be created under ACCESS if no intermediate catalog structure is specified. This type of file has the following characteristics:

    a. It can be accessed by its creator simply by the filename form of command and, in the case of data files (input or output), is accessed automatically upon execution of a program reference to it (i.e., it does not need to be pre-accessed by a command).

    b. It can be accessed with the READ permission only by any other user who can specify the (creator's user ID/filename).

2. Quick-access files with a password attached are permanent files that were automatically created by the system as a result of the use of a SAVE filename$password command as the first reference to a particular file name. This type of file is the same as the simple, quick-access type described above with the specified password attached. It can be accessed by its creator either by the filename or the filename$password form of commands; in the former case, if $password is omitted, the system explicitly asks for the password. Also, in the case of data files, it is accessed automatically upon execution of a program reference to it, but the system explicitly asks for the password.

3. Nonquick-access files are permanent files that either do not "belong" to the user (i.e., it was created by another user) or do not emanate directly from the master catalog. In the latter case, the file is not completely described by user-id and filename$password and, in general, use was made of the ACCESS subsystem in explicitly creating some or all of the catalog/file strings describing the file.

The nonquick-access type of file can be accessed either with the GET command or with similar extended forms of other commands.

NOTES: 1. A quick-access file for user A is by definition not a quick-access file for any other user.

2. Once a type of file is initially accessed, whether by a GET or any other command, it can then be referred to simply by file name, unless it is explicitly removed from the AFT.

FILE MODES

Four modes of files can be produced under the FORTRAN system.

Mode	Characteristics
ASCII	A linked (sequential) file of variable-length records in ASCII character code (i.e., a file composed of 9-bit character strings).
BCD	A linked (sequential) file of variable-length records in BCD character code (i.e., a file composed of 6-bit character strings).
Binary	A linked (sequential) file of variable-length records in binary.
Random	A random file of fixed-length records in binary.

All files, regardless of the mode, must be explicitly saved by using the SAVE or PERM commands in order to be retained as permanent files. If the specified permanent file does not already exist, it is implicitly created with the correct linked or random characteristic, as required by the file mode. (Linked is the default type of file created.) If, however, the specified permanent file was explicitly created (i.e., normally by use of the Create-File function of the ACCESS subsystem), the user must have been careful to create the file with the random (R) specification if a random-mode file is to be saved or made permanent. This is particularly true for the file specified as the savefile in the RUN statement on which the compiler output is saved. If this is a pre-existent file, it must have previously been created (either implicitly or explicitly) as a random file. (Refer to the TSS General Information manual, for a description of the ACCESS subsystem.)

Time Sharing References

More extensive uses of time sharing are discussed in the following manuals:

Use	Manual
Remote Batch Interface	*Network Processing Supervisor (NPS)*
	Remote Terminal Supervisor (GRTS)
File System Interface	*File Management Supervisor*
Terminal/Batch Interface	*TSS Terminal/Batch Interface Facility*
General Information	*TSS General Information Manual*
Text Editing	*Text Editor*

APPENDIX C

TIME SHARING BASED FORTRAN ERROR MESSAGES

File and Record Control Type Errors

1. GET CODE 5 - File Code

   Record size is zero in record control word

2. PUT CODE 4 - File Code

   Current logical record larger than buffer

3. CLOSE CODE 3 - File Code

   File to be closed is not in chain

4. GET CODE 4 - File Code

   Block serial number error

5. FILE SPACE EXHAUSTED - File Code

   Attempts to "grow" this file have been denied by the Time Sharing System.

6. BACK/FORWARDSPACE ERROR - File Code

   Bad Status returned on DRL FILSP

Compiler Abort

COMPILER ABORTING

This message is printed at terminal followed by DRL ABORT. The compiler abort code is stored into slave prefix cell 0.

RUN Command Error Messages

61 LAST RUN COMMAND NOT PROCESSED

"RUN" not first three characters of input.

CONCATENATION IMPOSSIBLE IF RANDOM

RUN "random file;" random file illegal.

LINE NO. INTERVAL ILLEGAL IF NOT ASCII

Line number interval specified for other than type 5 or 6 ASCII.

NOT IN RECOGNIZABLE FORMAT

The input file specified is not legal as compiler or loader input.

MULTIPLE ALTER FILES NOT PERMITTED

Only one alter file (A*) is permitted.

SAVE FILE(S) CANNOT BE SPECIFIED

"RUN HSTAR =; save file" is illegal.

ILLEGAL DELIMITER IMMEDIATELY FOLLOWING"="

Delimiter is not semicolon, comma, left parenthesis, pound sign, or carriage return.

MUST BE RANDOM TO SAVE H*

RUN fs = fh, where fh is not a random file.

MUST BE LINKED TO SAVE C*

RUN fs = fh; fc, where fc is not a linked file.

ILLEGAL OPTION -- xxxx

The compiler/loader option indicated by xxxx is illegal.

ILLEGAL DELIMITER FOLLOWING RUN OPTION "xxxx"

Delimiter must be comma or right parenthesis.

ILLEGAL NAME = SPECIFICATION

Illegal character in name in NAME = option.

USER LIBRARIES EXPECTED

ULIB option specified but no user libraries specified.

USER LIBRARIES NOT EXPECTED

ULIB option not specified but user libraries designated.

TOO MANY USER LIBRARIES SPECIFIED

Maximum of nine user libraries can be specified.

TOO MANY TTY FILE CODES

Maximum of ten terminal file codes can be specified.

LOGICAL FILE CODE NON-NUMERIC OR >63

FORTRAN File codes can range from 1-63.

TOO MANY FILES REQ'D FOR EXECUTION

Maximum of 20 files can be specified.

TEST FILE HAS NOT BEEN ACCESSED

TEST option specified but appropriate ** test compiler has not been accessed.

066 - SPAWN UNSUCCESSFUL--STATUS n

Unsuccessful status returned from derail TASK, where n is equal to

1 - undefined file
2 - no SNUMB available
3 - duplicate SNUMB
4 - no program number available
5 - activity name undefined
6 - illegal user limit (time,size, etc.)
7 - bad status on *J read or write

Refer to <u>TSS System Programmer's Reference Manual</u> for information on derail TASK.

CANNOT LOCATE MAIN PROGRAM IN LOAD FILE

The name of the main program cannot be found in the catalog block of the H* file.

<50> WORK FILE -- FILE TABLE FULL

An attempt to define a temporary work file (B*,R*,*J,etc.) has failed; AFT is full.

<50> WORK FILE -- SYSTEM TEMP. LOADED

System refuses to allocate a temporary work file through derail DEFIL.

Catalog file string errors - (xxxx = file name):

ILLEGAL DELIMITER IN FIELD FOLLOWING xxxx DESCRIPTION

ILLEGAL CHARACTER IN FIELD FOLLOWING xxxx DESCRIPTION

STRING ELEMENT TOO LONG IN FIELD FOLLOWING xxx DESCRIPTION

ILLEGAL PERMISSIONS IN FIELD FOLLOWING xxxx DESCRIPTION

ALTNAME ILLEGAL IN FIELD FOLLOWING xxxx DESCRIPTION

FILE DESCRIPTION TOO LONG IN FIELD FOLLOWING xxxx DESCRIPTION

NO DATA IN STRING IN FIELD FOLLOWING xxxx DESCRIPTION

File access errors:

```
<50> FILE xxxx -- STATUS nn
<50> FILE xxxx -- I/O ERROR
<50> FILE xxxx -- NO PERMISSION
<50> FILE xxxx -- FILE BUSY
<50> FILE xxxx -- NON-EXISTENT FILE
<50> FILE xxxx -- NO FILE SPACE
<50> FILE xxxx -- INVALID PASSWORD
<50> FILE xxxx -- FILE TABLE FULL
<50> FILE xxxx -- SYSTEM LOADED
<50> FILE xxxx -- ILLEGAL CHAR
```

Reading and writing I/O errors:

```
<51> FILE xxxx -- I/O STATUS nn
<51> WORK FILE -- I/O STATUS nn
```

where nn is status code returned from derail DIO.

RUNL Command Error Messages

FILE NAME MUST BE OBJECT DECK (C*) FILE

The file specified is not an object deck file.

If no C*'s are specified left of the equals sign, the message is:

*SRC MUST BE OBJECT DECK

INCORRECT LINK PHRASE IN RUNL COMMAND

```
For example:  Link(,B) or Link(A,)
              Link(A,B,) or Link (B,C)
              Link(A,,) or Link(,B,)
              Link (   )
```

INCORRECT SYNTAX FOR RUNL COMMAND

Generally, an illegal delimiter has been specified.

H* SAVE FILE NOT SPECIFIED

H* save file must be specified to right of equals sign.

ILLEGAL CHAR(S) IN LINK NAME

Characters must be alphabetic, numeric, and dash.

TOO MANY CHARS IN LINK NAME

More than six characters in link identifier.

028 - READ LINKED FILES ONLY WITH THIS COMMAND

The "PSTR" load map file is random; it must be linked.

SAVE FILE(S) CANNOT BE SPECIFIED

The H* save file appears to the left of the equals sign.

M6 - CALL/RSTR CHECKSUM

The H* save file is not sufficiently large enough (in current size) to contain the bound link/overlay structure.

