Honeywell

LEVEL 66
SOFTWARE
FORTRAN

SERIES 60 (LEVEL 66)

SOFTWARE

FORTRAN

SUBJECT

General Description, Capabilities, Rules and Definitions, User Interfaces, Statements, Input/Output, and Subroutines of the FORTRAN Language.

SPECIAL INSTRUCTIONS

For Series 6000 systems, this manual replaces the manual of the same name, Order No. BJ67, dated March 1973. Order No. BJ67 remains an active publication for Series 600 systems and for Series 6000 systems on prior software releases.

SOFTWARE SUPPORTED

Series 60 (Level 66) Software Release 2
Series 6000 Software Release H

Includes Update Pages Issued as Addendum A in December 1975

ORDER NUMBER

DD02, Rev. 0

January 1975

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.

File No.: 1723,1P23

DD02

Honeywell

FORTRAN ADDENDUM B

SERIES 60 (LEVEL 66)/6000

SOFTWARE

SUBJECT:

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

SPECIAL INSTRUCTIONS:

This update, Order Number DD02B, is the second addendum to DD02, Rev. 0, dated January 1975. 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 B.

SOFTWARE SUPPORTED:

Series 60 Level 66 Software Release 3
Series 6000 Software Release I

DATE:

September 1976

ORDER NUMBER:

DD02B, Rev. 0

21009
3678
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert	Remove	Insert
v thru x	v thru x	6-1, 6-2	6-1, 6-2
2-3, 2-4	2-3, 2-4	6-7 thru 6-16	6-7, 6-8
	2-4.1, blank		6-9, blank
2-5, 2-6	2-5, 2-6		6-9.1, 6-10
2-9 thru 2-12	2-9 thru 2-12		6-11 thru 6-16
2-13 thru 2-16	2-13, blank	6-25 thru 6-32	6-25, blank
	2-13.1, 2-14		6-25.1, 6-26
	2-15, 2-16		6-27, blank
2-19 thru 2-22	2-19 thru 2-22		6-27.1, 6-28
2-25, 2-26	2-25, 2-26		6-29 thru 6-32
3-1 thru 3-6	3-1, 3-2	6-35 thru 6-46	6-35 thru 6-42
	3-2.1, blank		6-42.1, blank
	3-3 thru 3-6		6-43 thru 6-46
3-9 thru 3-26	3-9 thru 3-14	6-49, blank	6-49, blank
	3-14.1, blank	B-1, B-2	B-1, B-2
	3-15, 3-16	B-31, B-32	B-31, B-32
	3-16.1, blank	B-33, blank	B-33, blank
	3-17 thru 3-26	C-1, C-2	C-1, C-2
3-33 thru 3-46	3-33 thru 3-46		F-1 thru F-24
3-47, blank	3-47, blank		F-25, blank
4-1 thru 4-8	4-1 thru 4-6		
	4-7, blank		
	4-7.1, 4-8		
4-11, 4-12	4-11, 4-12		
4-19, 4-20	4-19, 4-20		
4-27, 4-28	4-27, blank		
	4-27.1, 4-28		
4-45, 4-46	4-45, 4-46		
4-49, 4-50	4-49, 4-50		
4-53 thru 4-56	4-53 thru 4-56		
4-61 thru 4-64	4-61 thru 4-64		
5-1, 5-2	5-1, 5-2		
5-9, 5-10	5-9, 5-10		
5-13, 5-14	5-13, 5-14		
5-17 thru 5-28	5-17, blank		
	5-17.1, 5-18		
	5-19, blank		
	5-19.1, 5-20		
	5-21 thru 5-28		

File No.: 1723, 1P23

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

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE Series 60 (Level 66)/Series 6000:	ORDER NO.
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
I/O Via MME GEINOS	I/O Programming	DB82
System initialization:		
System Startup	System Startup	DD33
System Operation	System Operating Techniques	DD50
Communications System	GRTS/355 and GRTS/6600 Startup Procedures	DD05
Communications System	NPS Startup	DD51
DSS180 Subsystem Startup	DSS180 Startup	DD34
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) (Series 60 only)	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
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	DD08
COBOL-68 Language	COBOL	DD25
COBOL-68 Usage	COBOL User's Guide	DD26
JOVIAL Language	JOVIAL	DD23
FORTRAN Language	FORTRAN	DD02
Generators:		
Sorting	Sort/Merge Program	DD09
Merging	Sort/Merge Program	DD09

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE Series 60 (Level 66)/Series 6000:	ORDER NO.
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 (Series 60 only)	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
Remote communications:		
DATANET 30/305/355/6600 FNP	Remote Terminal Supervisor (GRTS)	DD40
DATANET 355/6600 FNP	Network Processing Supervisor (NPS)	DD48
DATANET 700 RNP	RNP/FNP Interface	DB92
Transaction processing:		
User's Procedures	Transaction Processing System User's Guide	DD41
Handbooks:		
System-operator communication	System Console Messages	DD13
Pocket guides:		
Control Card Formats	Control Cards and Abort Codes	DD04
FORTRAN	FORTRAN Pocket Guide	DD82

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

ILLUSTRATIONS

TABLES

SECTION I

INTRODUCTION

GENERAL

FORTRAN is a coding language closely resembling the ordinary language of mathematics and providing the facility for expressing any problem requiring numerical computation. In particular, problems involving large sets of equations and containing many variables can be handled easily. FORTRAN is especially suited for solving scientific and engineering problems, and it is also suitable for many business applications.

The FORTRAN language consists of words and symbols arranged into statements. A set of FORTRAN statements, describing each step in the solution of a problem, constitutes a FORTRAN program (a source language program).

The FORTRAN compiler is a processor that translates a FORTRAN program into machine language. This processor is provided as a part of the software system to translate FORTRAN source language programs to machine language programs in the form acceptable for execution with the General Comprehensive Operating Supervisor (GCOS).

The FORTRAN language is augmented by a library of routines that accompany the system. These routines evaluate the standard arithmetical functions, provide all input/output for the program, and furnish the user with other services to aid in the problem solution. Special purpose routines can be written by the user for use as subprograms.

CAPABILITIES

The FORTRAN compiler services both batch and time sharing, using the same compiler modules for both environments. Users have the capability of developing programs 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.

Users enter FORTRAN programs in exactly the same form regardless of the input medium or location. The only difference in the input stream at the 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

RULES AND DEFINITIONS

CHARACTER SET

FORTRAN utilizes two character sets - ASCII and BCD. The character set and byte size of the internal representation of generated object code is controlled by an option on the $ FORTY or $ FORTRAN card or the YFORTRAN or FORTRAN RUN command. The byte size is 6 or 9 bits, depending on the option selected (BCD or ASCII). Appendix A contains the ASCII and BCD character set with the octal and card representation for each character. The character set of the source program is self-determining and requires no options.

The FORTRAN character set is a subset of the full 128 ASCII characters and is used as follows:

1. FORTRAN statements and the verbs or prepositions do not differentiate between upper and lower case alphabetic characters.
2. No distinction is made between the cases in forming variable, function, common, etc. names.
3. Upper and lower case letters are recognized as different only in user character data and literals.
4. Character restrictions may be necessary for certain external routine procedures. For example, symbols in assembly language subroutines may be restricted to upper case.
5. Any character in the ASCII character set is valid as literal data.

A program unit is written using the following characters:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,
V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,
q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and

CHARACTER	NAME OF CHARACTER
b̸	Space
=	Equals
+	Plus
-	Minus
↑ or ∧	Vertical Arrow or Caret
*	Asterisk
&	Ampersand
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Radix Point
$	Currency Symbol
'	Apostrophe or Acute Accent
;	Semicolon
"	Quotation Marks

The order in which the characters are listed does not imply a collating sequence. All are ASCII characters.

The following special characters are used for FORTRAN syntax punctuation:

Space " $ ( ) + - , / ; = . ' & * ↑ ∧

The space character is not meaningful to the compiler except in character literals and can be used freely to enhance readability of programs.

Quotation marks and apostrophes are used as character literal delimiters. The apostrophe also precedes the record number in random file input/output statements.

The currency symbol identifies statement numbers used as arguments. It also serves as a delimiter of input data for NAMELIST read.

Parentheses are used to enclose subexpressions, complex constants, equivalence groups, format specification, argument lists, subscripts, and to specify the ranges of implied DO loops.

Plus sign indicates algebraic addition, printer carriage control, or a unary operator.

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 statement delimiter.

The equality sign indicates the assignment operator in arithmetic 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 symbolic logical, and relational operators and 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 as either punched cards or typed lines on a terminal. They can also be the product of (output from) the execution of some program, or one can be compressed in a compilation activity through 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 be copied to magnetic tape, or mass storage first. The mass storage file need not be permanent; a normal deck setup produces the compiler input file (S*) on a temporary file. The source program file must be recorded in standard system format (see the File and Record Control manual). The FORTRAN compiler accepts magnetic tape or mass storage files, in standard system format, with any of the following media codes:

0 - formatted BCD line images, without slew control for the printer
1 - compressed BCD card images
2 - (uncompressed) BCD card images
3 - formatted BCD line images, with trailing printer slew control information
5 - time sharing ASCII format (pre-Series 6000 Software Release E)
6 - time sharing ASCII standard system format
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 were 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 made up of statements and comments. A statement can be contained on from one to twenty lines. The first is called an initial line and the rest are called continuation lines. A comment is contained on one line, it is not considered as a statement, and merely provides information for documentary purposes. Comment lines can be placed freely in the program file, even between consecutive continuation lines.

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.

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 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 (...... if 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 sequence number errors.

Format Rules for Lines

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

Source files in standard format should be processed using the FORM option. Time sharing source files should normally use NFORM+LNO. These are the default options when jobs originate from batch and time sharing, respectively. If neither the LNO nor the NLNO option is specified with the NFORM option, LNO is the default option.

FORM FORMATTED LINES

Lines in FORM format have the following characteristics:

1. Comment lines are recognized by a C or * in character position 1.
2. Continuation lines are recognized by a nonblank, nonzero character in position 6 or by an & as the first nonblank character.

Lines containing more than 72 characters (e.g., card images) in FORM format have these additional characteristics:

3. Character positions 73-80 can be used for sequence identification information. This field is not considered part of the statement; it is provided for convenience.

4. No more than 80 characters are processed. If more are present, they are ignored.

FORM format files must not contain line numbers; therefore, the LNO option must not be specified for FORM format files. Where such files are specified, NLNO (rather than LNO) is the default option and the user must either specify NLNO or ignore the option entirely.

NFORM FORMATTED LINES - NLNO

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

1. Comment lines are recognized by a C or * in character position 1.

2. A continuation line is indicated by the ampersand character (&) as the first nonblank character of the line.

Card images in this format also have the characteristic:

3. Character positions 73-80 can be used for sequence identification information.

NFORM FORMATTED LINES - LNO

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

1. A line number field begins in character 1. The line number field can contain up to eight characters and can contain leading blanks. The magnitude of this line number is treated modulo 2^{18} (262,144).

2. Line numbers less than eight characters in length must be terminated by a nonnumeric character.

3. If the character following the line number is a #, it is ignored and the next character is considered to be following the line number.

4. Comment lines are recognized by a C or * as the next character following the line number.

5. A continuation line is indicated by the ampersand character (&) as the first nonblank character following the line number.

Card images in this format do not reserve characters 73-80 for sequence identification information. The statement text can extend into these positions.

Format Rules Common to FORM/NFORM

The above rules indicate that the format options are used to control the following functions:

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

2. Separation of comment lines from statement lines.

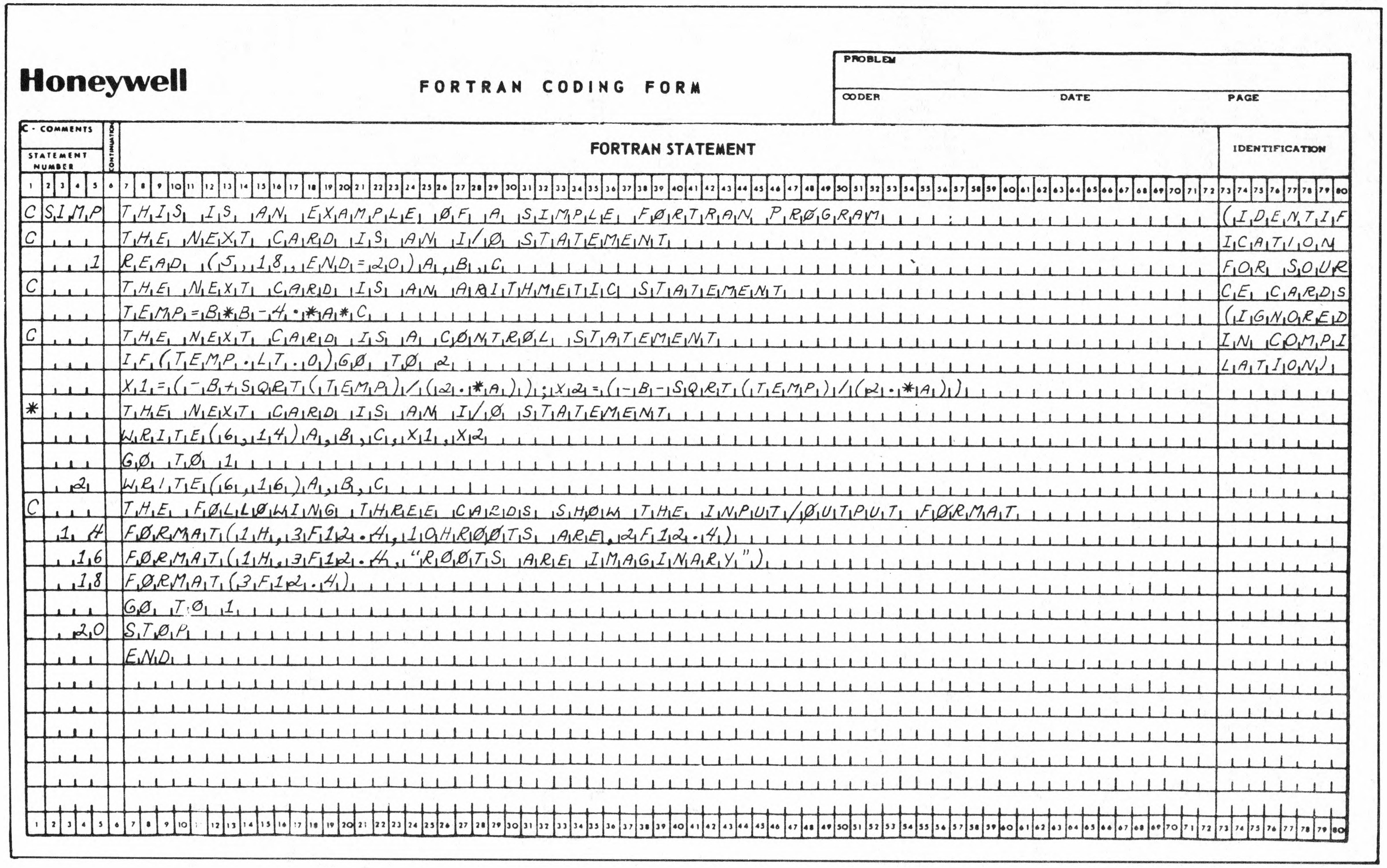
3. Distinction between initial statement lines and continuation lines.

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

Beyond this, the line format is the same. Initial lines can begin with a statement number. The statement number can begin anywhere on the line but must be in the range $1 \leq n \leq 99999$. There can be up to 19 continuation lines and the statement text continues with the first character following the continuation character.

A statement can be terminated by a semicolon on either an initial or continuation line. The information remaining on the line is processed as an initial line. The new statement can begin with a statement number and can be continued. Note that it is not possible to put comments on the same line as the statement line that ends with a semicolon.

Figure 2-1 illustrates the appearance and general properties of a FORTRAN program written on a coding sheet. This example illustrates the FORM format.



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(TEMP)/(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

SYMBOL FORMATION

A symbolic name consists of one to eight alphanumeric characters, the first of which must be alphabetic. Data types can be associated with a symbolic name either implicitly or explicitly. The implicit associations are determined by the first character of the symbol; integer if the name begins with the letters I,J,K,L,M, or N; otherwise real. This default implicit associative rule can be changed by the use of the IMPLICIT statement. This allows implicit association for all data types - integer, real, double precision, complex, logical, or character. An explicit declaration of type for some symbol always overrides its implicit type. 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)).

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

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

DATA TYPES

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^{35} to $2^{35}-1$.

2. A real datum is a processor approximation to the value of a real number. It can assume positive, negative, or zero values, possibly fractional. A real datum requires one 36-bit word of storage in floating point format. The permissible range of values for real type is approximately $\pm 10^{38}$ to $\pm 10^{-38}$, with a precision of eight digits.

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 $\pm 10^{38}$ to $\pm 10^{-38}$, with a precision of 18 digits.

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 of the pair represents the real part and the second, the imaginary part. Each part has, accordingly, the same degree of approximation as for 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.

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., and in DATA statements, the single character "T" can also be used. For the value "false", .FALSE. and .F. can be generally used with "F" being allowable in DATA statements. A logical datum requires one 36-bit word of storage with the value zero representing "false", and nonzero representing "true". Where input/output is involved, the external representations of "true" and "false" are the single letters "T" and "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 by nH. The character set (BCD or ASCII) is declared by an option on the $ FORTY or $ FORTRAN control card or the YFORTRAN or FORTRAN RUN command.