ADDRESS OUTSIZE OF FILE LIMITS

The H* save file is not sufficiently large enough (in current size) to contain the bound link/overlay structure and an attempt is made to "RUN" the file.

DIAGNOSTIC MESSAGES ISSUED BY TIME SHARING LOADER

All messages are prefixed by either W for warning or F for fatal. The majority of errors are diagnosed as warnings because the user has the ability to hit the break key at any time. Thus, the decision is left to the user to continue or stop.

XXXXXX UNDEFINED

Symbol (XXXXXX) is an undefined SYMREF. DRL ABORT is substituted for all references.

XXXXXX LOADED PREVIOUSLY

SYMDEF (XXXXXX) previously defined in load table.

INCONSISTENT PREFACE FIELD (Deck) (Card)

One of two conditions occur on card number (Card) in deck number (Deck). The conditions are: (1) a SYMREF (type 5) appears with a nonzero size field (bits 0-17) in the preface card; or, (2) a LABELED COMMON (type 6) appears with a zero size field (bits 0-17).

LABELED COMMON XXXXXX - SIZE INCONSISTENT

LABELED COMMON (XXXXXX) defined previously with smaller size. Loading continues using original size.

ILLEGAL CHECKSUM (Deck) (Card)

The checksum on card number (Card) of deck (Deck) does not compare when recalculated. Loading continues.

ILLEGAL BINARY CARD (Deck) (Card)

Card number (Card) of deck (Deck) is not either preface (type 4), binary (type 5), or BCD (type 6). Card is ignored. This message may also appear where a preface or binary card appears out of expected order.

COMMON SIZE INCONSISTENT (Deck) (Card)

Blank common already defined. A subsequent deck is encountered having a larger blank common region specified. The deck is ignored and loading continues.

ILLEGAL LOAD ADDRESS (Deck) (Card)

A calculated storage address falls outside loadable store. The deck is ignored but loading continues.

XXXXXX LOADED PREVIOUSLY, LABELED COMMON ILLEGAL

SYMDEF (XXXXXX) already defined. XXXXXX appearing in current preface record is a Labeled Common. Deck is ignored.

The following diagnostics are preceded by a printout of the record in error and are generally associated with OCTAL correction processing.

NON-OCTAL DIGIT IN LOCATION FIELD

Self explanatory.

FIELD EXCEEDS 12 DIGITS

Twelve octal digits is maximum allowed in word.

ILLEGAL TERMINATOR

Octal field is eliminated incorrectly. Check syntax rules in the General Loader manual.

IC MODIFICATION NOT POSSIBLE

Field requested IC modification ($code). In this case no other modifiers are allowed. Bits 30-35 of the constructed instruction are checked and found to be nonzero.

XXXXXX UNDEFINED LINK ID IS YYYYYY

Where XXXXXX is an object symbol(SYMDEF) name and YYYYYY is a link identifier. Meaning is XXXXXX is an unresolved SYMREF within the bounds of overlay YYYYYY.

XXXXXX UNDEFINED LINK ID

Link identifier XXXXXX is being used to define an origin point for the next overlay. It has yet been undefined.

XXXXXX NOT LINK ID

Symbol XXXXXX appearing here as a link identifier has been used and entered into the load table previously as another type symbol.

LINK ID XXXXXX USED PREVIOUSLY

The identifier, XXXXXX, for the upcoming overlay has been previously entered in the load table as a link identifier.

Fatal Diagnostics

EOF READING BINARY (Deck) (Card)

Unexpected EOF while reading binary, identification of last record read is supplied.

ENTRY NOT FOUND

Primary entry name (...... or first primary SYMDEF) was not found in load table. Diagnostic may also appear when subroutine .SETU. is not found.

H* TOO SMALL, TOTAL BLOCKS NEEDED xxxx

File specified as save file (H*) not large enough to hold program.

REQUEST FOR MORE STORE TO EXPAND LOAD TABLE - DENIED

A request for 1K to be added at the upper address end of the load table was denied by the system. Loading terminates. Suggest user rerun job.

REQUEST FOR MORE STORE TO EXPAND PROGRAM - DENIED

A request to expand memory size for object program denied by the system. Suggest user rerun job.

ILLEGAL STATUS WHILE READING (File)

Only status accepted other than EOF is ready.

BLOCK SERIAL ERROR READING (File)

Block number in file (File) does not agree with expected number.

LIBRARY SEARCH TABLE EXCEEDED

Table used to collect pointers into random library has been exceeded. Table size is arbitrarily set at 200.

REQUEST FOR MORE STORE TO EXPAND LOAD TABLE - DENIED

Addmem request denied. Probable need for increasing TSS memory size.

FORTRAN Compiler Aborts

NOTE: The abort code Y1 is always displayed as the reason code for any abort. The specific code is contained in the upper 18 bits of the Q-register, or in cell 0 of the ABRT file when a time sharing DRL abort occurs. (The reason codes follow the abort code Y1 in parentheses below.)

Y1 (X1)	Compiler space management module has unsuccessfully attempted to allocate contiguous memory block for internal table. Rerun with DUMP option and $ SYSOUT card for file code *F. Return dump to Honeywell Field Support.
Y1 (X2)	Compiler has attempted to execute request for additional memory more than 10 consecutive times (initial memory plus maximum of 30K). Increase allocation via $ LIMITS card or via "CORE=" option on TSS RUN. This error could also be caused by a recursive arithmetic statement function.
Y1 (X3)	GCOS has denied compiler request for additional memory for internal tables. Increase allocation via $ LIMITS card or via "CORE=" option on TSS RUN.
Y1 (P3)	Expression being handled has tree structure depth greater than 64. Expression must be divided.
Y1 (P4)	Unrecoverable error occurred in code generator; error message prints following source statement causing abort. Rerun with DUMP option and $ SYSOUT card for file code *F. Return dump to Honeywell Field Support.
Y1 (X4)	The ASCII option was not explicitly specified on the $ FORTRAN or $ FORTY control card for a DPS all-ASCII system.
Y1 (X5)	END; cannot be specified as the first statement of a multi-statement line.

Execution Aborts

LK	No $ ENTRY card for this link.
Q1	Logical Unit Table overflow.
Q2	Missing Logical Unit Table, may be caused by a missing or misplaced $ OPTION control card.
Q3	No space for Logical Unit 6 Buffer.
Q4	Machine error or unexpected error to FORTRAN compiler.
Q5	FXEM told to take an alternate return but an alternate return name was not supplied.
Q6	Termination of object program execution via FXEM (FORTRAN Execution Error Monitor).

APPENDIX D

SYSTEM CHARACTERISTICS

The compiler compiles all FORTRAN programs originating from batch or time sharing, local or remote. A collection of source programs can be compiled, some through time sharing, some through batch, and the object modules combined for execution in either environment.

SOURCE COMPATIBILITY

The source files processed by FORTRAN can be any combination of the following:

1. A BCD card image file, with or without alters.
2. A COMDK file, with or without alters.
3. A time sharing ASCII file.
4. A formatted BCD line image file, with or without slew controls.
5. A formatted ASCII line image file, with or without slew controls.

FILE CONTENTS

The source file contents can be in standard source format or in the relaxed "free-form" format especially suitable in time sharing, with or without line numbers. Files in any of the accepted file or source formats may be compiled without conversion, from either batch or time sharing.

COMPILATION of SUBPROGRAMS

Many compilations can be done within one activity provided that the options are the same for a collection of subprograms. The batch user stacks the source programs, back to back, behind one compiler call card. The time sharing user lists a series of source files to be compiled or provides multiple subprograms in a source file. To the compiler there is one input file, S*, and source programs are separated by END statements.

For larger programs requiring more memory to compile than that allocated to an activity, the compiler "grows" in an attempt to satisfy this need. Normally a satisfactory compilation will result; however, the operating system may deny more memory to the compiler. The user is warned, in any event, that the $ LIMITS card should be changed for subsequent recompilations.

ERROR DETECTION and DIAGNOSTICS

In batch mode, diagnostics are generated inline as part of the source listing report (LSTIN) wherever possible, following the line in error. If this report is being suppressed via the NLSTIN option, lines having no errors are not printed, but lines for which a diagnostic is being generated are displayed. In the time sharing mode, the error message is printed along with the source line location of the error. If the line numbers of the source file are not sequentially increased by one, the actual line number is that of the first executable statement whose line number is less than the line number printed.

Format

```
*****S    nnnn    text
```

where: S is a severity
nnnn is an error identification code
text is the diagnostic message.

Three severity codes are

Code	Meaning
W	This is a warning message only.
F	This is a fatal diagnostic; any subsequent execution activity is deleted.
T	This is a termination diagnostic; this compilation and any subsequent execution activity are deleted.

If only warning diagnostics are printed for a given compilation, these diagnostics can be suppressed by using the NWARN option.

The correspondence of error codes with the compiler module detecting the error is

Error Number	Compiler Module
1- 199	Executive
200- 299	Phase 2
300- 399	Phase 3
400- 499	Phase 4
1000-1499	Phase 1

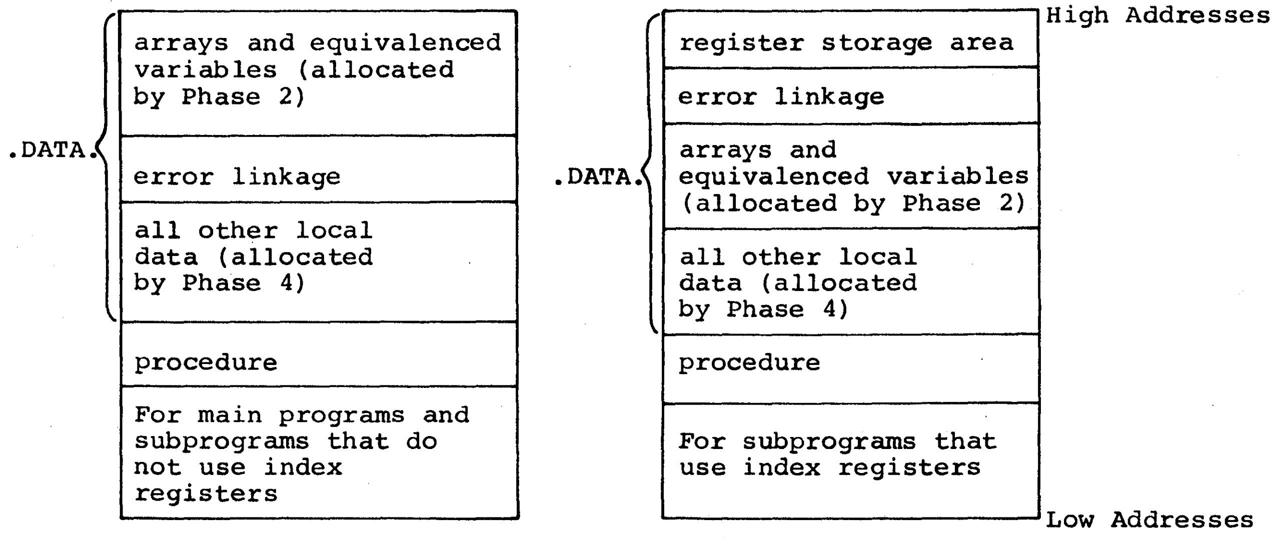
COMPILER CONSTRUCTION

The compiler is written in and generates object modules in "pure procedure". .DATA. space and instruction space are clearly separated and the instruction space remains constant over the life of the execution process.

ALLOCATION of STORAGE

Storage allocation for the object program is done in two phases of the compiler. Phase 2 allocates storage for arrays, equivalenced variables, and all data that is in blank or labeled common. Phase 4 allocates storage for local scalars, namelists, switch variables, and compiler generated constants and temporary data. Phase 4 also allocates space and generates code for the procedure.

All variables (except those in blank or labeled common), constants, and temporary data are allocated to the local data storage area .DATA. which is treated by the loader as a local labeled common. Figure D-1 shows the storage layout for two typical low-loaded FORTRAN object programs.



.DATA.
arrays and equivalenced variables (allocated by Phase 2)
error linkage
all other local data (allocated by Phase 4)
procedure
For main programs and subprograms that do not use index registers
.DATA.
register storage area
error linkage
arrays and equivalenced variables (allocated by Phase 2)
all other local data (allocated by Phase 4)
procedure
For subprograms that use index registers
High Addresses
Low Addresses

Figure D-1. Storage Allocation for Object Programs

ASCII Standard System Format Files

This file format is common for batch and time sharing users as are the library routines that read and write them. This common procedure for batch and time sharing guarantees symmetry and compatibility. The file format for ASCII conforms with the File and Record Control rules for "standard system format" because every line is recorded as a logical record.

PERFORMANCE

The performance objective of the FORTRAN compiler is to provide a fast compiler that can generate fast executing object modules. It is generally realized that the more analysis done to improve the efficiency of the object module, the greater the time spent in compilation. Consequently, this analysis is subdivided into two classes:

1. Local Optimization (LO) - the analysis generally done at the statement level.

2. Global Optimization (GO) - the analysis done over many statements, i.e., program blocks as defined by the ANSI FORTRAN standard.

To give the user some control over the balance between compilation and object efficiency it was decided to collect the GO analysis into a unique compiler phase, callable by option. LO analysis is always performed.

Local Optimization

Following are some of the object efficiency functions done on a local basis:

1. Logical expressions are sorted so that shorter alternative passages are executed first, and evaluation ceases as soon as the true/false state has been determined.

2. Subscript expressions may be register contained, eliminating multiple computations.

3. Constants may be register contained across statements.

4. Multiplication and division by powers of two are performed using shift or exponent register operations with the exception of integer operations.

5. Constant arithmetic is done at compile time.

6. Many special operator/operand relationships are recognized to capitalize on the machine instruction set. Examples are I*1, I**1, I=0, I=I+1, I=I+J.

7. Where possible, operations involving constants make use of the DU, and DL modifiers.

8. Where there is no redefinition of a scalar dummy argument within a subprogram, the value of that argument is stored locally. This eliminates an indirect cycle for each reference to that argument.

Compilation Performance

Compile speed is also a function of the properties of the program being compiled and directly related to the options selected on the $ FORTY or $ FORTRAN control card. The Global Optimization compiler phase increases compile time for most programs by a factor of about twenty percent. For many programs the specification of LSTOU doubles the compile time. Measured in statements per minute, the compilation rate improves with larger programs. The smaller the program the greater the effect of the basic overhead to start compilation, step through the phases, and terminate. Binary and compressed decks, source listing, storage maps, cross reference reports, etc. decrease the compilation rates.

APPENDIX E

FORTRAN EXECUTION ERROR MONITOR EXAMPLES

This appendix illustrates the use of the FORTRAN Execution Error Monitor (FXEM) in both time sharing and batch modes, utilizing CALL FXEM.

Figure E-1 lists a program and its execution in time sharing. The trace shown indicates that error number 61 (see Table 6-5) occurred in subroutine SUB2 at line 320, that SUB2 had been called from subroutine SUB1 at line 210, and that SUB1 had been called from the main program (......) at line 110. The message "Argument 0" indicates the reason for aborting the execution of the program via the call to FXEM.

Figure E-2 lists the program of Figure E-1 but shows its execution in batch. The trace shown indicates that error number 61 (see Table 6-5) occurred in Subroutine SUB2 at alter number 3. The octal value of the three arguments used for CALL FXEM are also shown. The trace also shows that SUB2 had been called from subroutine SUB1 at alter number 2, along with the octal representation for the floating-point argument (-20). SUB1 was called from the main program (......) at alter number 2 with the same argument. "Argument < 0" indicates the reason for aborting the program via the call to FXEM.

```
100       A = -2.0
110       CALL SUB1(A)
120       STOP
130       END
200       SUBROUTINE SUB1(B)
210       CALL SUB2(B)
220       RETURN
230       END
300       SUBROUTINE SUB2(C)
310       IF ( C .GT. 0. ) RETURN
320       CALL FXEM (61, "ARGUMENT < 0",3)
330       STOP
340       END

ready

*RUN
***PROG.  L#   (ERR #61)
SUB2      320
SUB1      210
......    110
ARGUMENT < 0
abort code    Q6


*
```

Figure E-1. FXEM Example in Time Sharing Mode

```
2723T 01  02-20-75   13.623                                                                              LABEL ......
                   1            A = -2.0
                   2            CALL SUB1(A)
                   3            STOP
                   4            END

2723T 01  02-20-75   13.624                                                                              LABEL SUB1
                   1            SUBROUTINE SUB1(B)
                   2            CALL SUB2(B)
                   3            RETURN
                   4            END

2723T 01  02-20-75   13.624                                                                              LABEL SUB2
                   1            SUBROUTINE SUB2(C)
                   2            IF ( C .GT. 0. ) RETURN
                   3            CALL FXEM (61, "ARGUMENT < 0",2)
                   4            STOP
                   5            END

<*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*>
 ERROR #61:  TRACE OF CALLS IN REVERSE ORDER
     CALLING         ID      ABSOLUTE       ARGUMENT          ARGUMENT          ARGUMENT          ARGUMENT
     ROUTINE         #       LOCATION          #1                #2                #3                #4
     SUB2             3      017730      000000000075      215127644425      000000000002
     SUB1             2      017752      003000000000
     ......           2      017770      003000000000
ARGUMENT < 0
<*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*><*>
```

Figure E-2. FXEM Example in Batch Mode

APPENDIX F

FORTRAN DEBUGGING SYSTEM

The FORTRAN debugging system (FDS) is a comprehensive monitoring system that provides a dynamic interactive debugging facility, a symbolic dump facility, an automatic subprogram timing measurement system, and post-execution wrapup procedures.

NOTE: The initial version of this debugging system was developed by Bell Laboratories.

FDS CAPABILITIES

The FORTRAN debugging system provides the following capabilities:

1. All output data produced by the debugging system uses notation similar to the FORTRAN source program being debugged. Analysis of this data requires only the knowledge necessary to prepare the source program.
2. The debugging requests are similar in syntactic construction to the FORTRAN language that is being debugged.
3. Unless it is invoked, the debugging system does not affect execution time or memory requirements.
4. All of the debugging aids and measurement tools are available in both the batch and time sharing environments of the operating system (GCOS).