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 containing the reference. If the datum is identified but not necessarily made available, the datum is said to be "named". One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

CONSTANTS

There are three general types of constants - single word, double word, and character. 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

A constant is a value known prior to writing a FORTRAN statement and does not change during program execution.

Integer Constants (Fixed-Point Binary)

An integer constant consists of one to 11 decimal digits with an accuracy of ten digits. The decimal point of the integer must always be omitted; however, it is always assumed to be immediately to the right of the last digit in the string. An integer constant can be as large as $(2^{35})-1$ ($\cong 3.4\times10^{10}$), except when used for the value of a subscript or as an index of a DO or a DO parameter, in which case the maximum value of the integer is $(2^{18})-1$ ($\cong 2.6\times10^{5}$).

Examples:

```
   -7
    152
    843517
```

Octal Constants

An octal constant is written as a string of up to 12 octal digits preceded by the letter O and an optional sign. The sign affects only bit 0 of the resulting literal (complementation does not take place). Octal constants can be used in preset data lists only (e.g., DATA statement).

Examples:

```
   O   777000
   O - 377777777
```

Real Constants (Floating-Point Binary)

A real constant is in floating-point mode and is contained in one computer word (single precision). This constant consists of one of the following:

1. One to nine significant decimal digits written with a decimal point, but not followed by a decimal exponent.

2. One to nine significant decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter E followed by a signed or unsigned one-or two-digit integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value can be explicitly 0, and the field following the E cannot be blank.

Examples:

```
   75.
 1234.
   21.083
   -3.2105
    7.0E2      (means 7.0 x 10^2; 700)
    7E-3       (means 7.0 x 10^-3; .007)
```

A real constant has precision to eight digits. The magnitude must be between the approximate limits of 10^{-38} and 10^{38}, or must be zero.

Double Precision Constants

A double precision constant is in floating-point mode and is contained in two computer words. This constant consists of one of the following:

1. Ten to eighteen significant decimal digits written with a decimal point, but not followed by a decimal exponent. In some cases, ten or eleven significant decimal digits will not generate a double precision constant because the mantissa of the real constant is less than 2^{28}.

2. Up to 18 significant decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter D followed by a signed or unsigned one- or two-digit integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value can be explicitly 0, and the field following the D cannot be blank.

Examples:

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

Double precision constants have precision to 18 digits. The magnitude of a double precision constant must lie between the approximate limits of 10^{-38} and 10^{38}, or must be zero.

Complex Constants

A complex constant consists of an ordered pair of signed or unsigned real constants separated by a comma and enclosed in parentheses.

Examples:

(10.1, 7.03) is equal to 10.1 + 7.03i
(5.41, 0.0) is equal to 5.41 + 0.0i
(7.0E4, 20.76) is equal to 70000. + 20.76i

where i is the square root of -1.

The first real constant represents the real part of the complex number; the second real constant represents the imaginary part of the complex number. The parentheses are required regardless of the context in which the complex constant appears. Each part of the complex constant can be preceded by a plus sign or a minus sign, or it can be unsigned.

Logical Constants

A logical constant can take either of the two forms:

```
.TRUE. (or .T.)
.FALSE. (or .F.)
```

and is represented in the machine as

$$\text{TRUE} \neq 0$$
$$\text{FALSE} = 0$$

Representation can be in either form in DATA statements or externally when performing input/output operations.

Character Constants

Character constants are of two kinds, characterized by their representation in either the ASCII or the BCD character set (see Appendix A). The kind is determined by an option on the $ FORTY or $ FORTRAN card or the YFORTRAN or FORTRAN RUN command. Character constants are formed in one of the following ways:

1. Preceding the character string by nH.

2. Enclosing the string in quotation marks.

3. Enclosing the string in apostrophes.

Character constants can be used as arguments to external subprograms, 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, or in a relational expression.

The maximum length of a character constant is 500 characters in the ASCII mode and 511 characters in the BCD mode.

The interpretation of quoted strings of both types is such that the appearance of the string delimiter in two consecutive character positions within a string is considered as a single occurrence of the delimiter as a member of that string. For example, the representation: "abc""ef" is represented internally as the literal abc"ef. Alternatively, the other delimiter type can be used (e.g. 'abc"ef').

VARIABLES

Variable Type Definition

A variable is any quantity referred to by name rather than by value. A variable can take on many values and can be changed during the execution of the program.

The type of a variable is specified implicitly by its name, or explicitly by 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, (upper or lower case) it is an integer name.

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

2. The IMPLICIT type statement redefines the implicit typing. See the IMPLICIT statement description in Section IV.

3. The explicit type statements assign a type to a variable or function subprogram.

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

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. A scalar variable occupies the same number of storage locations as a constant of the same type.

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.

Parameter Symbols

A parameter symbol is used when it is desired to compile a program several times when the only changes from one compilation to the next are to certain constants. The parameter symbol (described under the PARAMETER statement in Section IV) is used under these circumstances.

Switch Variable

A switch variable is an independent entity derived from a scalar variable and is associated only with an ASSIGN statement. Switch variables must be type INTEGER but can have the same name as an integer variable. Refer to the assigned GO TO statement in Section IV.

Character Variables

Character variables can be implicitly typed via the IMPLICIT statement or explicitly typed using the CHARACTER statement. Character variables are stored internally left-justified and blank-filled. The limit is 500 characters per character variable in the ASCII mode and 511 characters in the BCD mode.

Array

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

Array Element

An array element is an item of data in an array. It is identified by immediately following the array name with a subscript that 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 containing from one to seven dimensions by appending one to seven subscripts to the variable name. Subscript expressions are separated by commas. The number of subscript expressions must correspond with the declared dimensionality except in an EQUIVALENCE statement. Following evaluation of all of the subscript expressions, the array element successor function determines the identified element.

Form of Subscript

A subscript expression can take the form of any legal FORTRAN arithmetic expression. The result of any such expression is truncated to an integer before use.

Examples:

```
     IMAS             8*IQUAN          9+J
     J9               5*L+7            B**2
     K2               H*M-3            6**A-(1-SQRT(3.14))/8
     N+3              7+2*k            LIST (J)
```

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.

Subscripted Variables

A subscripted variable consists of a variable name, followed by parentheses, enclosing one to seven subscripts separated by commas.

Examples:

```
A(I)
K(3)
BETA (8*J+2,K-2,L)
MAX (K,J,K,L,M,N)
```

1. During execution, the subscript is evaluated so that the subscripted variable refers to a specific element of the array.

2. Each variable that appears in subscripted form must have the size of the array specified. This must be done by DIMENSION, COMMON, or type statements that contain the dimension information. The specification of dimensionality must precede the first reference to the array.

3. The first subscript refers to rows of the array, the second subscript to columns, and the third subscript to planes consisting of rows and columns.

Array Element Successor Function

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

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

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 (since they have a maximum of 511 characters) and up to 126 in ASCII (since they have a maximum of 500 characters). The formula for reducing size in characters to 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 and complex quantities:

```
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:

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

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

$$S = 11$$

Looking at the array in storage in ascending 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, and i is the declarator subscript. Declarator subscript (i) is composed of from one through seven elements each of which can be an integer constant, a parameter symbol or an integer variable. Each element is separated by a comma (if more than one).

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 whose sizes are not determined until the subprogram is called. The following rules apply to the use of adjustable dimensions:

1. Variables can be used as dimensions of an array only in the array declarator of a FUNCTION or SUBROUTINE subprogram. For any such array, 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 some calling program.

4. The calling program passes the specific dimensions to the subprogram. These specific dimensions are those that appear in the DIMENSION, COMMON, or type statement of the calling program. Variable dimension size can be passed through more than one level of subprogram. The specific dimensions passed to the subprogram as actual arguments cannot exceed the true dimensions of the indicated array.

5. Variables used as dimensions must be integers. If the variables are not implicitly typed by their initial letters, a type statement must precede the dimension statement in which they are used as adjustable dimensions.

6. If 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 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
                                          .
                                          .
                                          .
                                          .
```

EXPRESSIONS

Arithmetic

An arithmetic expression consists of certain legal sequences of constants, subscripted and nonsubscripted variables, and arithmetic function references separated by arithmetic operation symbols, commas, and parentheses.

The following are arithmetic operation symbols:

+	addition
-	subtraction
*	multiplication
/	division
** ↑ ∧	exponentiation

The rules for constructing arithmetic expressions are:

1. Figures 2-2 and 2-3 indicate which constants, variables, and functions can be combined by the arithmetic operators to form arithmetic expressions. The intersection of a row and column gives the type of the result of such an expression. Figure 2-2 gives the valid combinations with respect to the arithmetic operators +,-,*, and /. Figure 2-3 gives the valid combinations with respect to the arithmetic operators **, ↑, or ∧.

	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

Figure 2-2. Arithmetic Expressions +, -, *, and /

		POWER				
		I	R	D	C	T
BASE	I	I	R	D	N	N
	R	R	R	D	N	N
	D	D	D	D	N	N
	C	C	C	C	C	N
	T	N	N	N	N	N

Figure 2-3. Arithmetic Expressions - Exponent (**,↑, or ∧)

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.

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

4. The expression A**B**C is evaluated as A**(B**C).

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

6. In the hierarchy of operations, parentheses can be used in 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:

a.	Function Reference	
b.	**, ↑, or ∧	Exponentiation
c.	* and /	Multiplication and Division
d.	+ 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.

7. Operations on the same level (e.g. A*B/C) are evaluated left to right. Parentheses can be used to reorder this sequence if necessary.

The FORTRAN expression

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

represents the mathematical expression

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

Relational

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

The six relational operator symbols are:

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

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, and the value .FALSE. otherwise.

Logical

A logical expression consists of certain sequences of logical constants, logical variables, references to logical functions, and relational expressions separated by logical operation symbols. A logical expression always results in a true or false evaluation.

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

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.

Logical expression evaluation proceeds to determine the true/false state of the simpler subexpressions first, and stops (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. Since this may be of significance to some applications, the following example is given:

```
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. The second alternative is clearly simpler to determine than the first. Further, 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. For 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. Figure 2-4 indicates which constants, variables, functions, and arithmetic expressions can be combined by the relational operators to form a relational expression. In Figure 2-4, 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.

.GT., .GE., .LT., .LE., .EQ., .NE.	I	R	D	C	L	Ctr	T	Legend
I	Y	Y	Y	*	N	Y	Y	I = Integer R = Real D = Double Precision C = Complex L = Logical Ctr = Character T = Typeless
R	Y	Y	Y	*	N	N	N	
D	Y	Y	Y	*	N	N	N	
C	*	N	N	*	N	N	N	
L	N	N	N	N	N	N	N	
Ctr	Y	N	N	N	N	Y	N	* = .EQ.,.NE. only
T	Y	N	N	N	N	N	Y	Y = Valid N = Invalid

Figure 2-4. Use of Relational Operators

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

   a. 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 $+5 > -5$.

   b. 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 can consist of 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.

Operator Precedence

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 (from innermost operations to outermost operations):

1. Function Reference
2. **,↑, or ∧ Exponentiation
3. + and - Unary Addition and Subtraction
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; this procedure continues through the outermost set of parentheses until the entire expression has been evaluated.

Typeless

The following functions are considered as typeless:

FLD
AND
OR
XOR
BOOL
COMPL

A typeless result is regarded as a special form of integer. Typeless entities can be combined with integer or other typeless entities. With the arithmetic operators the result is typeless; with relational operators the result is logical; the logical operations cannot be used on typeless entities. Whenever the right of equals 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.

Evaluation of Expressions

A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it.

When two elements are combined by an operator, the order of evaluation of the elements is undefined because of possible reordering during optimization. If mathematical use of operators is associative, commutative, or both, full use of these facts can be made to revise orders of combinations, provided only that integrity of parenthesized expressions is not violated. 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 functions appearing in an expression cannot validly 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 primary to be raised to a real or double precision exponent. No factor can be evaluated that requires raising a zero valued primary to a zero valued exponent. No element can be evaluated whose value is not mathematically defined.

The mode of evaluation of arithmetic expressions is determined by the following order of type dominance:

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

When two primaries are combined by any of the arithmetic operators except the exponentiation operator, their 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.

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)
```

FORTRAN STATEMENTS

Types of FORTRAN Statements

The basic unit of FORTRAN is the statement. Statements are classified according to the following uses:

1. Arithmetic statements specifying numerical, 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.

Arithmetic Statements

assignment statements
arithmetic statement functions

Control Statements

ASSIGN
CONTINUE
DO
GO TO
IF
PAUSE
STOP

Input/Output Statements

BACKSPACE
DECODE
ENCODE
END FILE
FORMAT
PRINT
PUNCH
READ
REWIND
WRITE

Subprogram Statements

BLOCK DATA
CALL
ENTRY
FUNCTION
RETURN
SUBROUTINE

Specification Statements

ABNORMAL
COMMON
DATA
DIMENSION
EQUIVALENCE
EXTERNAL
IMPLICIT
NAMELIST
type
- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- LOGICAL
- CHARACTER

PARAMETER

Compiler Control Statement

END

Index of Statements

Table 2-1 contains an alphabetical listing of FORTRAN statements giving an example with the page number for the statement in Section IV.

Table 2-1. Alphabetical Listing of FORTRAN Statements

Statement	Example	Page
arithmetic statement function	F(X,Y)=(X+1)*Y(I)	4-2
assignment statement	A=4*B-SINE(C**2)	4-2
ABNORMAL	ABNORMAL SINE	4-7
ASSIGN	ASSIGN 2 TO K	4-7.1
BACKSPACE	BACKSPACE 5	4-8
BLOCK DATA	BLOCK DATA	4-9
CALL	CALL MATMPY (X,5,10,4,Z)	4-10
CHARACTER	CHARACTER ARRAY*14(10,10)	4-12
COMMON	COMMON X,Y,Z	4-13
COMPLEX	COMPLEX T,N1,D1	4-15
CONTINUE	CONTINUE	4-16
DATA	DATA A,B,C /3.5,2.9,6.0/	4-17
DECODE	DECODE (CHARS,95 01) A,I(3),X	4-20
DIMENSION	DIMENSION A(50)	4-21
DO	DO 35 K=10,20,2	4-22
DOUBLE PRECISION	DOUBLE PRECISION DENOM,PREF	4-26
ENCODE	ENCODE (CHARS(6),9001)A,I(3),X	4-27
END	END	4-28
ENDFILE	ENDFILE 5	4-29
ENTRY	ENTRY INVRT (B,C,D)	4-30
EQUIVALENCE	EQUIVALENCE (A,B,C)	4-31
EXTERNAL	EXTERNAL SIN,COS,RESULT	4-34
FORMAT	10 FORMAT (E17.2,F20.0)	4-35
FUNCTION	FUNCTION CALC (B,C,D)	4-37
GO TO, assigned	GO TO S4, (3,4,7)	4-40
GO TO, computed	GO TO (3,4,7),K	4-41
GO TO, unconditional	GO TO 20	4-40
IF,arithmetic	IF (A(J,K)-B)10,4,30	4-42
IF,logical	IF (A.GT.B) GO TO 3	4-43
IMPLICIT	IMPLICIT INTEGER (A-F,X,Y)	4-45

Table 2-1 (cont). Alphabetical Listing of FORTRAN Statements

Statement	Example	Page
INTEGER	INTEGER I,ABC	4-46
LOGICAL	LOGICAL Al,K	4-47
NAMELIST	NAMELIST/LIST/ R,S,T,U,V	4-48
PARAMETER	PARAMETER I=5/2,J=I*3	4-49
PAUSE	PAUSE 1234	4-50
PRINT, list directed	PRINT,A	4-52
PRINT,formatted	PRINT 20,A	4-52
PRINT,namelist	PRINT LIST	4-52
PUNCH,list directed	PUNCH,A	4-53
PUNCH,formatted	PUNCH 20,A	4-53
PUNCH,namelist	PUNCH LIST	4-53
READ,list directed	READ,A	4-54
READ,formatted	READ 20,A	4-54
READ,namelist	READ LIST	4-54
READ,formatted file	READ(5,20,END=90,ERR=95) A	4-54
READ,unformatted file	READ(5,END=90,ERR=95) A	4-55
READ,random binary file	READ(8'I)A	4-55
READ,namelist file	READ(5,LIST)	4-55
REAL	REAL J	4-56
RETURN	RETURN	4-57
REWIND	REWIND 5	4-58
STOP	STOP 100	4-59
SUBROUTINE	SUBROUTINE ALPHA (B,C,D)	4-60
type	INTEGER A,B,C,D	4-62
WRITE,formatted file	WRITE(6,30,ERR=S4)A	4-64
WRITE,unformatted file	WRITE(6,ERR=S4)A	4-64
WRITE,namelist file	WRITE(6,LIST)	4-64
WRITE,random binary	WRITE(8'I)A	4-64

SECTION III

USER INTERFACES

Users create programs by entering FORTRAN statements into remote and local peripheral or terminal devices connected to a computer operating under GCOS.

The interface between the user and the FORTRAN system consists of the transmission to the user's I/O device of compilation error messages and run-time diagnostics. The messages transmitted are sufficient to locate for the user the line on which the error occurred, and the form of the message is such that the error is explicitly defined.

Three modes of operation are available to the user: local batch, remote batch, and time sharing. The only user differences among the three modes are the I/O device assignments for the system output and input files, the presence of necessary user-GCOS communication via control cards or command language, and the assumed compiler options for the compilation process.

BATCH MODE

In the local batch mode, the system I/O devices 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. The execution of user programs submitted via the local batch mode is carried out directly under GCOS and the user's 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. The only difference is the assignment of the system I/O device to the remote computer as remote files rather than to the local card reader and local printer/punch. Refer to the Remote Terminal Supervisor (GRTS) and the Network Processing Supervisor (NPS) manuals for additional communication information.

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 the system options. Options available with time sharing FORTRAN are listed under the time sharing paragraph in this section. The following options 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.

NOMAP - No storage map is prepared.

XREF - A cross-reference report is prepared as output. A TO-FROM transfer table is generated.

NXREF - No cross-reference report is prepared.

DEBUG - A run time debug symbol table (.SYMT.) is included in the object program.

NOTE: This debug symbol table is used for debugging in the batch mode only. Refer to the General Loader manual for use of the debug feature and the debug symbol table.

NDEBUG - No debug symbol table is prepared.

BCD - The execution time character set is standard BCD (see Appendix A).

ASCII - The execution time character set is ASCII (see Appendix A).

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 - Do not restart this job following system interruption.

JREST - Enable job restart following system interruption.

NREST - Do not restart this job with current activity following system interruption.

REST - Enable activity restart 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 - Do not print any compilation warning messages.

FDS - Enables the FORTRAN Debugging System (FDS). See Appendix F.

NOTE: 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.

Sample Batch Deck Setup

The following are 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 Cards
$      FFILE Cards
$      ENDJOB
```

TIME SHARING SYSTEM OPERATION

From a user point of view there are two time sharing versions of the FORTRAN compiler. Each version is invoked by a different call. These versions and the language call for each are as follows:

Compiler Version	Language Call (at system level)
Batch based time sharing compiler	YFORTRAN
Time sharing based compiler	FORTRAN

In this document, the batch based time sharing interface is referred to as the YFORTRAN Time Sharing System and the time sharing based system is referred to as the FORTRAN Time Sharing System. 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) and differs from the batch based time sharing compiler in the following areas.

1. Compiles under the GCOS Time Sharing System.

2. Eliminates the need for configuring batch memory; YFORTRAN compiles through DRL TASK. (Refer to the TSS System Programmer's Reference Manual.)

3. Retains essentially the current RUN syntax with modifications as noted in this section.

4. Interfaces with 4K time sharing loader module (YLDA).

5. Significant overhead reduction in FORTRAN time sharing system.

6. Blank common allocation is common to both time sharing and batch.

7. "CORE=" clause is not required for compiles.

8. Compilers are identical except for the executive phase (YEXC vs YTEX).

The only user differences, other than those noted above, are the I/O device assignments for the system output and input files, the presence of necessary user GCOS communication via control cards or command language, and the assumed compiler options for the compilation process.

Time Sharing System Command Language

The standard means of communication with the GCOS Time Sharing System (TSS) is by way of a terminal. The user may choose either the keyboard/printer or paper-tape terminal unit for input/output, or combine both. In either 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 user "controls" the time sharing system primarily by means of a command language, a language 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, for the most part, the same for users of 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, in form and meaning, common to all component systems.

The commands relate to the generation, modification, and disposition of program and data files, and program compilation/execution requests. The complete time sharing command language is described in TSS General Information manual.

Once communication with the system has been established, any question or request from the system must be answered within ten minutes, except for the initial requests for user identification (user-ID) and sign-on password, that must be given within one minute. If these time limits are exceeded, the user's terminal is disconnected.

Time Sharing Commands of the YFORTRAN and FORTRAN Time Sharing Systems

The valid time sharing system commands are listed in Table 3-1. These commands are fully described in the TSS General Information manual. The RUN command for the YFORTRAN and FORTRAN Time Sharing Systems is more fully described in this manual.

Table 3-1. YFORTRAN and FORTRAN Time Sharing System Commands

Command	Applicable At Build Mode
ABC	Yes
ACCESS	Yes
AFT	Yes
ASCASC	Yes
ASCBCD	Yes
AUTOMATIC	Yes
BCDASC	Yes
BPRINT	Yes
BPUNCH	Yes
BYE	Yes
CATALOG	Yes
DELETE	Yes
DONE[a]	Yes
EDIT	Yes
ERASE	Yes
FDUMP	Yes
GET	Yes
HELP	Yes
HOLD	Yes
JABT	Yes
JOUT	Yes
JSTS	Yes
LENGTH	Yes
LIB[a]	Yes
LIST	Yes
NEW[a]	Yes
NEWUSER	Yes
NO PARITY	Yes
OLD[a]	Yes
PARITY	Yes
PERM	Yes
PRINT	Yes
PURGE	Yes
RECOVER	Yes
#RECOVER	No
RELEASE	Yes
REMOVE	Yes
RESAVE	Yes
RESEQUENCE[a]	Yes
ROLLBACK	Yes
#ROLLBACK	No
RUN[a]	Yes
SAVE	Yes
SCAN	Yes
SEND	Yes
STATUS	Yes
SYSTEM[a]	Yes
TAPE	Yes

[a]Not an applicable response to SYSTEM?

Log-On Procedure

The user, to initiate communication with the Time Sharing System, performs the following steps:

1. Activates the terminal unit.
2. Obtains a dial tone.
3. Dials one of the numbers of the time sharing center.

The user then receives either a high-pitched tone indicating that the terminal has been connected to the computer or a busy signal. The busy signal indicates that no free line is presently available.

Once the user's terminal has been connected to the computer, the time sharing system begins the log-on procedure after the user depresses the return ('CR') key by transmitting a message similar to the following:

```
HIS SERIES 60 ON(date)AT(time)CHANNEL(nnnn)
```

where time is given in hours and thousandths of hours (hh.hhh), and nnnn is the user's line number.

Following the message, the system asks for the user's identification:

```
USER ID --
```

The user responds, on the same line, with the user identification (user-ID) that has been assigned by the time sharing installation management. This user-ID uniquely identifies a particular user already known to the system, for the purposes of locating user programs and files and accounting for usage of the time sharing resources allocated. An example request and response might be:

USER ID -- <u>J.P.JONES</u>

NOTE: A carriage return must be given following any complete response, command, or line of information typed by the user.

(The user's response is underlined here for illustration.) After the user responds with user-ID, the system asks for the sign-on password that was assigned along with user-ID, as follows:

```
PASSWORD
ABCDEFGHIJKL
```

The user types the password directly on the "strikeover" mask provided below the request PASSWORD. The password is used by the system as a check on the legitimacy of the named user. The "strikeover" mask insures that the password, when typed, cannot be read by another person. (In the event that either the user-ID or password is twice given 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 per cent of current resource allocation, the user receives a warning message. If accumulated charges exceeds 110 per cent of current resources, the user receives the message:

```
RESOURCES EXHAUSTED - CANNOT ACCEPT YOU
```

and the terminal is immediately disconnected. (The user may also receive the following information message if the situation warrants it:

```
n BLOCKS FILE SPACE AVAILABLE
```

This condition does not affect the log-on procedure.)

Assuming that the user has responded with a legitimate user-ID and password and has not over extended resources, the time sharing system then asks the user to select the processing system that the user wants to work with; this is called the system-selection request. In this case, the user would respond with YFORTRAN or FORTRAN (can be abbreviated to YFORT or FORT):

```
SYSTEM ? YFORTRAN or FORTRAN
```

The user is then asked whether a new program (NEW) is to be entered or if the user wants to retrieve and work with a previously entered and saved program (OLD); the request message is:

```
OLD OR NEW -
```

If the user wishes to start a new program (i.e., build a new source file), the response is simply:

```
NEW
```

If, on the other hand, the user wants to recall an old source-program file, the response is:

```
OLD filename
```

where filename is the name of the file on which the old program was saved during a previous session at the terminal (see the SAVE command).

Following either response, the system types the message READY, returns the carriage, and prints an asterisk in the first character position of the next line:

```
READY
*
```

An example of a complete log-on procedure, up to the point where the YFORTRAN or FORTRAN system is ready to accept program input or control commands, might be as follows:

```
HIS SERIES 60 ON 07/26/74 AT 17.768 CHANNEL 0012

USER ID - J.P.JONES
PASSWORD
XXXXXXXXXXXX     - (user's password is typed over the mask)
SYSTEM-YFORTRAN or FORTRAN
OLD OR NEW - NEW - (NEW is shown arbitrarily for illustration)
READY
*                - (the user begins entering input on this line)
```

Entering Program-Statement Input

After the message:

```
READY
*
```

the system is 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. Normally the current file is the file that contains the program the user wants to compile and run at this session. If the user is building a new file (NEW response to OLD OR NEW--), the current file is initially empty. If an old file (OLD filename) is recalled the content of the named old file is initially on current file, and any input typed by the user -- excepting control commands -- will be either added to, merged into, or will replace lines in the current file, depending upon the relative line numbering of the lines in the file and the new input. (This process is explained under the heading "Correcting or Modifying a Program," below.)

Following each line of noncommand-language input and the terminating carriage response, the system supplies another initial asterisk, indicating that it is ready to accept more input.

Format of Program-Statement Input

A line of FORTRAN input -- as distinct from a control command -- can contain one of the following:

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 (2), in that order.
6. A combination of (1) and (2).

A line of input may begin with a line-sequence number of from one to eight numeric characters. The line-sequence number facilitates correction and modification of the source program (described below); hereinafter, the line-sequence number will be referred to simply as the "line number". (Note that a line number is distinct from a statement number; a statement number is a part of the FORTRAN language statement itself.)

The line number is always terminated with (i.e., immediately followed by) a single control character that can be a blank, an ampersand, a number sign, an asterisk, or the letter C. The control character merely serves to indicate what type information is to follow (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, of course, 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 then as follows:

```
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), and

c is a single-character 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.

SIGNIFICANCE OF THE CONTROL CHARACTER

The control character identifies the type of information that follows it.

ƀ (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 the next 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 a 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 information following is assumed to be a comment. The comment itself is terminated by a carriage return.

# (number sign) - If a numeric character is desired in column 1 of the card image and line numbers exist in the source file, a number sign (#) character immediately following the line number causes the character following it to be placed into column 1.

A semicolon within a noncomment line indicates both the end of the preceding statement and that any significant information (nonblank, noncarriage return) following it begins a new statement. The new statement can include a FORTRAN statement number, mm...m.

The format of a statement, as entered following a blank control character, is:

...nnɓ <u>ɓ...ɓ mm...m FORTRAN-language text</u>

(The statement-format portion is underlined.)

where: <u>ɓ...ɓ</u> are optional blanks, and

<u>mm...m</u> is an optional numeric statement number that must be equal to or less than 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, except that blanks are illegal within the line number and that the nonnumeric character (including ɓ) immediately following the line number is interpreted as a control character. Thus, spacing can be used quite freely within a line of input in the interest of legibility. (Blanks within character constants and nH fields -- i.e., alphanumeric information -- are meaningful however, and are retained in the object program coding.)

Note that the line/statement format is, except for the relative position of the control character, completely free-form, or position independent.

To this point, the discussion of line format has been oriented to the NFORM form described earlier in this discussion. This is generally the most convenient form to use in time sharing. It is not mandatory, however. The source file can be built using the Text Editor and be without line numbers. The NLNO option permits this. Or, the source can be in "fixed" format (without line numbers). The FORM option can be used to indicate this. The full spectrum of line formats and source file recording modes is available to the time sharing user.

Correcting or Modifying a Program

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 program (i.e., the contents of the user's current source file) by the replacement or deletion of lines contained therein, or the insertion of new lines.

The correction of a typing error that is detected by the user 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 or cancel the incomplete line and start over. Use of the delete control character deletes from the line the character preceding the deletion character; use of n consecutive deletion characters deletes the n preceding characters (including blanks). Delete control characters differ between makes of terminals.

Correction or modification of the current source file is done on the basis of line numbers and proceeds according to the following rules:

1. Replacement. A numbered line replaces any identically numbered line that was previously typed or contained on the current file (i.e., the last entered line numbered nnn will be the only line numbered nnn in the file).

2. Deletion. A "line" consisting of only a line number (i.e., nnn) causes the deletion of any identically numbered line that was previously typed or contained on the current 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.

At any point in the process of entering program statement input, the LIST command can be given, which results in an up-to-date copy of the current file being printed. In this way, the results of any previous corrections or modifications can be verified visually. Following the response (or command) OLD filename, the LIST command can be used initially to inspect the contents of the current source file (i.e., the "old" program).

Input Error Recovery

The decimal input/output routine permits the time sharing user (BCD or ASCII) to correct a string of characters entered from a terminal when a character is illegal for the current format conversion. For example: a decimal point is illegal in an "I" field. The current input line is printed on the terminal with a pointer to the illegal character. The user can now enter a correction to replace the corresponding characters previously entered. The input/output routine resumes with the new string. If the user responds with a carriage return, an error message is printed.

Time Sharing System Definitions and File Specification

An explanation of time sharing terms and file specification is given in Appendix D.

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 giving date, time, and SNUMB. Any of the seven following options can be specified with the RUN (or RUNH) command:

```
-nnn  fs  =  fh   ;fc   (opt)   ulib  #fe
```

If any of the options fh, ;fc, or (opt) of the RUN or RUNH command are used, the equal sign (=) must be included in the string. The RUN command options are described below:

-nnn   nnn is the maximum processor time the program is to be allowed for execution (in seconds).

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. Where a BCD or COMDK source file is supplied, fs can also include a descriptor for an alter file in BCD format. 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. Alternatively, the list fs can consist of a single file descriptor that points to a previously generated system loadable (H*) file. Concatenation of source files is provided by using a separate semicolon between each file descriptor.

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 (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 the named file does not exist, a permanent linked file of three blocks is created and added to the user's catalog. This file expands as necessary to hold the object decks. In this case, the field fs plus the libraries need not 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 which, when specified, must be separated by commas. Some of these options affect the compilation process and some affect the loading process. The following options are available for time sharing; the default options are underlined.

DEBUG - Generate run time debug symbol table.

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> - No run time debug symbol table is generated.

BCD - Object character set is BCD. If applicable, this option must be specified whenever 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).

<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 - Optimize the object module.

<u>NOPTZ</u> - Do not optimize the object module.

NWARN	- Do not print any compilation warning messages.
CORE=nn	- The compilation activity memory requirement is set to nnK+6K or 26K, whichever is larger. If not specified, nn is set to 20.
FDS	- Enables the FORTRAN Debugging System (FDS). See Appendix F.

The remaining options concern the loading process. The underlined option is the default (i.e., assumed) case:

<u>GO</u>	- The program is executed at the completion of compilation.
<u>NOGO</u>	- The program is not executed at the completion of the compilation. If specified, the object program is saved. If no object (H*) save file is specified, only the compilation is performed (General Loader is not called).
ULIB	- File descriptors 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.
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 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. Can be used both 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 (see Time Sharing Command Language and File Usage in the <u>TSS General Information</u> manual). The file description can be in any of the following formats:

1. <u>filename</u> specifying a filename in the form nn where 01 $\leq$ nn $\leq$ 43 and nn represents a logical file code referenced by the I/O statements in the program.

2. <u>filedescr</u> specifying a full description.

   a. <u>filename</u>

   b. <u>filename$password</u>

   c. <u>userid/catalog$password...</u>

The FORTRAN Time Sharing System RUN Command

The FORTRAN 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 giving date and time. Any of the seven following options can be specified with the RUN (or RUNH) command:

```
-nnn  fs  =  fh  ;fc  (opt)  ulib  #fe
```

If any of the options fh, ;fc, or (opt) of the RUN or RUNH command are used, the equal sign (=) must be included in the string. The RUN command options are described below:

-nnn — is the maximum processor time the compiled object program is to be allowed for execution (in seconds).

fs — is the set of semicolon-separated file descriptors for source files in the time sharing ASCII standard system format, in the standard BCD format, in COMDK form, and/or descriptors for binary card image object files. These files serve as inputs to the compiler and/or time sharing loader. When a BCD or COMDK source file is supplied, fs may also include a descriptor for an alter file in BCD format. The $ ALTER file must begin with a $ UPDATE card and must be in alter number sequence. If there is more than one BCD or COMDK source file in the list, the alter file updates the first. The list fs can also consist of a single file descriptor that points to a previously generated system loadable (H*) file. Concatenation of source files is provided by using a separate semicolon between each file descriptor.

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 time sharing loader is saved if the compilation is successful. If the named file does not exist, a permanent (quick access) random file of 36 llinks is created and added to the users' catalog. If the field is missing, no temporary H* file is created; instead, 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 the named file does not exist, a quick access permanent file of three llinks is created. This file expands as necessary up to a maximum of 20 llinks to hold the object deck(s). When C* is specified, a compiler temporary file (*1 scratch) file of 48 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 time sharing loader is called. This is required for compile, compile and load, and load activities; it is not required (or interpreted) for execute only runs (from 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.
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	-	Optimizer phase is called.
NOPTZ	-	Optimizer phase is not called.
NWARN	-	Do not print any compilation warning messages.
FDS	-	Enables the FORTRAN Debugging System (FDS). See Appendix F.