INVOKING THE FORTRAN DEBUGGING SYSTEM

The FDS is an optional feature rather than a default function and is invoked at the discretion of the user.

Batch Mode

The FORTRAN debugging system is invoked in the batch mode by including the FDS option in the operand field on the $ FORTY or $ FORTRAN control card.

Time Sharing Mode

The FORTRAN debugging system is invoked in the time sharing mode by including the FDS option with the RUN command on the terminal:

```
RUN=(FDS)
```

DYNAMIC DEBUGGING FACILITY

The dynamic debugging module is named FDEBUG.

In the batch mode, FDEBUG is called into execution when:

1. A CALL FDEBUG statement is encountered during the execution of a FORTRAN source program.

   ```
   CALL FDEBUG(di,do)
   ```

   where: di represents the file designator from which the debugging requests are to be read.

   do represents the file designator on which the debugging output is to be written.

   If di is omitted or is not a positive number, the requests are read from file designator 44. If do is omitted or is not a positive number, the debugging output data is written to file designator 6.

2. File designator 44 is present in the EXECUTE (or RLHS or PROGRAM) activity. In this case, the FDEBUG module is entered before the execution of the main FORTRAN program is initiated; it reads any debugging requests from file designator 44 until an end-of-file or FDS RETURN request is encountered, whereupon control returns to the main program.

3. An FDS PAUSE request (breakpoint) is encountered during the execution of the program.

   NOTE: The FDS PAUSE request is defined below in the Debugging Requests paragraph; it has no relationship to the FORTRAN PAUSE statement described in Section III.

In the time sharing mode, FDEBUG is called into execution when:

1. A CALL FDEBUG statement is encountered during the execution of a FORTRAN source program.

   ```
   CALL FDEBUG(di,do)
   ```

   where: di represents the file designator from which the debugging requests are to be read.

   do represents the file designator on which the debugging output is to be written.

   If di is omitted or is not a positive number, the debugging requests are read from the terminal. If do is omitted or is not a positive number, the debugging output data is written to the terminal.

2. The FDS option is specified with the RUN command. In this case, the FDEBUG module is entered before the execution of the main program is initiated. It reads any debugging requests from the terminal until an end-of-file or FDS RETURN request is encountered, whereupon control returns to the main program.

3. An abnormal termination (abort or break) is encountered and no preventive action has been taken. The FDEBUG module is called from the wrapup procedures; these procedures are described later in this appendix.

4. An FDS PAUSE request (breakpoint) is encountered during the execution of the program.

FDEBUG Entry Messages

In the batch mode, messages that indicate the method by which FDEBUG is invoked are printed on the execution report. The 'name' used in the messages designates the name of the program in control when FDEBUG is engaged.

1. If file designator 44 is present, FDEBUG is always entered before the program is initiated. The message is:

```
FDEBUG
```

2. If the method of entry is via a CALL FDEBUG statement in the source program, the message is:

```
FDEBUG  CALLED  FROM  name  IN  LINE  lineno
```

3. If an FDS PAUSE request (breakpoint) is encountered during the execution of the program, the message is:

```
FDEBUG:  PAUSE  IN  name  AT  STMT # n
```

In the time sharing mode, messages that indicate the method by which FDEBUG is invoked are printed on the terminal:

1. When the FDS option is used with the RUN command, FDEBUG is entered before the program is initiated. The message is:

```
FDEBUG
```

2. If the method of entry is via a CALL FDEBUG statement in the source program, the message is:

```
FDEBUG  CALLED  FROM  name  IN  LINE  lineno
```

3. If a program terminates abnormally, FDEBUG prints

```
FDEBUG  CALLED  FROM  name
```

following the termination message.

4. An interrupt (break) will cause the FDEBUG module to be re-entered and the following message is printed:

```
FDEBUG:  BREAK  IN  name
```

When FDEBUG regains control, it reads the input from the terminal to obtain the debugging requests.

5. If an FDS PAUSE request (breakpoint) is encountered during the execution of the program, the message is:

```
FDEBUG:  PAUSE  IN  name  AT  STMT  # n
```

Debugging Requests

The following conventions apply to the descriptions of the debugging requests:

a. The first two characters of the request (underlined) can be used as the abbreviated form of the request.

b. Whenever the term 'expr' is shown, it represents an expression that is formed from variables or array elements, constants, and the operators +, -, *, /, **, .EQ., .NE., .LE., .LT., .GE., .GT., .AND., .OR., and .NOT. . The exponent following ** must be type INTEGER. No function references are allowed.

c. If the request is preceded by 'n', that request is inserted (implanted for interpretation during execution) at the location of the FORTRAN statement label 'n'.

The names and descriptions of the FDEBUG requests are listed below:

n CALL name(expr,expr,...)

The CALL request allows user-supplied or system-supplied subroutines to be called; a maximum of ten arguments can be supplied. Statement label 'n' is optional. A CALL FDEBUG request cannot be inserted. Subroutines that are to be called from an inserted CALL request cannot contain CALL FDEBUG statements in the source program, nor can they have FDEBUG requests inserted into them. If FORTRAN input-output statements are contained in the called subroutine, the CALL request should not be invoked if FDS was entered by pressing the interrupt (break) key while the FORTRAN program was performing input-output operations.

If the preceding restrictions are violated and the named subroutine has previously invoked FDEBUG, the interpretation of the illegal CALI request causes a RECURSIVE CALL error message to be printed and the request is ignored. Otherwise, the results of interpreting the CALI request are unpredictable. The results are usually an abnormal program termination or, in time sharing, a loop that can be resolvec only by entering a DONE, QUIT, or STOP request. (It may be necessar to press the interrupt key to invoke FDEBUG to accept an inpu request.)

n CONTINUE

The CONTINUE request causes all debugging requests inserted a statement label 'n' to be removed. If statement label 'n' is omitted the request is ignored.

n DONE

Causes the execution of the program to be terminated. Statement labe 'n' is optional.

FUNCTION name

An identifier request; this request identifies FUNCTION 'name' as the program unit in which subsequent requests will be interpreted until another identifier request is encountered. When FDEBUG is invoked, the default identification in which subsequent requests are interpreted is that of the FORTRAN program unit currently in control.

n GOTO label

This request causes an unconditional transfer to the indicated source statement label to be inserted at statement label 'n'. If statement label 'n' is omitted, the request is ignored and an error message is printed.

n IF(expr) request

The logical expression 'expr' is evaluated. If the value is .TRUE., the debugging request will be interpreted. If statement label 'n' is omitted, the request is ignored.

MAIN

An identifier request; this request identifies the main program as the program unit in which subsequent requests are interpreted until another identifier request is encountered. When FDEBUG is invoked, the default identification in which subsequent requests are interpreted is that of the FORTRAN program unit currently in control.

n PAUSE

The PAUSE request causes a breakpoint to be inserted at statement label 'n'. Whenever the breakpoint is encountered during program execution, the FDEBUG module is invoked. If statement label 'n' is omitted, the request is ignored.

n PRINT expr,expr...

The PRINT request causes the values of the expressions 'expr' to be printed in the appropriate format. If a nonsubscripted array name appears in 'expr', only the value of the first element of the array is printed. Statement label 'n' is optional.

n QUIT

Causes the execution of the program to be terminated. Statement label 'n' is optional.

RETURN

The RETURN request causes the FDEBUG module to return control to the program that is being executed. Control is always returned to the point where FDEBUG was entered.

SHOW

The SHOW request displays the location and text of all currently inserted requests in all program units.

n   STOP

Causes the execution of the program to be terminated. Statement label 'n' is optional.

SUBROUTINE name

An identifier request; this request identifies SUBROUTINE 'name' as the program unit in which subsequent requests will be interpreted until another identifier request is encountered. When FDEBUG is invoked, the default identification in which subsequent requests are interpreted is that of the FORTRAN program unit currently in control.

n   var=expr

This request causes the value of the scalar variable or array element 'var' to be set to the value of the expression 'expr'. The rules of allowable assignment apply except that a CHARACTER expression may be assigned to an INTEGER. Statement label 'n' is optional.

!text

This request causes all text that follows the exclamation point to be transmitted to the time sharing system as a command to be executed. Time sharing system commands that are applicable at the system level are accepted. This request is not available in the batch mode of operation. If a statement label 'n' is included, a SYNTAX ERROR error message is printed.

Debugging Request Execution

The execution of debugging requests can be accomplished by two methods:

1. If a debugging request is preceded by statement label 'n', FDEBUG inserts the request at the indicated executable FORTRAN source statement. When the program is executed, the FDEBUG requests are interpreted in the order of insertion before the original source statement is executed.