The following remaining options concern the loading process:

GO	-	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 saved. If no object (H*) save file is specified, only the compilation is performed.
ULIB	-	File descriptors 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. Up to nine user library files can be specified, separated by semicolons.
NOLIB	-	No user libraries are searched. 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.

ulib - A list of file descriptors (separated by semicolons) pointing to file(s) containing subprograms that have SYMDEF symbols that satisfy the undefined SYMREFs in the load table. The list must be provided by the user when the ULIB option is specified. The user library or libraries are searched in the order they are encountered and before the system subroutine library. Each user library must be created as a random permanent file by using the batch procedures UTILITY, RANLIB, and the Object File Editor.

#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 (see Time Sharing Command Language and File Usage in the _TSS General Information_ manual). The file description can be in any of the following formats:

1. _filename_ specifying a filename in the form nn where $01 \leq nn \leq 43$ and nn represents a logical file code referenced by the I/O statements in the program.

2. _filedescr_ specifying a full description.

   a. _filename_

   b. _filename$password_

   c. _userid/catalog$password_ ...

Example:

1. Create a random file to contain the user's library with the ACCESS subsystem. ACCESS CF,/ULIB1,B/50,50/,R,MODE/R/

2. Deck setup for creating and saving a user library file (through CARDIN 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/UL1B1
 $      FILE      R*,F1R,10L
 $      ENDJOB
```

Information Common to the FORTRAN and YFORTRAN Time Sharing Systems

Descriptions of compiler diagnostics are included in Appendix B.

For files required during execution the user will, most commonly, apply an alternate name specified in the following format.

filedescr "altname" where altname = nn; attaching the logical file code nn to the specified file.

File codes 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". For example:

```
RUN#"10"
```

If a given file descriptor consists of only an unquoted 2-digit logical file code, a temporary file is created for the user 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. For 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 directed to the logical file code 10 is routed to the temporary file.

The fe list of the RUN command serves two additional functions: creation of a file control block and 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 if so indicated) and added to the user's available file table (AFT). The file is then said to be 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. There are 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 (some allocated file has a 2-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 (there has been no file allocation for the current file designator), a comment is displayed on the terminal identifying the undefined file designator as follows:

```
FILE XX NOT IN AFT. ACCESS CALLED
```

XX is the 2-digit file designator being referenced by the running program. At this point, the ACCESS subsystem is called (as indicated by the above message) and the following is displayed (by ACCESS):

```
FUNCTION?
```

Commands can now be given to ACCESS. When the dialogue 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 the user accessed some file), 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. For example, consider that PROGRAM contains I/O statements with a file designator of 10 and the following dialogue has transpired:

```
SYSTEM? YFORTRAN or FORTRAN
OLD OR NEW -- OLD PROGRAM
READY
*RUN

FILE 10 NOT IN AFT. ACCESS CALLED

FUNCTION?
```

If the user responds with a carriage return, the terminal is used for file 10. If the user responds:

```
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.

Specify RUN Command as First Line of Source File

A user can include the RUN command as the first line or lines of his source file subject to the following restrictions:

1. This feature is available on time sharing ASCII files only.

2. The line can be in the current file (*SRC) or a referenced perm-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 *#; intervening blanks are not permitted.

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 a field separating delimiter such as the following: equal sign, left parenthesis, right parenthesis, comma, or pound sign.

6. The line(s) are treated as comment line(s) by the FORTRAN compiler.

7. The user can override the first-line contained RUN command by indicating save files, options, or concatenation on his RUN type-in.

The following example illustrates this capability.

```
*SYSTEM? YFORTRAN or FORTRAN
 OLD OR NEW?  NEW
*010*#RUN    *(20,30)=HSTAR(BCD,NOGO)
*020  PRINT, "HELLO DOLLY..."
*030  STOP; END
*RUN (Invokes first line syntax)
```

TSS Run Examples

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 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 user intends to execute the object file in the 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.

Additional examples are given in Section V under "File Designation" and in Time Sharing Applications Library Guide, Volume III Industry manual.

Batch Activity Spawned by the YFORTRAN Time Sharing System RUN Command

As an example of the simplest case, consider that some source file is current in *SRC, and a RUN command is typed with none of the optional fields. A job setup comparable to the following is dispatched to the batch system.

```
 $        SNUMB     nnnnT,40
 $        USERID
 $        IDENT
 $        LOWLOAD
 $        USE       .GRBG./36/
 $        OPTION    NOFCB,FORTRAN
 $        OPTION    NOGO,NOSREF,NOMAP,SAVE/OBJECT
 $        USE       .GTLIT,.TSGF.,.FTSU.,.FXEMA
A$        FORTY     NLSTIN,NFORM,ASCII
 $        LIMITS    2,26K
 $        FILE      S*,X1R          (source file *SRC)
 $        FILE      P*,X2S          (diagnostic report only)
A$        EXECUTE
 $        FILE      P*,X2R
 $        FILE      H*,X3R,3R       (bound program)
 $        ENDJOB
```

The results of compilation and loading are returned on files P* and H*. P* is read and scanned for compiler and/or loader diagnostics. These are displayed on the terminal and if there have been no fatal errors, the fully bound program is loaded from H* and execution proceeds.

Batch Activity to Build Time Sharing H* File

The following example program illustrates a method of building a time sharing H* file in batch.

```
 $        SNUMB
 $        IDENT
 $        LOWLOAD
 $        LIBRARY   DC,JT
 $        USE       .GRBG./36/
 $        OPTION    NOFCB,NOGO,SAVE/OBJECT
 $        USE       .GTLIT,.TSGF.,.FTSU.,.FXEMA
A$        FORTY     NFORM,NLND,ASCII,NWARN
 $        SELECTA   RDCNET/SSTAR
A$        EXECUTE   DUMP
 $        PRMFL     JT,R,R,FY/F8LIB
 $        PRMFL     DC,R,R,FY/ULIB
 $        PRMFL     H*,R/W,R,FY/HSTAR
 $        ENDJOB
```

NOTE: The inclusion of the $ USE .GRBG./36/ control card in this example causes the first 36 words in the blank common storage area to be lost.

Time Sharing System RUNL Command for Link/Overlay

A special form of the RUN command, RUNL, permits link/overlay H* files to be constructed. When a bound object program is too large for execution under time sharing, segmentation using the RUNL command offers an alternative.

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 used in the RUNL command.

The RUNL command has the form:

```
RUNL [H] C*file list = H*file (options) [ulib files]; link list
```

The command is RUNL or RUNLH. The latter form displays heading with date and time (and SNUMB if YFORTRAN).

C* file list	- The set of file descriptors for the binary object image 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 following 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+6K or 26K, whichever is larger. If not specified, nn is set to 20. The FORTRAN link loader memory requirement is nnK if < 23K or 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 syntax after creating the link/overlay H* file. For 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*. The link phrase has the following 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
```

Name1 (a five- or six-character constant or variable) is a unique identifier for the new link; name2, if present, is the identifier of 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. Entry, 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 cross-references between subprograms in 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 on Use of RUNL Command

1. To ascertain the size required to allocate to a permanent H* save file, create a temporary file by means of RUNL. Then use the LENGTH command to display "used" number of llinks. This number can be used as a current size on the permanent H* save file creation. A temporary H* file created by RUNL has a size of 216 llinks.

2. The "PSTR" load map generated by the General Loader can be sent to a remote station or central site printer, provided it is a permanent file. For example:

```
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. For example:

```
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)SUB7;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 "entry" field of the link phrase. LLINK can be called to load a particular link and return control to the part 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, 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.

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 and specify run time input/output files.

```
RUN HSTAR#INPUT"41";OUTPUT"13"
```

Examples of use of LINK/LLINK

1. Compile and save C* object deck files for main program and two subroutines.

```
010 PRINT,"MAIN EXECUTING"
020 CALL LLINK ("A    ")
030 CALL SUBA
040 CALL LINK ("B    ")
050 STOP;END

RUN =;MAIN(NOGO)


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 H* file.

```
RUN HSTAR
```

Example of Loader Input File

The following control card setup would appear on R* for the example above illustrating 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; in actual use, no underlining is done. Explanations are enclosed in parentheses; they are not part of the printout.

```
USER ID - J.P.JONES
PASSWORD--
XXXXXXXXXXXXXX
SYSTEM?YFORTRAN or FORTRAN
ØLD ØR NEW-NEW
READY
*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

READY
*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  (corrected version of program saved)

*BYE (finished)
**RESØURCES USED $  2.08, USED TØ DATE $  263.85= 27%
**TIME SHARING ØFF AT  15.421 ØN  10/10/74
```

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 (=), and sign-on (X-ON). The sign-on character activates the paper tape reader if the reader is in the ready state. A ready state 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). 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, facilitating preparation of 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. The terminal buffer is flushed. Control returns to the build-mode 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 either the end-of-tape occurs, the user turns off the reader, an X-OFF character is read by the paper tape reader, or a jammed tape causes a delay of over one second between the transmission of characters.

At present a maximum of 80 characters are permitted per line of paper tape input. Excessive 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.

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 CARDIN time sharing subsystem allows the user to submit a batch job from a time sharing terminal. This capability 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 best meets the needs of the application or that is most convenient for the normal mode of execution.

Specification of BCD or ASCII is possible in both batch and time sharing. In batch, the $ FORTY or $ 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. One exception exists, and that is when running in time sharing in the BCD mode, where the run consists of object decks only (no compilations required; not running a saved H*). In this case the BCD option must be given in the RUN command.

When BCD is elected, internal character data and FORMATS are carried in BCD; storage is allocated at a rate of six characters per word; and for I/O, ENCODE, PAUSE, etc., library calls are made to the entry names which work with BCD.

Similarly, when ASCII is elected, the object module will have all ASCII properties. Character data and FORMATs are carried in ASCII; storage is allocated at a rate of four characters per word; and, for I/O, ENCODE, PAUSE, etc., library calls are made to the entry names which work with ASCII.

Therefore, one generally cannot mix object modules of contradicting character sets. 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 internal 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 a BCD program under time sharing.
    a. Input and output can be directed to the terminal.
    b. Input files can be ASCII.
2. Execution of an ASCII program in the batch mode.
    a. Input and output can be directed to the reader, printer, punch, or SYSOUT.
    b. Input files can be BCD (media 0, 2, or 3) or ASCII.
3. Execution of a BCD program in the batch mode. Input files can be ASCII.

4. Execution of an ASCII program under time sharing. Input files can be ASCII or BCD (media 0, 2, or 3).

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

```
        $  USE  .GTLIT
```

In YFORTRAN time sharing, the RUN command places the $ USE .GTLIT control card on the R* file.

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

FILE FORMATS

All output files generated by FORTRAN, whether formatted or unformatted, ASCII or BCD, sequential or random[1], generated in time sharing or batch, 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 the 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 line printer (BATCH verb of SCAN).

[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".

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 $ FORTY or $ FORTRAN control card or the FORTRAN or YFORTRAN Time Sharing System 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.

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. For example:

1. 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. For example:

```
        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.

They are pre-calculated following the prologue. 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.

```
      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
```

The OPTZ generation sometimes produces a divide check when (A/B) is evaluated following the prologue; i.e., whenever B = 0.

This situation can be avoided in either of two ways.

a. The above 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 <u>only</u> referenced within the truth clause of a logical IF statement are <u>not</u> 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
```

2. Another situation results from using certain outdated library "flag" routines. For example, if a program uses FLGEOF, 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. For 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. For 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 of object code usually results.

BATCH COMPILATION LISTINGS AND REPORTS

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

The following listings and reports are produced when the indicated options are specified (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

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 DECK or COMDK is called for.

Figure 3-1 contains an example program with all reports. The following descriptions explain each report in more detail, using Figure 3-1 as a base for the description.

Source Program Listing (LSTIN)

Each line of this report, (page 1 of Figure 3-1), is divided into three fields. The left-most 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, F, or T to indicate a warning; a fatal error; or premature termination of compilation, respectively. A complete description of the diagnostics generated in the compiler is included in Appendix B.

In Figure 3-1, a warning diagnostic appears after line 5. Correct code is generated and the program runs as expected. To be error free, a specification statement should be added to the program typing EOF as INTEGER.

If the XREF option is on, the GMAP offset is printed in the leftmost column of the report. This gives the relative location in the object code of each executable source statement.

To-From Transfer Table (XREFS)

This table, page 2 of Figure 3-1, 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 GOTO statements, where no label list is provided, the label variable name is displayed. In Figure 3-1, page 2, lines 28 and 29 contain transfers. Line 29 includes the statement GOTO 7; statement 7 begins on line 10; the first entry in the transfer report indicates this path. Line 28 contains a STOP statement; the second entry in the transfer report indicates this. A From-To table is also provided in the same format.

If the line numbers of the source file are not sequentially increased by one, the actual line number is the one that has a value that is less than or equal to the line number printed.

Program Preface Summary (LSTOU)

The Program Preface summary, page 3 of Figure 3-1, 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 instructions with special (type 3) relocation.

The SYMDEFs entry shows the relative offset of the internal location corresponding to that symbol definition, in octal.

Next is a list of labeled common blocks known or referenced by this module. Associated with each symbol are three octal and one decimal fields. The first gives the global symbol number associated, for this compilation, with the common name. This is the number that will appear in the V field of any instruction referencing this labeled common region. The number will appear 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 will be 020000 (bit zero is the sign bit). If the V field is six bits wide, the display will be 010000. The second field contains the size, in octal, of the labeled common region. The third decimal field continues 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. The first, .DATA., is allocated space to contain all local data required by the program. This includes arrays and scalars not appearing in common or as arguments, constants, encoded FORMAT information, NAMELIST lists, temporary storage for intermediate results, argument pointers, the error linkage pair (.E.L..), etc. The second, .SYMT., is generated when the DEBUG option is employed. 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)

This report, page 4 of Figure 3-1, provides information on the allocation of storage for identifiable program elements. This report is divided into three parts: variables and arrays, statement numbers, and constants.

The first part of the report lists all program variables and arrays in alphabetical order. It contains four fields as follows:

1. The first field contains the global symbol name relative to which the 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.; 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 relative to the origin of the indicated global name assigned to this variable or array. For arrays, this is the starting location; subsequent elements of the array are allocated 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 either eight dots (........) if this is a main program, or the first SYMDEF if this is a subprogram. The OFFSET field is 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)

This report, pages 5-8 of Figure 3-1, 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; next is the compiled machine language instruction, usually divided into address, operation code, and modifier field. The location field and machine language instruction field are in octal. The next three digits are the relocation bits applicable to the instruction.

Following these is the symbolic equivalent of the generated instruction. This 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 except for 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 3-1.

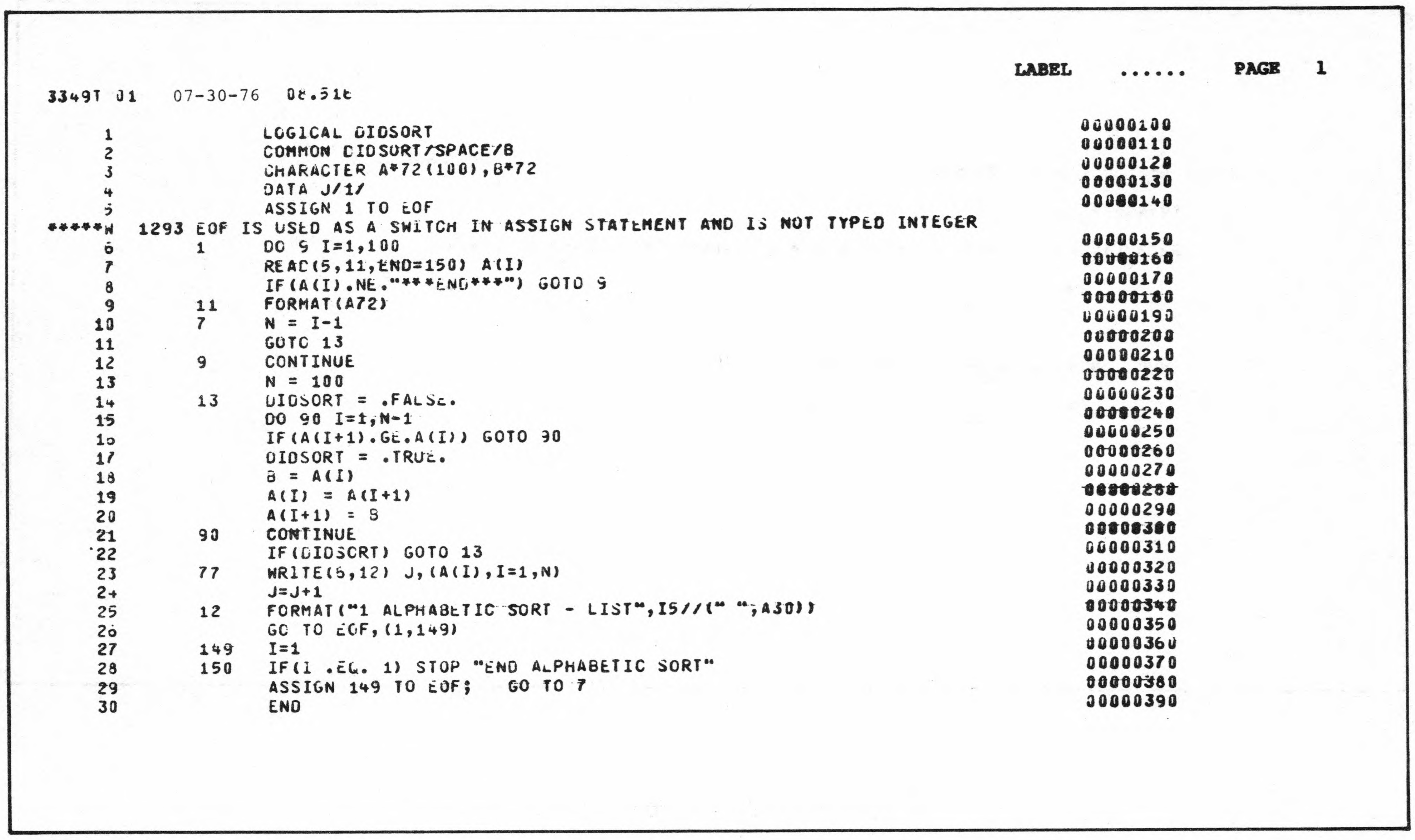
Cross Reference List (XREF)

This report, page 10 of Figure 3-1, 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. Where required, additional lines are written.

This report is divided into two parts: the second part for statement labels, the first part for everything else.

Miscellaneous Data

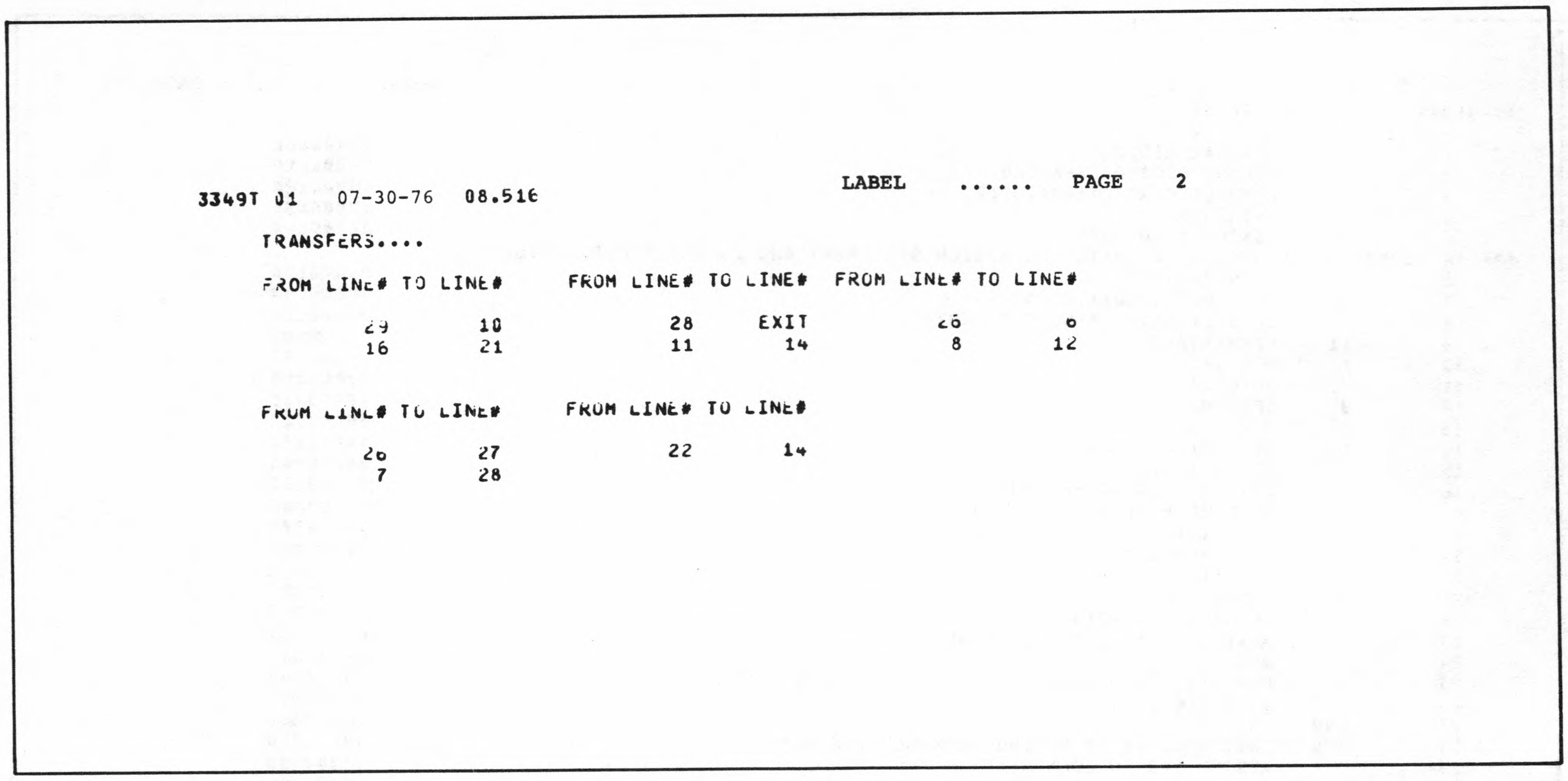
Additional compilation data is printed at the end of the report listing when the report is produced. This data consists of the edit date, the software release under which this report has been compiled, the processor time and compilation speed in terms of source lines per minute, the number of diagnostics printed, and the amount of memory space required for the compilation.



```
                                                                              LABEL   ......      PAGE   1
33491 01    07-30-76   08.51E

     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 3-1. Compilation Listings and Reports



```
3349T 01   07-30-76   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
```

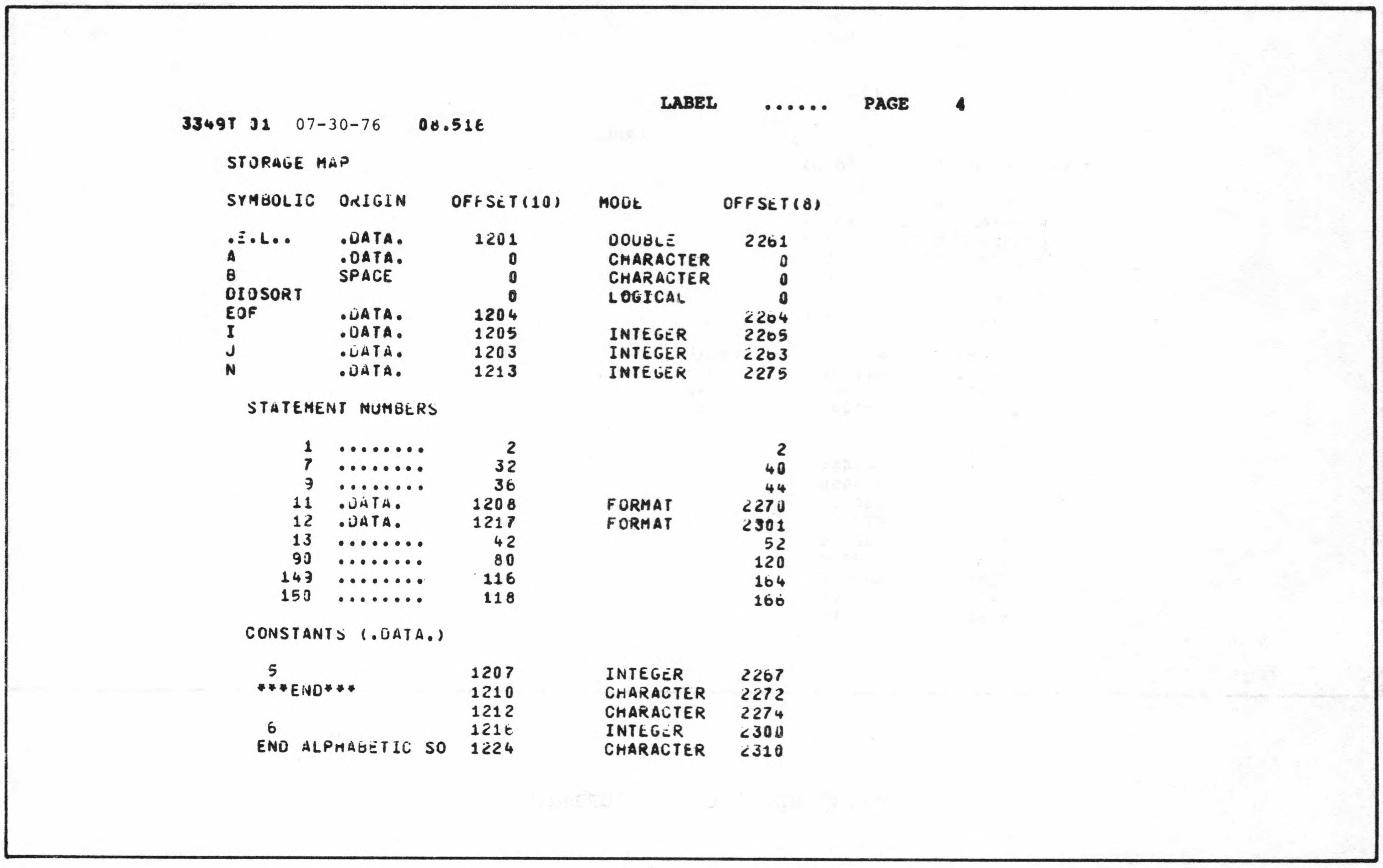
To-From Transfer Table

Figure 3-1 (cont). Compilation Listings and Reports

```
                                                  LABEL    ......   PAGE    3
3349T 01  07-30-76   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
      .FRDD.       140000
```

Program Prefix Summary

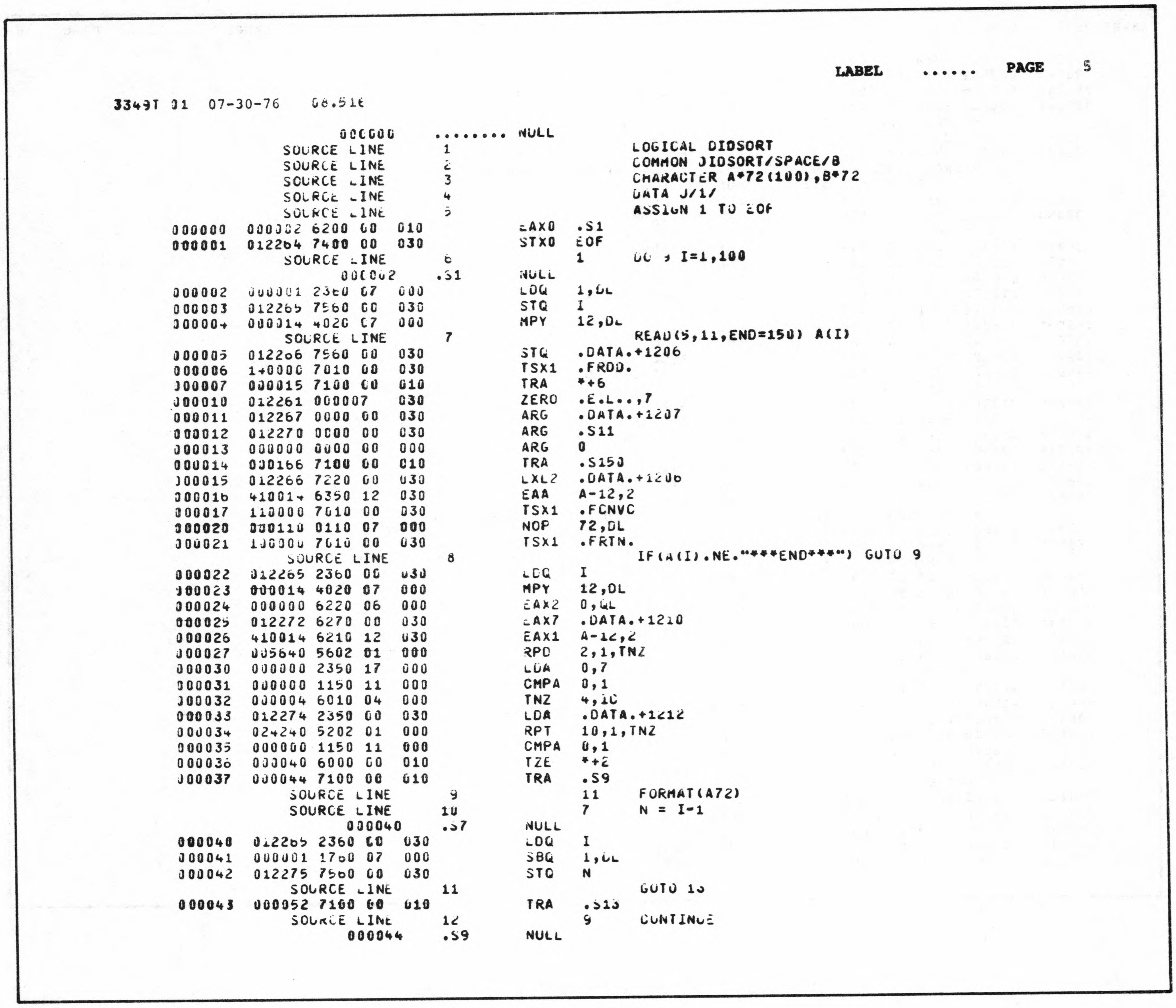
Figure 3-1 (cont). Compilation Listings and Reports



```
                                                          LABEL    ......   PAGE    4
3349T 01  07-30-76    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
   DIDSORT                   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

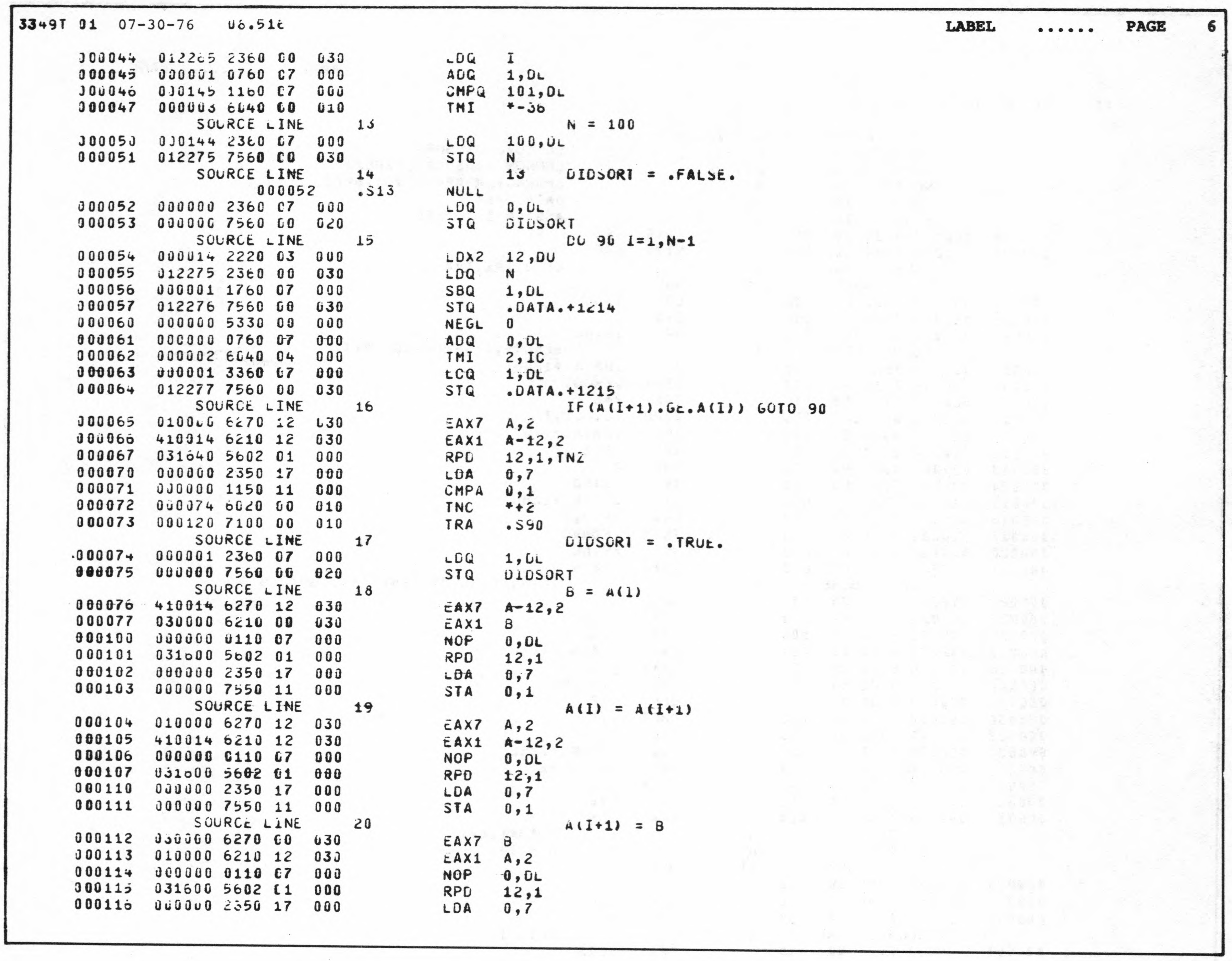
Figure 3-1 (cont). Compilation Listings and Reports



```
                                                                        LABEL    ......   PAGE    5
3349T 01  07-30-76    08.516

                     000000      ........ NULL
          SOURCE LINE         1                         LOGICAL DIDSORT
          SOURCE LINE         2                         COMMON JIDSORT/SPACE/B
          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  012206 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  100000 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 13
000043  000052 7100 00  010          TRA   .S13
          SOURCE LINE        12                  9       CONTINUE
                     000044   .S9        NULL
```