2. If a debugging request is not preceded by statement label 'n', FDEBUG interprets the request immediately.

FDEBUG Error Messages

The following error messages are produced by the FDEBUG module:

Error Message	Description
ANSWER PROMPT WITH PROGRAM INPUT	The BREAK key was pressed while data was being entered at the terminal, or FDEBUG was called just prior to program input and a RETURN request is received. Respond with program input.
BREAKPOINT OVERWRITTEN	An inserted request in object code has been overwritten.
( ) - CHARACTER SIZE ILLEGAL	An adjustable character variable size is out of range.
CONSTANT TOO BIG OR TOO SMALL	A constant contained in an expression that is used in an FDEBUG request is either too large or too small.
( ) - ENTRY NOT FOUND	A CALL request was given to FDEBUG but the entry point to the subroutine could not be found.
( ) - ILLEGAL ADDRESS	An attempt was made to reference a dummy argument that has been passed incorrectly to a subprogram.
ILLEGAL TYPE CONVERSION/ COMBINATION	An attempt was made to assign data of incompatible types or to combine incompatible data types with an operator.
INTEGER OR REAL TOO LARGE	An integer or real number used in an FDEBUG request was too large to process.
LABEL NOT ALLOWED	An FDEBUG request has a label 'n', but a label is not allowed with this request.
LABEL NOT FOUND	A request containing a source program label was given to FDEBUG but the label could not be found.
LABEL REQUIRED	An FDEBUG request requires label 'n' and the label is missing.
NAME NOT FOUND	A CALL request to a subroutine was made and the subroutine name cannot be found.
NESTING LIST OVERWRITTEN	The nesting list, maintained for traceback purposes, has been overlayed in such a manner that the traceback activity cannot be performed. Usually occurs when FDEBUG executes a CALL that performs I/O.
( ) - NOT FOUND	An FDEBUG request specified a name that could not be found.
OUT OF SPACE	Insufficient memory is available to accommodate all inserted FDEBUG requests.
RECURSIVE CALL	A call to FDEBUG was made but FDEBUG is already in control.

Error Message	Description
STACK OVERFLOW	Internal stack overflow; indicates that an expression is too complicated.
STATEMENT TOO COMPLEX	An arithmetic expression used in an FDEBUG request was too complex for the system to evaluate.
SUBPROG NOT FOUND	A subprogram referenced by an FDEBUG request cannot be found.
( ) - SUBSCRIPT OR DIMENSION ILLEGAL	A subscript or adjustable dimension associated with the named variable is out of range.
SYMBOL TABLE EMPTY OR MISSING	Either the FDS option was not used for the compilation of the subprogram or no symbol table could be found for the FDEBUG requests. Use the MAIN, SUBROUTINE, or FUNCTION request and the requests will be processed.
SYMBOL TABLE OVERWRITTEN	The symbol table could not be found or has been overlayed. FDEBUG is unable to process this request.
SYNTAX ERROR	An FDEBUG request is either misspelled, incomplete, or not recognized.
TOO MANY BREAKPOINTS	Too many FDEBUG requests have been inserted.
UNDERFLOW, OVERFLOW OR DIVIDE CHECK	An expression used in an FDEBUG request caused an underflow, overflow, or divide check condition to occur.
WRONG # OF SUBSCRIPTS	An FDEBUG request contained a subscripted variable, but the number of subscripts does not match the number of declared dimensions.

FDS Examples

Examples of the use of the FORTRAN debugging system are presented in Figures F-1, F-2, and F-3. In both the batch mode and the time sharing mode, FDEBUG prints six periods (......) to indicate the main FORTRAN program.

```
10##S,J :,8,16,32
20$:IDENT
30$:OPTION:FORTRAN
40$:FORTY:NFORM,NLNO,FDS
50 A=1.0; B=1.0
60 X=2.0; Y=2.0
70 Z=0; ANS=0
80 CALL FDEBUG(44)
90 CALL SUMF(A,B,ANS)
100 WRITE(6,25)A,B,ANS
110 25 FORMAT(3F8.2)
120 CALL FDEBUG(46)
130 STOP;END
140 SUBROUTINE SUMF(ZA,ZB,ZANS)
150 ZANS=ZA+ZB
160 52 CONTINUE
170 RETURN;END
175$:EXECUTE
180$:DATA:44
190 MAIN
200 RETURN
210 MAIN
220 CALL SUMF(X,Y,Z)
230 PRINT X,Y,Z
240 SU SUMF
250 52 PR,ZA,ZB,ZANS
260 52 IF(ZANS.EQ.2.0)PR,ZA,ZB
270 SHOW
280 RETURN
290$:DATA:46
300 CALL FDUMP
310 RETURN
320$:ENDJOB
```

OUTPUT OF RUN

```
 1  FDEBUG
 2  FDEBUG CALLED FROM ...... IN LINE 4
 3  X =2.,   Y =2.,   Z =4.
 4  SUMF
 5     52      PR,ZA,ZB,ZANS
 6             IF(ZANS.EQ.2.0)PR,ZA,ZB
 7  ZA = 1.,   ZB = 1.,   ZANS = 2.
 8  ZA = 1.,   ZB = 1.
 9     1.00    1.00    2.00
10  FDEBUG CALLED FROM ...... IN LINE 8
11  FDUMP CALLED FROM ...... IN LINE NUMBER   1
12  SUBPROGRAM ......
13  A          1.0000000E 00
14  B          1.0000000E 00
15  X          2.0000000E 00
16  Y          2.0000000E 00
17  Z          4.0000000E 00
18  ANS        2.0000000E 00
19  FDUMP COMPLETE
```

Figure F-1. FDS Example in the Batch Mode

In the batch mode example described in Figure F-1, file designators 44 and 46 are used for the CALL FDEBUG statements.

The FDEBUG module is entered before program execution. For this reason, the first two requests on file 44 are MAIN and RETURN. If desired, additional FDEBUG requests can also be entered at this location.

The next time the FDEBUG module is entered is when the CALL FDEBUG(44) statement is executed at line 80. On file 44, the FDEBUG CALL request is demonstrated by calling a user-supplied subroutine and then printing the variables X, Y, and Z.

Two FDEBUG requests, PRINT (PR) and IF, are then inserted in statement label 52 of the subroutine named SUMF. These two requests will be executed whenever SUMF is called and can be removed by using a CONTINUE request.

The SHOW request at line 270 causes lines 4, 5, and 6 of the output to be printed during program execution. Control is then returned to the calling program. Lines 7 and 8 of the output contain the results of the PRINT and IF requests inserted in the subroutine SUMF.

The FDEBUG module is next entered when the CALL FDEBUG(46) statement at line 120 is executed. The only request contained on file 46 is CALL FDUMP. Lines 11 through 19 of the output contain the results of the FDUMP routine.

Figure F-2 illustrates the procedure for using FDEBUG in the batch mode with linked overlays.

The FDEBUG module is first entered before program execution but the only request interpreted on file 44 is the RETURN request.

The only explicit call to the FDEBUG module occurs in line 70. Two IF requests are inserted at statement label 1 in the subroutine (SU) LODLNK. Control is then returned to the main program.

NOTE: Refer to "Debugging Linked Overlay Programs" in this appendix for information concerning the LODLNK subroutine.

When the CALL LLINK("ASUBA") statement is executed, FDEBUG is entered since the PAUSE request is inserted in the LODLNK subroutine. The SU SUBA instruction establishes subroutine SUBA as the context for the next two requests. Note that these two requests are inserted at statement label 40 in subroutine SUBA.

The same procedure is followed for the CALL LINK("BSUBB") statement. The FDEBUG module is again entered and two FDEBUG requests are inserted at statement label 45 in subroutine SUBB. The results of inserting these requests in subroutines SUBA and SUBB are shown in the output printed from the run.

```
10##S,J :,8,16,32
20$:IDENT
30$:OPTION:FORTRAN
40$:FORTY:NFORM,NLNO,FDS
50 WRITE (6,15)
60 15 FORMAT(14H THIS IS MAIN)
70 CALL FDEBUG(44)
80 CALL LLINK("ASUBA")
90 CALL SUBA
100 CALL LINK("BSUBB")
110 STOP;END
120$:LINK:ASUBA
130$:FORTY:NFORM,NLNO,FDS
140 SUBROUTINE SUBA
150 40 WRITE(6,26)
160 41 WRITE(6,26)
170 26 FORMAT(14H THIS IS LINKA)
180 27 CONTINUE
190 RETURN;END
200$:LINK:BSUBB,ASUBA
210$:ENTRY:SUBB
220$:FORTY:NFORM,NLNO,FDS
230 SUBROUTINE SUBB
240 45 WRITE(6,28)
250 46 WRITE(6,28)
260 28 FORMAT(14H THIS IS LINKB)
270 29 CONTINUE
280 RETURN;END
290$:EXECUTE:DUMP
300$:DATA:44
310 MAIN
320 RETURN
330 SU LODLNK
340 1 IF(LINK.EQ."ASUBA")PAUSE
350 1 IF(LINK.EQ."BSUBB")PAUSE
360 RETURN
370 SU SUBA
375 40 PRINT, "HI FROM LINKA"
380 40 GOTO 41
390 RETURN
400 SU SUBB
405 45 PRINT,"HI FROM LINKB"
410 45 GOTO 46
420 RETURN
430$:ENDJOB
```

<u>OUTPUT OF RUN</u>

```
FDEBUG
THIS IS MAIN
FDEBUG CALLED FROM ...... IN LINE 3

FDEBUG: PAUSE IN LODLNK AT STMT # 1
"HI FROM LINKA      "
THIS IS LINKA

FDEBUG: PAUSE IN LODLNK AT STMT # 1
"HI FROM LINKB      "
THIS IS LINKB
```