Object Program Listing

Figure 3-1 (cont). Compilation Listings and Reports



```
3349T 01  07-30-76    06.518                                                                     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(1)
     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
```

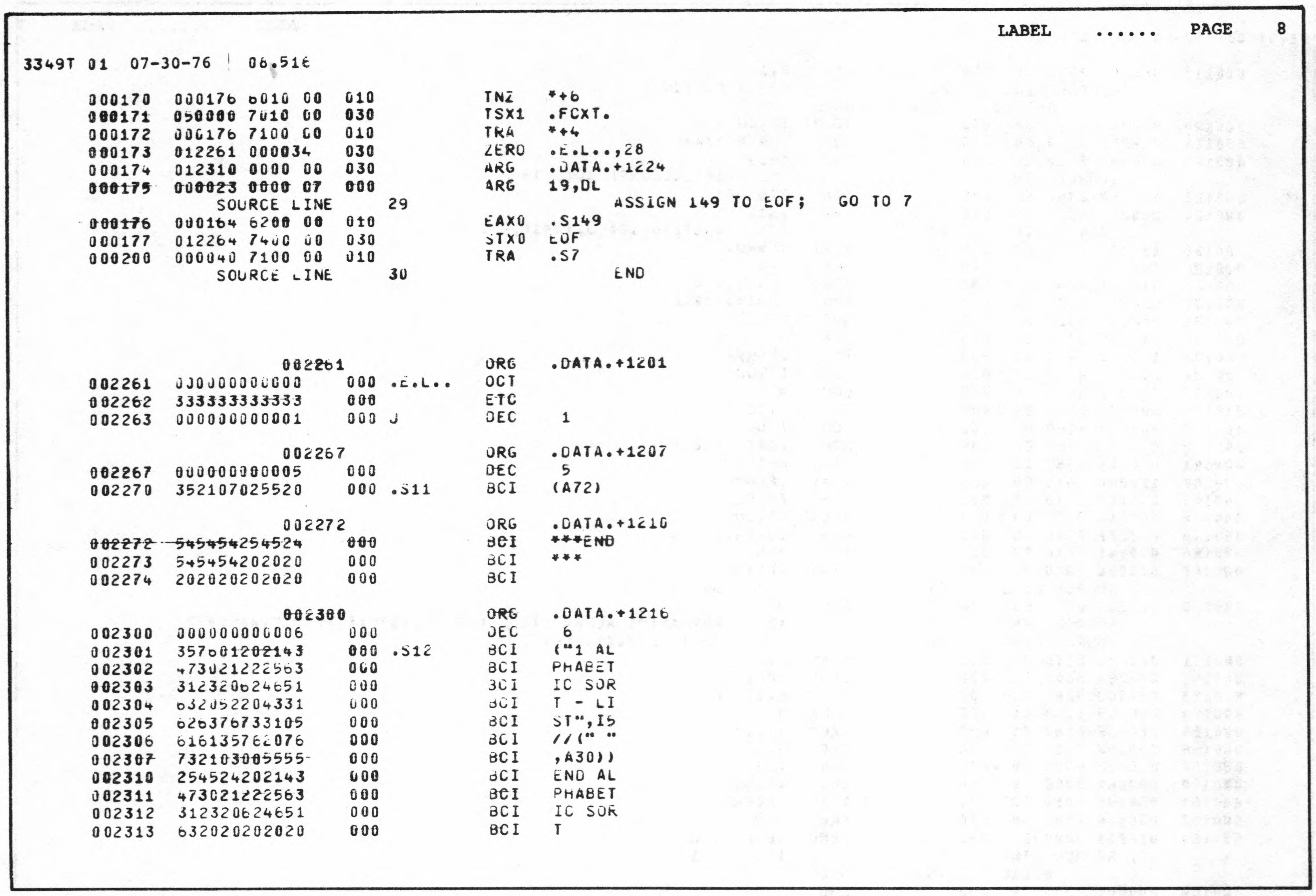
Object Program Listing (cont)

Figure 3-1 (cont). Compilation Listings and Reports

```
3349T 01  07-30-76    08.516                                                                    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(DIDSORT) GOTO 13
   000123   000000 2340 00  020          SZN    DIDSORT
   000124   000052 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 3360 00  030          LCQ    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 3-1 (cont). Compilation Listings and Reports



```
                                                                                      LABEL     ......    PAGE    8
3349T 01  07-30-76  | 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
```

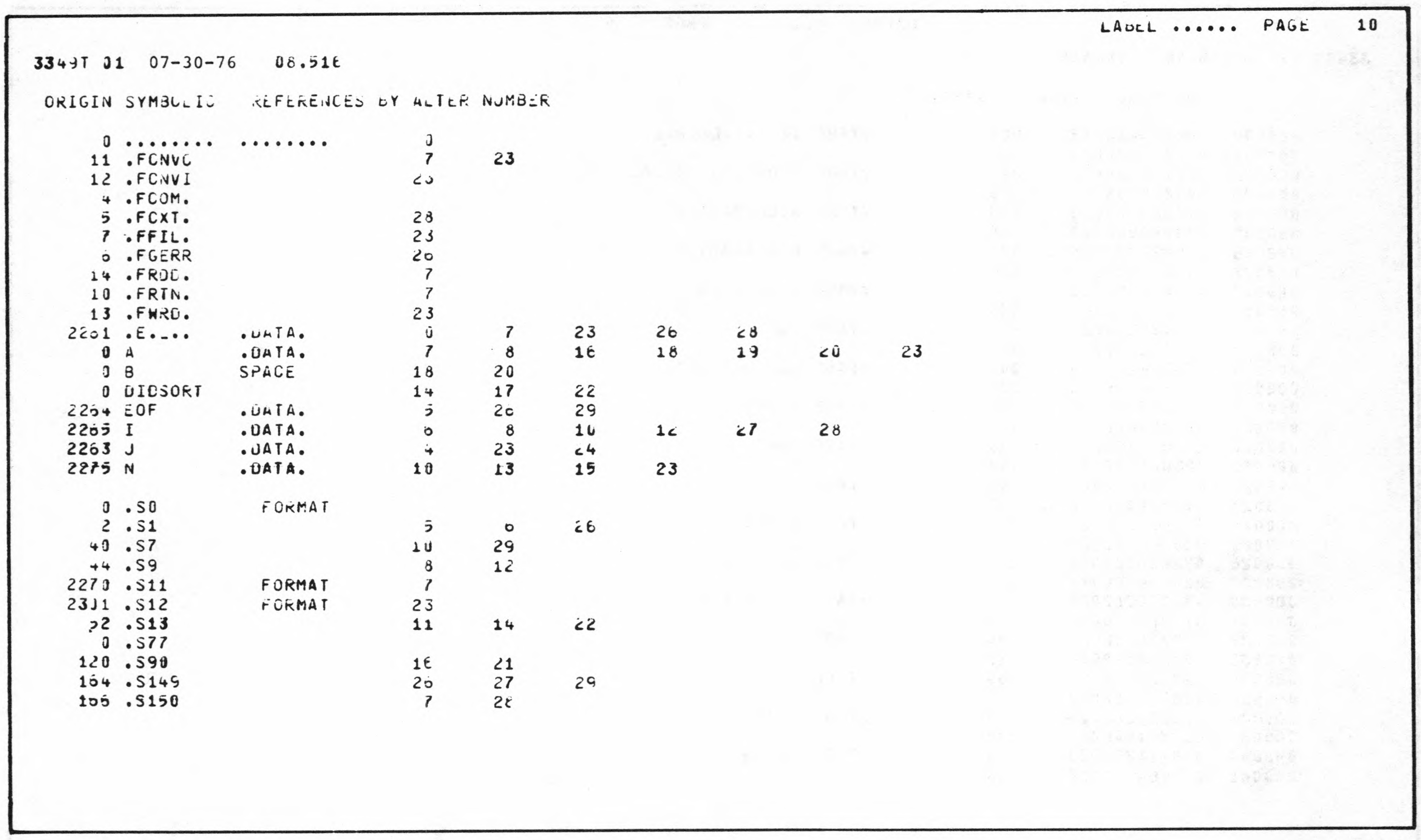
Object Program Listing (cont)

Figure 3-1 (cont). Compilation Listings and Reports

```
                                                        LABEL     ......     PAGE     9
 3349T 01   07-30-76    08.516

           DEBUG SYMBOL TABLE (.SYMT.)

      000000  332533433333    000            VTABF .E.L..,DOUBLE
      000001  012261000023    030
      000002  243124624651    000            VTABF DIDSORT,LOGICAL
      000003  000000000025    020
      000004  222020202020    000            VTABF B,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 3-1 (cont). Compilation Listings and Reports



```
                                                                                LABEL ......  PAGE     10
 3349T 01  07-30-76    08.516

  ORIGIN SYMBOLIC    REFERENCES BY ALTER NUMBER

       0 ........  ........          0
      11 .FCNVC                      7      23
      12 .FCNVI                     23
       4 .FCOM.
       5 .FCXT.                     28
       7 .FFIL.                     23
       6 .FGERR                     26
      14 .FRDC.                      7
      10 .FRTN.                      7
      13 .FWRD.                     23
    2261 .E....     .DATA.           0       7      23      26      28
       0 A          .DATA.           7       8      16      18      19      20      23
       0 B          SPACE           18      20
       0 DIDSORT                    14      17      22
    2264 EOF        .DATA.           5      26      29
    2265 I          .DATA.           6       8      10      12      27      28
    2263 J          .DATA.           4      23      24
    2275 N          .DATA.          10      13      15      23

       0 .S0          FORMAT
       2 .S1                         5       6      26
      40 .S7                        10      29
      44 .S9                         8      12
    2270 .S11         FORMAT         7
    2301 .S12         FORMAT        23
      52 .S13                       11      14      22
       0 .S77
     120 .S90                       16      21
     164 .S149                      26      27      29
     166 .S150                       7      28
```

Cross Reference List

Figure 3-1 (cont). Compilation Listings and Reports

```
          EDIT DATE    02-21-76     **SR3I**

       ELAPSED TIME   (SEC)      1.05         LINES/MINUTE      1704

         THERE WERE     1 DIAGNOSTICS  IN ABOVE COMPILATION
           26K WORDS WERE USED FOR THIS COMPILATION
```

Miscellaneous Data

Figure 3-1 (cont). Compilation Listings and Reports

SECTION IV

FORTRAN STATEMENTS

A FORTRAN statement is a sequence of syntactic items preceded by a keyword (see Table 2-1). The assignment statements are exceptions to this definition (see below).

The syntactic items are formed using letters, digits, and special characters of the FORTRAN character set (see Section II and Appendix A). The basic syntactic items of the FORTRAN language are constants, symbolic names, statement labels, keywords, operators, and special characters used for syntax punctuation.

Constants, symbolic names, operators, and the special characters are defined in Section II.

A statement label takes the form of an unsigned integer constant and is used to refer to individual statements. Any statement except an END statement may be labeled, but only labeled executable statements and FORMAT statements may be referenced.

A keyword consists of a specified sequence of letters; the keyword that begins a FORTRAN statement is used to identify that statement. For example, a DATA statement begins with the keyword DATA.

STATEMENT CLASSIFICATION

Each FORTRAN statement is classified as executable or nonexecutable. Executable statements specify activities to be accomplished. Nonexecutable statements describe the characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units, and specify entry points within subprograms.

ASSIGNMENT STATEMENTS

The execution of an assignment statement causes an entity to be defined (given a value). There are four types of assignment statements:

1. Arithmetic assignment statement.
2. Logical assignment statement.
3. Character assignment statement.
4. Label assignment (ASSIGN) statement.

Arithmetic Assignment Statement

An arithmetic assignment statement has the form

v = e

where: v is an unsigned variable name or array element name of an arithmetic type (integer, real, double-precision, complex).

e is an arithmetic expression.

An arithmetic assignment statement causes FORTRAN to compute the value of the expression on the right and to assign that value to the variable on the left of the equal sign.

*

The following examples show various arithmetic assignment statements:

where:

Rl and R2	are	real variables
Cl and C2	are	complex variables
D	is	a double-precision variable
I	is	an integer variable

Rl = R2	R2 replaces Rl
I = R2	R2 is truncated to an integer and stored in I.
Rl = I	I is converted to a real variable and stored in Rl.
Rl = 3*R2	The expression contains a real variable and an integer constant. The statement is compiled as Rl = 3.*R2.
Rl = R2*D+I	Multiply R2 by D using double-precision arithmetic, add the double-precision equivalent of I to that result, store the most significant part of the result as a real variable Rl.
Cl = C2* (3.7,2.0)	Multiply using complex arithmetic and store the result in Cl as a complex number.
C2 = R2	Replace the real part of C2 by the current value of R2. Set the imaginary part of C2 to zero.

Logical Assignment Statement

A logical assignment statement has the form:

```
v = e
```

where v is a logical variable name or logical array element and e is a logical expression. Thus if L1,L2, etc. are logical variables, logical assignment statements can be written:

```
L1 = .TRUE.
L2 = .F.
L3 = A.GT.25.0
L4 = I.EQ.0 .OR.A.GT.25.0
L5 = L6
```

The first two are the logical equivalent of statements of the form

```
variable = constant
```

L3 would be set .TRUE. if the value of the real variable A is greater than 25.0, and to .FALSE. if A is equal to or less than 25.0. L4 would be set to .TRUE. if the value of I was zero or A is greater than 25.0 and to .FALSE. otherwise. L5 would be set to the same truth value as L6 currently has.

Character Assignment Statement

A character assignment statement has the form

```
v = e
```

where v is a character variable name or character array element name and e is a character constant, variable, function or array element. Characters are stored left-adjusted in the destination location with trailing blanks if applicable. If the declared length of v is less than e, then e is truncated to the size of v for the assignment, and the leftmost characters are assigned. Thus if C1, C2, etc. are character variables, character assignment statements can be written:

```
C1 = "ABCD"           The four characters are assigned to variable C1.
C2 = C1
C3 = 'ABCDEFGHIJKLMNOP'
```

Label Assignment Statement

A label assignment statement has the form:

```
ASSIGN k TO i
```

where k is a statement number and i is a nonsubscripted switch variable. The statement number must refer to an executable statement in the same program unit in which the ASSIGN statement appears. For example:

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

Figure 4-1 presents an abbreviated summary of the legitimate combinations of expressions and variables in the assignment statements.

Variable	Expression						
	I	R	D	C	L	Ctr	T
I	I	I	I	I	N	N	I
R	R	R	R	R	N	N	R
D	D	D	D	D	N	N	N
C	C	C	C	C	N	N	N
L	N	N	N	N	L	N	L
Ctr	N	N	N	N	N	H	N

Legend	
I	= Integer
R	= Real
D	= Double Precision
C	= Complex
L	= Logical
Ctr	= Character
T	= Typeless
N	= Illegal

Figure 4-1. Legal Combinations of Assignment Statements

When the arithmetic assignment, logical assignment, and character assignment statements are executed, the evaluation of the expression 'e' and the alteration of the expression 'v' is performed in accordance with the rules given in Table 4-1.

Table 4-1. Rules for Assignment of E to V

IF V TYPE IS:	AND E TYPE IS:	THE ASSIGNMENT RULE IS:
Integer	Real	Fix and Assign
Integer	Integer	Assign
Integer	Double Precision	Fix and Assign
Integer	Complex	Fix the Most Significant Real Part and Assign
Integer	Character	Illegal
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	Illegal
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 4-1 (cont). Rules for Assignment of E to V

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.

STATEMENT FORMATS

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

ABNORMAL

The ABNORMAL statement has the following form:

```
ABNORMAL
   or
ABNORMAL  a1, a2, ..., an
```

where: a_i is a FUNCTION subprogram name whose characteristics are being qualified by this statement.

A function whose name appears in an ABNORMAL statement (or in an EXTERNAL statement with an ABNORMAL modifier) is treated as one whose references, for optimization purposes, cannot be treated the same as a variable or array element reference in an expression. That is, the function has side effects which may alter its arguments or locations in common, it performs I/O, or it is capable of returning different results even if the same actual arguments are given.

All functions should be analyzed to determine whether appropriate ABNORMAL statements should be included. If a subprogram has FUNCTION references, and none of the referenced functions are abnormal, it may be useful to include an ABNORMAL statement in the subprogram. The first form with no list may be used for this purpose and has the effect of setting all functions 'normal'.

If no functions are typed as ABNORMAL in a given subprogram, then all functions are treated as ABNORMAL, with the exception of the supplied functions listed in Tables 6-1, 6-2, and 6-3. The appearance of an ABNORMAL statement reverses the default interpretation, and all nonqualified functions are treated as normal.