Figure F-2. FDS Example in the Batch Mode with Linked Overlays

```
010 I=10
015 CALL FDEBUG(44)
020 PRINT,"HELLO FROM MAIN"
030 CALL SUBA
040 CALL SUBB
050 5 STOP;END
060 SUBROUTINE SUBA
070 PRINT,"HELLO FROM SUBA"
080 ISUB=1
090 10 RETURN;END
100 SUBROUTINE SUBB
110 PRINT,"HELLO FROM SUBB"
120 ISUB=2
130 20 RETURN;END
```

Terminal Input

```
FRN=(FDS)
```

```
 1  FDEBUG
 2  ?RETURN
 3  FDEBUG CALLED FROM ...... IN LINE 15
 4  I = 10
 5  SUBA
 6      10   IF(ISUB.EQ.1)PRINT,"HI FROM A"
 7  HELLO FROM MAIN
 8  HELLO FROM SUBA
 9  "HI FROM A    "
10  HELLO FROM SUBB
11  ISUB = 1000
12  "HI FROM SUBB"
```

Output from Program and FDEBUG

```
MAIN
PRINT,I
SUBROUTINE SUBA
10 IF(ISUB.EQ.1)PRINT,"HI FROM A"
SHOW
SUBROUTINE SUBB
20 IF(ISUB.EQ.2)ISUB=1000
20 PR ISUB
20 PRINT,"HI FROM SUBB"
RE
```

FDS Requests on File 44

Figure F-3. FDS Example in the Time Sharing Mode

Figure F-3 illustrates the procedure for using the FORTRAN debugging system in the time sharing mode.

The FDEBUG module is entered before program execution and control is given to the terminal. The message FDEBUG is displayed on line 1. Whenever FDEBUG expects terminal input, a question mark (?) or equal sign (=) is displayed on the terminal (line 2 of the terminal output). Since no terminal commands are required, the terminal operator enters a RETURN request following the question mark.

The FDEBUG module is next entered when the CALL FDEBUG(44) statement is encountered (line 015 of the terminal input), and the requests contained on file 44 are then interpreted. Following the PRINT request, one request is inserted at statement label 10 in subroutine SUBA and three requests are inserted at statement label 20 in subroutine SUBB. The abbreviated form of the RETURN request (RE) is used on file 44.

The SHOW request on file 44 causes lines 5 and 6 of the terminal output to be printed. Lines 9, 11, and 12 of the terminal output contain the results of interpreting the FDEBUG requests from file 44 in subroutines SUBA and SUBB.

SYMBOLIC DUMP FACILITY

In the batch mode, a symbolic dump can be produced in two ways:

1. A symbolic dump is automatically produced when a program that contains the FDS option on the $ FORTY or $ FORTRAN control card in the job control language terminates abnormally.

2. A symbolic dump can be produced after the FDS has been invoked by specifying the following FORTRAN statement:

   ```
   CALL FDUMP(n,6)
   ```

   The symbolic dump will be written on file designator 6 (defaults to SYSOUT) and will include the 'n' subprograms that were most recently entered into the nesting list.

In the time sharing mode, a symbolic dump can be produced after the FDS has been invoked by entering the following FORTRAN statement at the terminal:

```
CALL FDUMP(n,6)
```

The symbolic dump is displayed on the terminal and includes the 'n' subprograms that were most recently entered into the nesting list.

Example

If a main program calls subprogram A, which in turn calls subprogram B, and subprogram B executes the statement

```
CALL FDUMP(n,6)
```

then: If $n \leq 0$, the call is ignored.

If n = 1, a symbolic dump of subprogram B is written to SYSOUT or displayed on the terminal.

If n = 2, a symbolic dump of subprograms B and A is written to SYSOUT or displayed on the terminal.

If $n \geq 3$, a symbolic dump of subprograms B, A, and also the main program is written to SYSOUT or displayed on the terminal.

If n is omitted, the nesting list will be traced back to the main program.

The format of the dump output begins with a heading that indicates the method by which the dump facility was invoked, followed by a symbolic dump of each subprogram that was contained in the nesting list when the dump was produced.

If the dump facility was invoked using a CALL statement, the heading reads:

```
FDUMP  CALLED  FROM  name  IN  LINE  NUMBER  lineno
```

If the dump facility was invoked from the wrapup procedures after the execution of the program is terminated, the heading reads:

```
FDUMP  CALLED  FROM  WRAPUP
```

After printing the heading, the dump process traces the nesting list back to the main program and prints out the names and values of the variables used in each subprogram. If the dump facility was invoked with a CALL FDUMP statement in the source program, the variables of the subprogram that executed the CALL FDUMP statement appear first in the dump. If the dump is produced as the result of an abnormal program termination, the FORTRAN subprogram that was in control when the termination occurred appears first in the dump.

The following subheading is printed at each level of the nesting list:

```
SUBPROGRAM name1

CALLED  FROM  name2  IN  LINE  NUMBER  lineno
```

where: name1 is the name of the subprogram whose variables will follow.

name2 is the name of the subprogram that is calling name1.

lineno is the line number of the CALL name1 in subprogram name2.

When the main program level is reached, the second line of the subheading is omitted.

The subheading is followed by a listing of the nonsubscripted variables and arrays, together with their associated values. The arrays are printed in column form; the ellipsis (...) is used to indicate successive lines of identical output. The ellipsis is also used to indicate successive columns that are identical.

The format used for each type of variable is listed below:

Integer	I13
Real	1PE15.7
Logical	O13
Complex	1P2E15.7
Double precision	1PD26.18
Character	An

Symbolic Dump Example

An example of a symbolic dump is presented in Figure F-4.

```
FDUMP CALLED FROM WRAPUP
SUBPROGRAM JOE
CALLED FROM ...... IN LINE NUMBER   170
ISTART              0
NPTRS              78
L1                623
L2                545
LL2       000735000000
TIME        1.3800000E-06

TYPE        (*)
      1:           60            78           0           0
      5:            0             0           0           0
    ...
     97:            0             0           0           0

SUBPROGRAM ......
I                  30

A           (*,       1)
      1:     1.0000000E+00  2.0000000E+00  3.0000000E+00  4.0000000E+00
      5:     5.0000000E+00  6.0000000E+00  7.0000000E+00  8.0000000E+00
      9:     0.             0.             0.             0.
    ...
     25:     2.5000000E+01  2.6000000E+01  2.7000000E+01  2.8000000E+01
     29:     2.9000000E+01  7.0000000E+00

A           (*,       2)
      1:     0.             0.             0.             0.
    ...
     29:     0.             8.0000000E+00

* * * *

A           (*,      10)
      1:     0.             0.             0.             0.
    ...
     29:     0.             0.
FDUMP COMPLETE
```

Figure F-4. Example of a Symbolic Dump

Symbolic Dump Messages

The symbolic dump facility provides several error condition messages and a final termination message.

If a symbol table is not available or has been overwritten, or there is not enough memory available in which to load the table, the following message is printed:

```
SYMBOL TABLE NOT AVAILABLE OR OVERWRITTEN
```

When a portion of the nesting list has been overwritten in such a manner that it cannot be traced back to the main program, the dump terminates prematurely and the following message is printed:

```
NESTING LIST OVERWRITTEN, DUMP TERMINATED
```

When a program has called other programs recursively, intentionally or not, the nesting list is caused to loop back on itself. When this condition occurs, the dump terminates prematurely and the following message is printed:

```
CIRCULAR  CALL  DETECTED,  DUMP  TERMINATED
```

An example of this condition occurs when subprogram A calls subprogram B, which in turn calls subprogram C, and subprogram C then calls subprogram A.

The symbolic dump facility will occasionally detect errors in the methods in which arguments are passed to subprograms. One of the following two messages is printed:

```
ERROR  IN  ACT.  ARG.  FOR  (     )

ERROR  IN  ADJ.  DIM.  OR  ACT.  ARG.  FOR  (     )
```

The first message usually occurs for scalar variables and indicates that the address passed to the subprogram for the actual argument is out of range (usually zero). The second message occurs for array variables and indicates that an adjustable dimension has an implausible value.

If no error conditions are encountered during the processing of the dump and the dump has been successfully completed, the following message is printed:

```
FDUMP  COMPLETE
```

CALL FDUMP Examples

Figure F-5 contains an example of an FDS program and a subroutine referenced within the program from which the FDUMP feature is called. An example of the results produced when the CALL FDUMP statement is executed is contained in Figure F-6. Each variable and array in Figure F-6 is displayed by type.

FDS Program

```
INTEGER IARR(5,5)
DIMENSION ARR(3,3)
DO 10 I=1,3
DO 20 J=1,3
ARR(I,J)=I*J
20 CONTINUE
10 CONTINUE
DO 30 I=1,5
DO 40 J=1,5
IARR(I,J)=I+J
40 CONTINUE
30 CONTINUE
A=1.3;B=2.3
CALL CALC(A,B,RESU)
PRINT,A,B,RESU
C=A*B;R=RESU**2
55 CONTINUE
KINDX=KINDX+1
IF (KINDX.LT.5)GO TO 55
STOP;END
```

Figure F-5. Example of FDS Program and Subroutine used with FDUMP

Subroutine Referenced in Line 14

```
SUBROUTINE CALC(X,Y,ANSW)
X=X*Y+X
ANSW=Y+Y*X
INDX=INDX+1
CALL FDUMP
RETURN;END
```

Figure F-5 (cont). Example of FDS Program and Subroutine used with FDUMP

```
FDUMP CALLED FROM CALC   IN LINE NUMBER     5

SUBPROGRAM CALC
CALLED FROM ...... IN LINE NUMBER     14

INDX                 1
X          4.2900000E 00
Y          2.3000000E 00
ANSW       1.2167000E 01

SUBPROGRAM ......

I                    5
J                    5
KINDX                0
A          4.2900000E 00
B          2.3000000E 00
RESU       1.2167000E 01
C          0.
R          0.

IARR      (*,     1)
    1:               2             3             4             5             6

IARR      (*,     2)
    1:               3             4             5             6             7

IARR      (*,     3)
    1:               4             5             6             7             8

IARR      (*,     4)
    1:               5             6             7             8             9

IARR      (*,     5)
    1:               6             7             8             9            10

ARR       (*,     1)
    1:    1.0000000E 00  2.0000000E 00  3.0000000E 00

ARR       (*,     2)
    1:    2.0000000E 00  4.0000000E 00  6.0000000E 00

ARR       (*,     3)
    1:    3.0000000E 00  6.0000000E 00  9.0000000E 00

FDUMP COMPLETE
```

Figure F-6. Example of FDUMP Output

SUBPROGRAM TIMING MEASUREMENT SYSTEM

The FORTRAN debugging system provides an option that allows the performance of subprograms to be measured in terms of the amount of processor time required to execute those subprograms. This option is called the subprogram timing measurement system. The measurements are given only for those subprograms compiled with the FDS option.

In the batch mode, the timing measurement system is invoked either by including a CALL FTIMER statement in the main program or by including the name FTIMER in the variable field on a $ USE card.

In the time sharing mode, the timing measurement system is invoked by including a CALL FTIMER statement in the main program. The CALL FTIMER statement cannot be inserted from the FDEBUG module.

The timing measurement system determines the following information for each executed subprogram:

1. The number of times the subprogram was called.
2. Global timing, including the processor time used by all called subsidiary subprograms:
   a. Total processor time
   b. Percentage of processor time used
   c. Average processor time per call
3. Local timing, excluding the processor time used by timed subsidiary subprograms:
   a. Total processor time
   b. Percentage of processor time used
   c. Average processor time per call

All times are reported in milliseconds.

Timing Measurement System Examples

Figure F-7 contains an example of the listing that is printed when the subprogram timing measurement system is invoked. Figure F-8 contains an example of the execution of a time sharing program using a CALL FTIMER statement.

NOTE: When the total amount of global time is the same as the total amount of local time, the subprogram has no subsidiaries.

```
                 NO. OF    TOT. MS.    GLOBAL %    AVG. MS.   TOT. MS.   LOCAL %    AVG. MS.
                  CALLS     GLOBAL      OF RUN     PER CALL    LOCAL      OF RUN    PER CALL


.....                1      885.59      100.00      885.59      17.89      2.02       17.89
TESTS                1      867.70       97.98      867.70      54.61      6.17       54.61
REDUN                1      150.20       16.96      150.20     137.38     15.51      137.38
SBSCR4               1      135.97       15.35      135.97       0.19      0.02        0.19
SUB4                 1      135.78       15.33      135.78     129.05     14.57      129.05
SBSCR1               1       93.38       10.54       93.38       0.16      0.02        0.16
SBSCR3               1       93.27       10.53       93.27       0.17      0.02        0.17
SUB1                 1       93.22       10.53       93.22      88.94     10.04       88.94
SUB3                 1       93.09       10.51       93.09      88.95     10.04       88.95
SBSCRI               1       62.11        7.01       62.11       0.13      0.01        0.13
SUB                  1       61.98        7.00       61.98      57.17      6.46       57.17
SBSCR2               1       57.06        6.44       57.06       0.17      0.02        0.17
SUB2                 1       56.89        6.42       56.89      52.64      5.94       52.64
COMP               290       56.86        6.42        0.20      56.86      6.42        0.20
SUBZZA               1       49.84        5.63       49.84       0.25      0.03        0.25
SUBZZZ               1       49.59        5.60       49.59      41.48      4.68       41.48
SPEC                 1       43.00        4.86       43.00      28.47      3.21       28.47
LEXICA               1       41.48        4.68       41.48      38.48      4.35       38.48
CONST                1       27.45        3.10       27.45      24.36      2.75       24.36
DOIF                 1       25.00        2.82       25.00      22.88      2.58       22.88
ONESB                1       15.28        1.73       15.28      13.98      1.58       13.98
COMMON               1       12.58        1.42       12.58      11.83      1.34       11.83
CON                  1        8.98        1.01        8.98       8.42      0.95        8.42
COMPLX               1        5.13        0.58        5.13       4.56      0.52        4.56
ASFL                 1        4.75        0.54        4.75       4.00      0.45        4.00
CLEARA              26        2.30        0.26        0.09       2.30      0.26        0.09
EOS                  1        0.19        0.02        0.19       0.19      0.02        0.19
RDDN                 2        0.05        0.01        0.02       0.05      0.01        0.02
IDOIF                2        0.05        0.01        0.02       0.05      0.01        0.02
TOTAL ELAPSED TIME                     2361.56
TOTAL MEASURED TIME                     885.59
TIMER OVERHEAD                         1475.97
```

Figure F-7. Timing Measurement System Parameters

```
0010 CALL FTIMER
0020 DO 100 I=1,5
0030 CALL SUBA1
0040 CALL SUBA2
0050 PRINT,"BACK TO MAIN"
0060 100 CONTINUE
0070 STOP;END
0080 SUBROUTINE SUBA1
0090 PRINT,"WE ARE IN SUBA1"
0100 DO 200 J=1,1000
0110 200 K=K+J
0120 RETURN;END
0130 SUBROUTINE SUBA2
0140 PRINT,"WE ARE IN SUBA2"
0150 CALL SUBB2
0160 RETURN;END
0170 SUBROUTINE SUBB2
0180 PRINT,"WE ARE IN SUBB2"
0190 RETURN;END
```

```
*LINELENGTH 81
*RUN=(FDS)
WE ARE IN SUBA1
WE ARE IN SUBA2
WE ARE IN SUBB2
BACK TO MAIN
WE ARE IN SUBA1
WE ARE IN SUBA2
WE ARE IN SUBB2
BACK TO MAIN
WE ARE IN SUBA1
WE ARE IN SUBA2
WE ARE IN SUBB2
BACK TO MAIN
WE ARE IN SUBA1
WE ARE IN SUBA2
WE ARE IN SUBB2
BACK TO MAIN
WE ARE IN SUBA1
WE ARE IN SUBA2
WE ARE IN SUBB2
BACK TO MAIN

          NO. OF   TOT. MS.   GLOBAL %   AVG. MS.   TOT. MS.   LOCAL %   AVG. MS.
           CALLS    GLOBAL     OF RUN    PER CALL    LOCAL     OF RUN    PER CALL


......         1    153.72     100.00     153.72      20.52     13.35      20.52
SUBA1          5     89.52      58.23      17.90      89.52     58.23      17.90
SUBA2          5     43.69      28.42       8.74      24.56     15.98       4.91
SUBB2          5     19.13      12.44       3.82      19.13     12.44       3.82

TOTAL ELAPSED TIME             230.50
TOTAL MEASURED TIME            153.72
TIMER OVERHEAD                  76.78
```

Figure F-8. Timing Measurement System in Time Sharing

WRAPUP PROCEDURES

The FORTRAN debugging system provides a mechanism called a wrapup list that allows a user to designate one or more subprograms to be called when a program terminates. The user can also add subprograms to the wrapup list to allow post-execution diagnostic activities or additional functions to be performed. For example, complex data structures such as symbol tables may be analyzed and printed in a readable format.

The wrapup list is maintained dynamically by the FDS in a first-in/first-out sequence; the first subprogram that is entered into the list will be called first.

In the batch mode, the wrapup list is inspected whenever a program terminates abnormally or is terminated by the execution of a FORTRAN STOP statement. When a program terminates abnormally, the first entry in the wrapup list is FDUMP and a symbolic dump is automatically produced.

In the time sharing mode, the wrapup list is inspected whenever a program terminates abnormally with an interrupt (break) or is terminated by the execution of a FORTRAN STOP statement. When a program terminates abnormally, the first entry in the wrapup list is FDEBUG and the dynamic debugging module is entered.

Adding Wrapup Subprograms

An external subprogram can be added to the wrapup list by including the following statements in the source program:

```
EXTERNAL subr
CALL ATCALL(subr)
CALL NTCALL(subr)
```

If an external subprogram is added to the wrapup list by including the CALL ATCALL statement, it is called whenever the program terminates abnormally.