Subroutines, as referenced by CALL statements, are always considered ABNORMAL.

This discussion, and the ABNORMAL statement itself, applies only to programs being compiled with the OPTZ option. When optimization is not performed, the presence or absence of ABNORMAL statements is immaterial.

ASSIGN

The ASSIGN statement has the following form:

```
ASSIGN  i  TO  j
```

where: i is an executable statement label.

j is an integer switch variable.

The ASSIGN statement assigns the value of a statement label to a switch variable.

Example:

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

The next statement to be executed would be statement number 17.

BACKSPACE

The BACKSPACE statement has the following form:

```
BACKSPACE f
```

where: f is the file reference.

The BACKSPACE statement is applicable only for sequential files.

When the BACKSPACE statement is executed, the file is positioned so that the record that was the preceding record prior to execution becomes the next record. If the last READ statement resulted in an end-of-file condition, two BACKSPACE commands are required to cause the file to be positioned prior to the last logical record. If the file is positioned at its initial point, the BACKSPACE statement has no effect.

If the device is tape, one BACKSPACE command causes an input file to be set as an output file, thereby making the execution sequence READ, BACKSPACE, READ illegal.

BLOCK DATA

One way to enter data into a labeled common block during compilation is by using a BLOCK DATA subprogram. (Data cannot be entered into blank common by the use of BLOCK DATA.) This subprogram can contain only type, EQUIVALENCE,PARAMETER, IMPLICIT, DATA, DIMENSION, and COMMON statements in addition to the BLOCK DATA and END statements.

A BLOCK DATA statement is of the form:

BLOCK DATA

The following rules also apply:

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 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 at most 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.

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 such subprogram is retained for any area of a common block that is referred to more than once.

Example of BLOCK DATA

```
      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
```

This example contains two labeled common blocks, ELN and RNC. All variables in each block must be listed even though not all variables receive values from the DATA statement. (The variable A in the example does not appear in the DATA statement.)

CALL

The CALL statement is used to refer to a SUBROUTINE subprogram (see "SUBROUTINE Subprograms" in Section VI).

A CALL statement is one of the forms:

```
CALL s(a1,...,an)
or
CALL s
```

where s is the name of a SUBROUTINE and the a_i are actual arguments or alternate returns.

The execution of a CALL statement references the designated subroutine. Execution of the CALL statement is completed upon return from the designated subroutine.

Example: `CALL MATMISL(A,B,C,I,J,K)`

Execution of the user program continues with the first executable statement of the SUBROUTINE (or SUBROUTINE entry point) MATMISL.

Additional examples:

```
CALL MATMPY(X,5,10,Y,10,2)
CALL QDRTIC(9.732,Q/4.536,R-S**2,X1,X2)
CALL OUTPUT
CALL ABC(X,B,C,$5,$200)
```

The CALL statement transfers control to a SUBROUTINE subprogram and presents it with the actual arguments. For purposes of optimization, all subroutine calls are treated as abnormal function references.

The arguments can be any of the following:

1. A constant.
2. A subscripted or nonsubscripted variable or an array name.
3. An arithmetic or logical expression.
4. The name of a FUNCTION or SUBROUTINE subprogram.

5. An omitted argument can be indicated by successive commas in the argument list. A reference to an omitted argument by the called subprogram is undefined.

6. $n where n is a statement number or a switch variable for a nonstandard return.

The arguments presented by the CALL statement must agree in number, order, type, and array size (except as explained under the DIMENSION statement) with the corresponding dummy arguments in the SUBROUTINE or ENTRY statement of the called subprogram.

The calling arguments generated for nonstandard returns in object program coding are listed in the reverse order from the way they appear in source program coding; this reverse order must be considered if subroutines written in GMAP are to be called by FORTRAN programs.

Example:

```
     CALL SUB (A,I,$10,$20)

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

CHARACTER

The CHARACTER statement is a form of the explicit type statement. It has the form

CHARACTER *b a_1 *s_1 $(k_1)/d_1/,\ldots,a_n$ *s_n $(k_n)/d_n/$

where:

b is a positive integer constant which defines the maximum number of characters ($\leq$ 500 characters in the ASCII mode and $\leq$ 511 characters in the BCD mode) of all variables in the statement unless otherwise specified by s_i.

a_i is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement.

s_i is the maximum number of characters that can be contained by the CHARACTER element being defined. An adjustable size specification is permitted within a subprogram when both the character variable and its size parameter(s) are included as dummy arguments. For example:

```
SUBROUTINE  MOVE (A,I,J,B,K)
CHARACTER  A*I(J,4),B*I
B = A(K,2)
RETURN
END
```

In this example, the number of characters associated with A and B are variable.

Adjustable size specifications are not allowed for the following:

1. As the size specification for a character function.
2. As the size specification in an IMPLICIT statement.
3. For types other than CHARACTER.

k_i supplies the dimension information necessary to allocate storage to arrays.

d_i is the initial data value.

If a compare is made of character fields of unequal length, the shorter field is left-justified and blank-filled to make the shorter field the same length as the larger field, after which an equivalence comparison is made.

The CHARACTER statement is more fully described under the type statement explanation in this section.

COMMON

A COMMON statement is of the form:

COMMON/x_1/a_1.../x_n/a_n

where each a_i is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x_i is a symbolic name or is empty. If x_i is empty, the first two slashes are optional. Each x_i is a block name that bears no relationship to any variable or array having the same name. COMMON assigns two elements in different subprograms or in a main program and a subprogram to the same location(s).

All variables named in a COMMON statement are assigned to storage in the sequence in which the names appear in the COMMON statement. For example if the following statement appeared in the main program:

```
COMMON A,B,C,D
```

the four variables are assigned to storage locations in the order named in a special section of storage called unnamed or blank common. Thus A is a specific storage location followed by B, etc. If in a subprogram we have the statement:

```
COMMON W,X,Y,Z
```

it means W is assigned the first location in blank common, and X the next, etc. Since the storage assigned to blank common is the same for the subprogram as the main program, A and W, B and X, C and Y, and D and Z share the same locations.

Additional blocks of storage can be established by labeling common. Labeled common is established by writing the label between two slashes as follows:

```
COMMON/X/A,B,C
```

Labeled and blank common can be included in the same statement. For example, if the following two statements were to appear in a main program and in a subprogram:

```
COMMON A,B,C/Y1/D,E/Y2/F(50),G(3,10)
COMMON H,I,J/Y1/K,L/Y2/M(50),N(3,10)
```

Blank common would contain A,B,C (in that order) in the program containing the first COMMON statement and H,I,J in the program containing the second. A and H would be assigned the same location as would B and I, and C and J. The common block labeled Y1 would establish D and E in the same locations as K and L. Y2 in the first program contains the 50 locations of F and the 30 locations of G. The same 80 locations are assigned to M and N in the second program. The following rules apply:

A double precision or complex entity is counted as two logically consecutive storage units. A logical, real, or integer entity is one storage unit. A character entity is given as many consecutive storage units as are required to contain the specified number of characters.

The following applies to labeled common blocks with the same number of storage units or to blank common:

1. 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 quantity.

2. 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.

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

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

COMPLEX

The COMPLEX statement is an explicit type statement with the following form:

COMPLEX $a_1*s_1(k_1)$ $/d_1/,\ldots,$ $a_n*s_n(k_n)/d_n/$

where

a_i	is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement.
$*s_i$	is an optional size-in-bytes qualification and is ignored.
k_i	supplies the dimension information necessary to allocate storage to arrays.
d_i	represents the initial data value.

The COMPLEX statement is more fully described under the Type statement in this section.

CONTINUE

The CONTINUE statement is a dummy statement most often used as the last statement in the range of a DO, when the last statement would otherwise have been a GO TO or IF. (See description of the DO statement in this section). It has the following form:

```
     CONTINUE
```

For example:

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

Execution of this statement causes a continuation of the normal execution sequence.

DATA

A data initialization statement is of the following form:

DATA $k_1/d_1/,k_2/d_2/,\ldots,k_n/d_n/$

where each k_i is a list containing names of variables, arrays, array elements and implied DOs. Each d_i is a list of optionally signed constants of the form:

C or J* C

where C is a constant and 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.

The DATA statement enables the programmer to enter data into the program at the time of compilation. For example:

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

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

The following is an additional example:

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

This makes ZERO the value zero, puts 1.0 in the first five elements of A, and 100.5 in A(9).

The following rules apply:

1. Dummy arguments and names in blank common cannot appear in the list k_i.

2. Any subscript expression must be an integer expression of the form C_1 $*V \pm C_2$ where C_1 and C_2 are unsigned integer constants or parameters and V is an integer variable that appears as the induction variable of some enclosing implied DO.

3. When J* appears ahead of a constant, it indicates the constant is to be applied J times, i.e., it initializes the next J items in the list with C_i.

4. Any type of constant can appear in the list d. However, type checking is performed to verify that a variable is being initialized with a constant of the same type, subject to the condition in rule 5.

5. Constants of type octal or character can be used to initialize variables of any type.

6. Character variables are initialized with character constants, and truncation or blank-filling can take place if the sizes of the two differ.

7. 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 longer than one word of storage can accommodate, then the variable must appear as an array element reference. The constant is assigned to consecutive locations in memory beginning with referenced location in the array. Thus in the example:

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

   there is a one-to-one relationship between the two lists (one variable, one constant) but locations G(1), G(2), G(3) and possibly G(4) (if the mode is ASCII) are affected.

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

9. Where data is to be compiled into an entire array, the name of the array (with indexing information omitted) can be placed in the list. The number of data literals must exactly equal the size of the array. For example, the statements

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

   define the values of A, B(1), ..., B(23) to be 4.0 and the values of B(24), B(25), and C to be 3.0, 2.0, and 1.0 respectively.

10. DATA statements appearing in a BLOCK DATA subprogram can pre-set data into labeled common storage only. A maximum of 63 such common areas can be pre-set from any one BLOCK DATA subprogram.

11. DATA statements appearing in other than a BLOCK DATA subprogram can pre-set data into program storage local to that subprogram, or labeled common. A maximum of 62 such common areas is permitted.

12. The type statements, described in this section, can also be used to initialize data values, and are subject to the same rules as given here for the DATA statement.

DECODE

A DECODE statement has the following form:

DECODE (a,t,optl) list

where: a is a character scalar, array element, or an array of any type that indicates the starting location of the internal buffer.

t can be a FORMAT statement number, a character scalar, or an array name that provides the format information required for decoding.

optl 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.

list has the same requirements as the list specified for the READ statement.

The DECODE statement causes the character string beginning at location a to be converted to data items according to the format specified by t; and stored in the elements of the list.

The format information and list should not require more characters than are in a.

Example:

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

After execution, the array A is not altered but the variable I contains an integer one.

Example:

```
10        CHARACTER A*4(4),B*1(16)
20        DATA A/4*"ABCD"/,B/16*"X"/
30        DECODE (A,4,ERR=100)B
40      4 FORMAT(4A1)
50        GOTO 11
60    100 PRINT,"ERROR"
70        STOP
80     11 PRINT 9,B
90      9 FORMAT(1X,16A1)
100       STOP
110       END

*RUN
ABCDABCDABCDABCD
```

Array A is placed into array B.

Additional information on the DECODE statement is contained in Section V under "Internal Data Conversion".

DIMENSION

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program, and it defines the maximum size of the arrays. An array can be declared to have from one to seven dimensions by placing it in a DIMENSION statement with the appropriate number of subscripts appended to the variable. The DIMENSION statement has the form:

DIMENSION $v_1(i_1)/d_1/,v_2(i_2)/d_2/,...v_n(i_n)/d_n/$

Each v_i is an array declarator (see "Variables" in Section II) with each v being an array name. Each i_i is composed of from one to seven unsigned integer constants, integer parameters, or integer variables separated by commas. Integer variables can be a component of i_i only when the DIMENSION statement appears in a subprogram, and the array can not be in COMMON. Each $/d_i/$ represents optional initial data values. The form for each $/d_i/$ is as specified for the DATA statement.

1. The DIMENSION statement must precede the first use of the array in any executable statement.

2. A single DIMENSION statement can specify the dimensions of any number of arrays.

3. If a variable is dimensioned in a DIMENSION statement, it must not be dimensioned elsewhere.

4. Dimensions can also be declared in a COMMON or a Type statement. If this is done, these statements are subject to all the rules for the DIMENSION statement.

5. The initial data values are optional, and if specified, apply to the array immediately preceding their declaration.

In the following examples A, B, and C are declared to be array variables with 4, 1, and 7 dimensions respectively. Note that each element of array B is initialized to contain the value 1.

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

DO

This statement enables the user to execute a section of a program repeatedly, with automatic changes in the value of a variable between repetitions. The DO statement can be written in either of these forms:

```
DO n i = m1,m2
or
DO n i = m1,m2,m3
```

In these statements, n must be a statement number of an executable statement, i must be a nonsubscripted integer variable, and m_1,m_2,m_3 can be any valid arithmetic expression. If m_3 is not stated, it is understood to be 1 (first form). These parameters (m_1,m_2,m_3) are truncated to integers before use.

The statements following the DO up to and including the one with statement number n are executed repeatedly. They are executed first with i = m_1; before each succeeding repetition i is increased by m_3 (when present, otherwise by 1); when i exceeds m_2 execution of the DO is ended, and execution continues with the first executable statement following statement n.

1. The terminal statement (n) cannot be a GO TO (of any form), RETURN, STOP, or DO statement.

2. The terminal statement (n) can be an arithmetic IF statement with at least one null label field. The null path is a simulated CONTINUE statement terminating the DO.

3. The range of a DO statement includes the executable statements from the first executable statement following the DO to and including the terminal statement (n) associated with the DO.

4. Another DO statement is permitted within the range of a DO statement. In this case, the range of the inner DO must be a subset of the range of the outer DO.

5. The values of m_1, m_2 and m_3 must all be nonnegative and m_3 cannot be zero; m_1 cannot be the constant zero but can be a variable whose value is zero. If m_2 is less than or equal to m_1 the loop will be processed once.

6. None of the control parameters, i, m_2, or m_3, can be redefined within the loop or in the extended range of the loop, if such exists.

A completely nested set of DO statements is a set of DO statements and their ranges such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement.

If a statement is the terminal statement of more than one DO statement, the statement number of that terminal statement cannot be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the innermost DO with that as its terminal statement.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described in the following steps:

1. The induction variable, i, is assigned the value represented by the initial parameter (m_1).

2. Instructions in the range of the DO are executed.

3. After execution of the terminal statement the induction variable of the most recently executed DO statement associated with the terminal statement is changed by the value represented by the associated step parameter (m_3).

4. If the value of the induction variable after change is less than or equal to the terminal value, then the action described starting at the 2nd step is repeated, with the understanding that the range in question is that of the DO, whose induction variable has been most recently changed. If the value of the induction variable is greater than the terminal value, then the DO is said to have been satisfied.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the induction variable of the next most recently executed DO statement is changed by the value represented by its associated step parameter and the action described in the 4th step is repeated until all DO statements referring to the particular termination statement are satisfied, at which time all such nested DO's are said to be satisfied and the first executable statement following the terminal statement is executed.

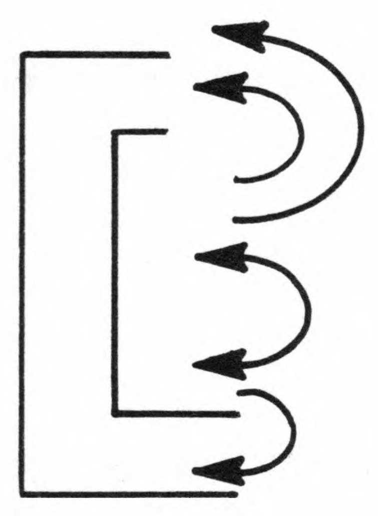
   (In the remainder of this section a logical IF statement containing a GO TO or an arithmetic IF as its conditional statement is regarded as a GO TO or an arithmetic IF statement, respectively.)

6. Upon exiting from the range of a DO by the execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the induction variable of the DO is defined and is equal to the most recent value attained as defined in the preceding paragraphs. The induction value is always undefined if the upper limit of the DO is reached; i.e., the DO is satisfied.

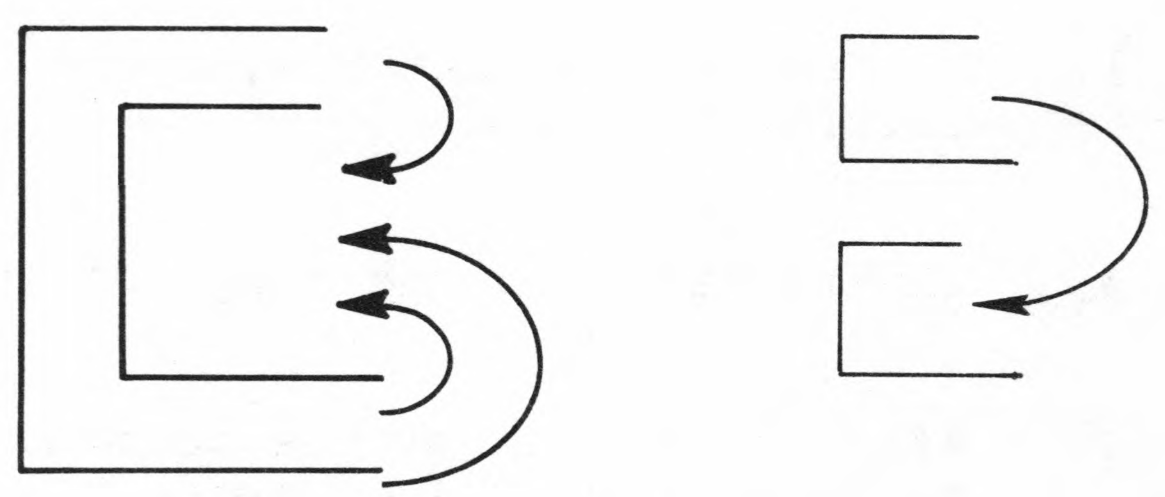
Transfer of Control

The following configurations show permitted and nonpermitted transfers.

Permitted



Not Permitted



An example of the DO statement follows:

```
    K = 0
    DO 10 I = 1,3
    DO 10 J = 1,2
    K = K + I + J
10 CONTINUE
```

where the K values are computed as:

```
       OLD          NEW
        K    I    J    K

     K = 0
     K = 0 + 1 + 1 = 2
     K = 2 + 1 + 2 = 5
     K = 5 + 2 + 1 = 8
     K = 8 + 2 + 2 = 12
     K =12 + 3 + 1 = 16
     K =16 + 3 + 2 = 21
```

Extended Range

A DO statement is said to have an extended range if the following two conditions exist:

1. There exists at least one transfer statement inside a DO that can cause control to pass out of this DO, or out of the nest if the DO is nested.

2. There exists at least one transfer statement, not inside any other DO, that can cause control to return into the range of this DO.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that can be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is.

A transfer statement cannot cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a DO that has an extended range or a DO with the same induction variable.

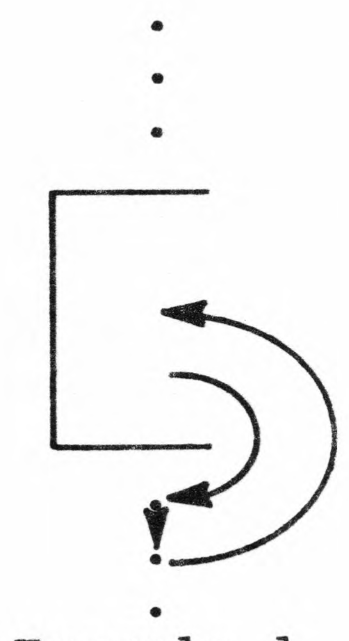
When a procedure reference occurs in the range of a DO the actions of that procedure are considered to be temporarily within that range; i.e., during the execution of that reference.

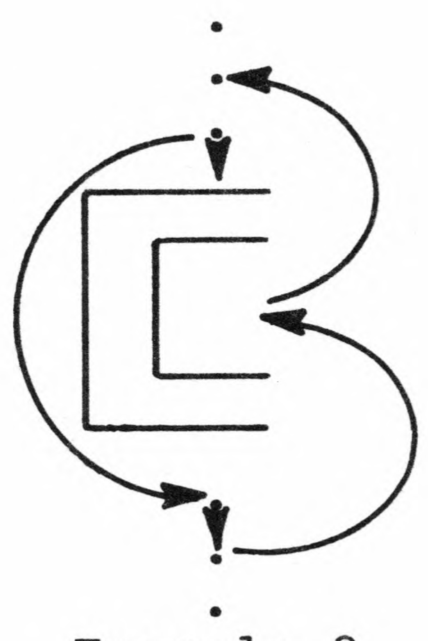
NOTE: Use of extended range DO's should be minimized, especially when global optimization is desired.

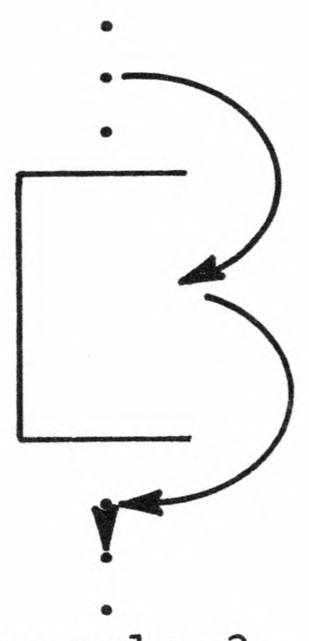
Example:

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

The following illustrate usage of the extended range of a DO. Examples 1 and 2 are permitted; example 3 is not permitted.


Example 1
Permitted


Example 2
Permitted


Example 3
Not permitted

DOUBLE PRECISION

The DOUBLE PRECISION statement is an explicit type statement with the following form:

DOUBLE PRECISION $a_1{*}s_1(k_1)$ $/d_1/,\ldots,$ $a_n{*}s_n(k_n)/d_n/$

where

a_i	is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement.
$*s_i$	is an optional size-in-bytes qualification and is ignored.
k_i	supplies dimension information necessary to allocate storage to arrays.
d_i	represents initial data value(s).

This statement is used to declare real data with extended precision. Such data can also be declared via the REAL statement with a size qualifier of 8 or more.

The DOUBLE PRECISION statement is more fully described under the Type statement in this section.

ENCODE

The ENCODE statement has the following form:

ENCODE (a,t,optl) list

where: a is a character scalar, array element, or an array of any type that indicates the starting location of the internal buffer.

t can be a FORMAT statement number, a character scalar, or an array name that provides the format information of the sending field for encoding.

optl is the error transfer option, designated as ERR=Sl, where Sl is the statement label or switch variable that is to receive control when an error condition is encountered.

list has the same requirements as the list specified for the WRITE statement.

The ENCODE statement causes the data items specified by list to be converted to the character mode under control of the format specified by t; and placed in storage beginning at location a.

The number of characters generated by the format information and the list should not be greater than the size of a.

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, array A contains (beginning with the first character position of A(1)):

b̸b̸b̸1

where b̸ indicates a blank.

Additional information concerning the ENCODE statement is given in Section V under "Internal Data Conversion."

NOTE: Any numerical variable in the list whose value is such that it requires more spaces than are provided in the format specified by i will be replaced by asterisks in the storage beginning at a, as described for output in the "Numeric Field Descriptors" paragraph of Section V. If such use is necessary, as when developing leading zeros for the character form of a numeric, then judicious use of CALL NASTRK and CALL YASTRK statements (refer to the "Supplied SUBROUTINE Subprograms" paragraph of Section VI) will be required to allow ENCODE to function as intended.

END

The END statement specifies the physical end of the source program. It must be the last statement of every program and must be completely contained on that line. END creates no object-program instructions. It has the form:

```
     END
```

ENDFILE

This statement is operable only for sequential files. Its execution causes the indicated file to be closed with an end-of-file signal. For an output file, the buffer(s) is flushed and a file mark is written. Nothing is done for an input file. (The end-of-file signal is a unique record indicating demarcation of a sequential file.) This statement has the form:

```
ENDFILE f
```

where f is the file reference.

NOTE: If it is necessary to span two lines/cards for this statement, and if the break is between the letters D and F, then a comment line cannot appear between the initial and continuation lines. Specifically, the following is not permitted:

```
1    6
_______________________________
      END
C     COMMENT
     1 FILE 3
```

ENTRY

The general form of the ENTRY statement is:

```
ENTRY name (b1,b2,...,bn)
    or
ENTRY name
```

Name is the symbolic name of an entry point, unique within the first six characters.

Each b_i is a dummy argument corresponding to an actual argument in a CALL statement or in a function reference. An ENTRY into a FUNCTION subprogram must have at least one argument.

An ENTRY into a SUBROUTINE subprogram can have arguments of the form * indicating nonstandard returns (dummy statement references).

The following rules apply to the use of multiple entry points:

1. All of the rules regarding adjustable dimensions given with "Adjustable Dimensions", Section II.

2. 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.

3. An ENTRY name can appear in an EXTERNAL statement in the same manner as a FUNCTION or SUBROUTINE name.

4. Entry into a subprogram initializes all references in the entire subprogram from items in the argument list of the CALL or function reference. (For instance, if, in the example that appears in the section "Multiple Entry Points into a Subprogram" of Section VI, entry is made at SUB2, the variables in statement 10 will refer to the argument list of SUB2.)

5. The appearance of an ENTRY statement does not alter the rules regarding the placement of arithmetic statement functions in subroutines. Arithmetic statement functions can follow an ENTRY statement only if they precede the first executable statement following the SUBROUTINE or FUNCTION statement.

6. None of the dummy arguments of an ENTRY statement can appear in an EQUIVALENCE or COMMON statement in the same subprogram.

EQUIVALENCE

The EQUIVALENCE statement is of the form:

EQUIVALENCE $(k_1),(k_2),\ldots,(k_n)$

where each k_i is a list of the form:

$a_1,a_2,\ldots,a_n$

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only integer constants or parameter symbols, and m is greater than or equal to 2. The EQUIVALENCE statement causes two or more variables, or arrays, to be assigned to the same storage location(s). EQUIVALENCE differs from COMMON in that EQUIVALENCE assigns variables within the same program or subprogram to the same storage location; COMMON assigns variables in different subprograms or a main program and a subprogram to the same locations.

One EQUIVALENCE statement can establish equivalence between any number of sets of variables. For example:

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

In this example, part of the arrays C and D are to be shared by other variables. Specifically, the variable A is to occupy the same location as the array element C(5,4), and the array B is to begin in this same location; the variable E shares location D(1,4,3) of the D array.

The following rules apply:

1. Each pair of parentheses in the statement list encloses 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 using the EQUIVALENCE statement with subscripted variables, two methods can be used to specify a single element in the array. For example, D(1,2,1) or D(p) may be used to specify the same element, where D(p) references the p_i element of the array in storage. (See the discussion "Array Element Successor Function" in Section II.)

3. Quantities or arrays that are not mentioned in an EQUIVALENCE statement are assigned unique locations.

4. Locations can be shared only among variables, not among constants.

5. The sharing of locations requires a knowledge of which FORTRAN statements cause a new value to be stored in a location. There are six such statements:

   a. Execution of an arithmetic assignment statement stores a new value in the location assigned to the variable on the left side of the equal sign.

   b. Execution of a DO statement or an implied DO in an input/output list sometimes stores a new indexing value.

   c. Execution of a READ or DECODE statement stores new values in the locations assigned to the variables mentioned in the input list.

   d. Execution of an ENCODE statement stores new values in the character variable or array locations named as the internal buffer.

   e. Execution of a CALL statement or an abnormal function reference may assign new values to variables in common or to arguments passed to that subprogram.

   f. An initial value can be stored in some location via a DATA statement, or a Data clause in a type statement.

6. Variables brought into a common block through EQUIVALENCE statements can increase the size of the block indicated bv the COMMON statements, as in the following 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. Since arrays must be stored in consecutive forward locations, a variable cannot be made equivalent to an element of an array in such a way as to cause the array to extend below the beginning of a common block.

8. The rule for making double-word variables equivalent to single-word variables is:

   a. The effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of locations away from the start of the data space to which it is allocated (common or local).

   b. The effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of words away from the start of any other double-word variable linked to it through EQUIVALENCE statements.

9. Two variables in one common block or in two different common blocks must not be made equivalent.

10. The EQUIVALENCE statement does not make two or more elements mathematically equivalent.

11. Equivalenced variables must not appear as dummy arguments in a FUNCTION, SUBROUTINE, or ENTRY statement.

EXTERNAL

The EXTERNAL statement has the following form:

EXTERNAL a_1, a_2, ..., a_n

where

a_i is a subprogram name whose characteristics are being qualified by this statement.

each a_i may be of the form:

f or f(ABNORMAL)

where f is the subprogram name. Use of the second form serves to define the subprogram f as both EXTERNAL and ABNORMAL (see the ABNORMAL statement, in this section).

FORTRAN permits the use of a subprogram name as an argument in a subprogram call. When this is done, the name must be included in an EXTERNAL statement in the calling program to distinguish the FUNCTION or SUBROUTINE name from a variable name. The following example illustrates this use in a main calling program and a subroutine subprogram:

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 in conjunction with formatted input/output statements and the ENCODE and DECODE statements to provide conversion and editing information between the internal representation and the external character string.

A FORMAT statement has the form:

m FORMAT $(q_1 t_1 z_1 t_2 z_2 \ldots t_n z_n q_2)$

or

m FORMAT (V)

or

m FORMAT ( )

where

m is the statement number
q is a series of slashes or empty
t is a field descriptor or group of field descriptors
z is a field separator

The first form is used for formatted input/output under FORMAT control. The second form is used for formatted input/output under list control, and is generally called list directed input/output in this manual. The syntax of the READ, PRINT, and PUNCH statements make it possible to perform list directed I/O in either of two ways: by omitting a FORMAT reference (e.g., READ,) or by including a reference to a FORMAT statement of the second form. Only the second alternative is permitted when used in conjunction with a WRITE statement, since the syntax of WRITE requires a FORMAT reference. The third form is ignored and causes no action.

When the first form is used, the following field descriptors are permitted:

```
pr F w.d           }
pr E w.d           }
pr G w.d           } Numeric and Logical Field Descriptors
pr D w.d           }
r O w              }
r I w              }
r L w              }
r A w              }
r R w              }
w H hlh2 ... hw    }    Character Field Descriptors
"hlh2 ... hn"      }
'hlh2 ... hn'      }

Tt                 }    Field Positioning Descriptors
wX
```

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_i is a single character

t is a character position, where the positions of a line/card are numbered 1 through the number present.

The F, E, and G descriptors are for REAL values, D is for DOUBLE PRECISION, O is for octal conversion, I is for INTEGER, L is used with LOGICAL values, A, R and H are for CHARACTER values, X and T are for skipping over text. The following briefly describes how these descriptors are formed. Note that the last three, H, T and X, do not require a variable in the input/output list; all others do.

Fw.d	= Real mode without exponent
Ew.d	= Real mode with exponent
Gw.d	= F or E editing code is taken dependent on value of output item
Dw.d	= Double precision mode
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 logical variable
Aw	= Field occupies w print positions - Left justified data
Rw	= Field occupies w print positions - Right justified data
wH	= Hollerith field to occupy w print positions
Tt	= Next operation begins with position t of record
wX	= Field of width w is blank filled on output, skipped on input

See "Input and Output", Section V for details on the fields of the FORMAT statement.

FUNCTION

The FUNCTION statement is the first statement of a FUNCTION subprogram. The type of the function can be explicitly stated by preceding the word FUNCTION with the type, by the subsequent appearance of the function name in a type statement, or implicitly by the first letter of the function name. The FUNCTION statement has the forms:

FUNCTION name (a_1, a_2, ..., a_n)
REAL FUNCTION name (a_1, a_2, ..., a_n)
INTEGER FUNCTION name (a_1, a_2, ..., a_n)
DOUBLE PRECISION FUNCTION name (a_1, a_2, ..., a_n)
COMPLEX FUNCTION name (a_1, a_2, ..., a_n)
LOGICAL FUNCTION name (a_1, a_2, ..., a_n)
CHARACTER FUNCTION name (a_1, a_2, ..., a_n)

where

name is the symbolic name of a single-valued function

the arguments a_1, a_2, ..., a_n (there must be at least one) are non-subscripted variable or array names or the dummy name of a SUBROUTINE or FUNCTION subprogram.

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 IFTRU (D,E,F)
```

1. The FUNCTION statement must be the first statement of a FUNCTION subprogram. At least one dummy variable must be enclosed in parentheses.
2. The name of the function must appear at least once in some definitional context (see EQUIVALENCE statement in this section). This name cannot be used in a NAMELIST or COMMON statement.
3. Length of character function can be specified as in the following example:

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

Example:

```
FUNCTION CALC (A,B)
.
.
.
CALC=Z+B
.
.
.
RETURN
END
```

By this method the output value of the function is returned to the calling program.

The calling program is the program in which the function is referred to or called.

The called program is the subprogram that is referred to or called by the calling program.

3. The arguments can be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments must correspond in number, order, size and type with the dummy arguments.

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

5. None of the dummy arguments can appear in an EQUIVALENCE, NAMELIST, or COMMON statement in the FUNCTION subprogram.

6. The FUNCTION subprogram must be logically terminated by a RETURN statement (see "Returns from Function and Subroutine Subprograms", Section VI) 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. The actual arguments of a FUNCTION subprogram can be any of the following:

   a. A constant.

   b. A subscripted or nonsubscripted variable or an array name.

   c. An arithmetic or a logical expression.

   d. The name of a FUNCTION or SUBROUTINE subprogram.

   e. An omitted or null argument, indicated by successive commas. References to null arguments from within the called function are undefined.

9. 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.

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

11. A FUNCTION name must be unique and limited to 6 characters.

See Tables 6-2 and 6-3 for supplied FUNCTION subprograms.

The following example shows the use of a FUNCTION subprogram:

```
Calling Program            Called Program


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

GO TO

GO TO, Unconditional

The unconditional GO TO indicates the next statement to be executed. It has the form:

```
     GO TO k
```

where k is the statement number of another statement in the program. When this statement is encountered, the next statement to be executed is the statement having statement number k. This statement can be any executable statement in the program either before or after the GO TO statement subject to the rules for transferring into and