If an external subprogram is added to the wrapup list by including the CALL NTCALL statement, it is called whenever the program terminates in a normal manner.

If a CALL NOCALL(subr) statement is included and executed, all occurrences of 'subr' are deleted from the wrapup list.

The FDS option is not required to process the CALL ATCALL, CALL NTCALL, or CALL NOCALL statements, but the subroutine name must be declared EXTERNAL or else an op code fault is generated.

Example

The following statements are used to remove FDUMP from the wrapup list and to insert FDEBUG in its place:

```
EXTERNAL FDUMP,FDEBUG
CALL NOCALL(FDUMP)
CALL ATCALL(FDEBUG)
```

In this example, FDEBUG is called if the program terminates abnormally.

NOTE: In the batch mode, the desired debugging requests must be present on file 44 and must begin with a RETURN request to enable them to be read by FDEBUG when it is called at program termination. A CALL NOCALL (subr) statement cannot be inserted as a debugging request.

Excluding Wrapup Subprograms

The wrapup mechanism provides a method to avoid calling any of the subprograms contained in the wrapup list. The list is not inspected or called when a CALL FTERM statement is executed.

NOTE: The execution of a CALL FTERM statement causes the immediate termination of the program. A CALL FTERM statement cannot be inserted as a debugging request.

OPTIONAL DEBUGGING FEATURES

Special Printing Formats

If the values of variables or arrays are to be printed in a format other than the default format, subroutines similar to the following may be included in a program:

```
SUBROUTINE PR(A,N,FORMAT)
INTEGER A(N),FORMAT(1)
WRITE(6,FORMAT)A
RETURN
END
```

An FDEBUG request such as

```
CALL PR(ARRAY,3,"(1X,3A6)")
```

can then be used to print data under a special format. In this example, the first three elements of ARRAY are printed with the A6 format.

Debugging Linked Overlay Programs

If linked overlay programs are to be debugged, a subroutine supplied by the FDS can be used to assist in this process. This subroutine is called by the LINK/LLINK overlay subroutine immediately after a link is loaded; it consists of the following statements:

```
      SUBROUTINE LODLNK(LINK)
      CHARACTER*6 LINK
    1 RETURN
      END
```

To allow control to pass to FDEBUG after a certain link has been loaded, the following FDEBUG requests may be inserted:

```
      SUBROUTINE LODLNK
      1 IF(LINK .EQ. "linkname")PAUSE
```

where: "linkname" represents the name of a link having six characters or less.

This coding inserts a request that causes FDEBUG to be entered immediately after "linkname" is loaded. Any FDEBUG requests previously inserted into the overlay area are ignored. (The SHOW request can be used to determine if any previous requests are still present in the program.)

Since a CALL LINK statement can cause the currently executing link to be overlayed, thereby eliminating the subroutine nesting list and possibly LODLNK, control is passed directly to the link entry point by LINK without calling LODLNK. In this case, control cannot be passed to FDEBUG, and it is recommended that LLINK be used instead. In addition, when LLINK is used, the program is more easily moved to other environments by supplying a dummy subroutine named LLINK.

Refer to Figure F-2 for an example of FDEBUG requests that are inserted into linked overlay structures.

Debugging Optimized Programs

When optimized programs are to be debugged, the procedure may be complicated by the fact that the values of certain variables are often stored in registers rather than in memory. This condition is particularly applicable to DO loop indices in loops that exit only from the bottom. The value of the DO loop index cannot be printed (it appears to remain constant), and the value cannot be used in other ways.

FDS Programming Techniques

The following information is provided to assist in the most effective use of the FORTRAN debugging system:

1. The FDEBUG requests represent a language of considerable complexity since:

   a. Conditional requests can be used.

   b. The inserted FDEBUG requests can be dynamically modified.

   c. The GOTO request, particularly when used with the IF request, can significantly change the executed logical flow of the subprogram(s) being debugged from the logical flow specified in the source coding.

   FDEBUG output data can be difficult to interpret unless strongly supported by using the SHOW request. It is generally helpful to provide a SHOW request prior to each RETURN request (except, perhaps, at the initial invocation of the FDEBUG module). When debugging a complex loop, it is also helpful to create a display of all inserted requests prior to each pass through the loop.

2. Since the FDEBUG module is always entered prior to program execution in the batch mode when file 44 is present, a program that is being processed in the batch mode should contain a RETURN request as the first instruction on file 44 unless FDEBUG requests are to be interpreted or inserted before the program is executed.

3. When the first CALL FDEBUG (fc) statement in a program is executed, the FDEBUG module processes debugging requests beginning with the first request contained on file 'fc'. If another CALL FDEBUG (fc) statement is encountered during the execution of the program, FDEBUG begins to process requests immediately following the most recently processed RETURN request. A CALL FCLOSE (fc) statement does not force file 'fc' to be rewound.

4. If an attempt is made to call or otherwise invoke the FDEBUG module and FDEBUG is already currently in control, a RECURSIVE CALL error message is printed and the call or invocation is ignored.

5. Files containing FDEBUG requests cannot be line numbered.

6. A GOTO request cannot be used to transfer from the FDEBUG module to a statement label of a user's program because the GOTO request is always inserted at statement label 'n'; it does not affect FDEBUG control logic. Control is always returned to the next instruction following the CALL FDEBUG statement. (It is possible to circumvent the control return mechanism by issuing a DONE, QUIT, or STOP request; however, these requests terminate the program.)

7. More than one debugging request may be inserted at a statement label in the user's program. All requests that have been inserted at a given statement label can be removed by providing one CONTINUE request at that statement label.

8. If FDUMP or FDEBUG is invoked for a subroutine that contains no symbols or statement labels, a 'SYMBOL TABLE NOT AVAILABLE OR OVERWRITTEN' message is printed.

9. The FDEBUG module does not operate in a correct manner when FTIMER has been invoked.

10. The timing measurement system cannot be called from within the FDEBUG module. To obtain timing data for time sharing programs, a CALL FTIMER statement must be present in the source program during the compilation phase. In the batch mode, as an alternative, the name FTIMER may be included in the variable field on a $ USE card.

11. In the time sharing mode, the FDEBUG module is entered before program execution and the message FDEBUG is displayed on the terminal. A prompting question mark (?) is printed as the first character on the next line, indicating that data is expected; FDEBUG requests can be inserted into the program at this time. The program begins to execute when a RETURN request is entered at the terminal.

12. If a carriage return is the initial response when FDEBUG is entered in the time sharing mode, a traceback is printed. A carriage return following a new identifier request also produces a traceback.

13. When the wrapup list is inspected, a traceback includes the FDS WRAPUP routine.

14. If the !text request is issued when operating in the time sharing mode, the FDEBUG module may lose control. For example, FDEBUG loses control if the time sharing command !RUN=PROG is entered at the terminal, since the program named PROG would then be executed.

15. If FDUMP is called as the result of a "SYSOUT LIMITS EXCEEDED" error message, it is not possible for FDUMP to produce a symbolic dump.

APPENDIX G

FORTRAN TRANSLITERATION TABLE

IBM (IBMF) Character Set to Level 66 (BCD) Character Set

IBMF Character	Card Image Punch	BCD Octal Value	Series 6000 BCD Character
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	10	8
9	9	11	9
none	8-2	12	[
=	8-3	13	#
'	8-4	14	@
none	8-5	15	:
none	8-6	16	>
none	8-7	17	?
blank	blank	20	blank
A	12-1	21	A
B	12-2	22	B
C	12-3	23	C
D	12-4	24	D
E	12-5	25	E
F	12-6	26	F
G	12-7	27	G
H	12-8	30	H
I	12-9	31	I
+	12	32	&
.	12-8-3	33	.
)	12-8-4	34	]
none	12-8-5	35	(
none	12-8-6	36	<
none	12-8-7	37	\
none	11-0	40	↑
J	11-1	41	J
K	11-2	42	K
L	11-3	43	L
M	11-4	44	M
N	11-5	45	N
O	11-6	46	O
P	11-7	47	P
Q	11-8	50	Q
R	11-9	51	R
-	11	52	-
$	11-8-3	53	$
*	11-8-4	54	*
none	11-8-5	55	)

IBMF Character	Card Image Punch	BCD Octal Value	Series 6000 BCD Character
none	11-8-6	56	;
none	11-8-7	57	'
none	12-0	60	+
/	0-1	61	/
S	0-2	62	S
T	0-3	63	T
U	0-4	64	U
V	0-5	65	V
W	0-6	66	W
X	0-7	67	X
Y	0-8	70	Y
Z	0-9	71	Z
none	0-8-2	72	←
,	0-8-3	73	,
(	0-8-4	74	%
none	0-8-5	75	=
none	0-8-6	76	"
none	0-8-7	77	!

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 66)/6000 FORTRAN REFERENCE MANUAL ADDENDUM A	ORDER NO.	DG75A, REV. 0
		DATED	JULY 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here ☐ and furnish complete mailing address below.

FROM: NAME ______________________ **DATE** ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 66)/6000 FORTRAN REFERENCE MANUAL	**ORDER NO.**	DG75, REV. 0
		DATED	JULY 1978

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME ____________________ **DATE** ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

FIRST CLASS
PERMIT NO. 395
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell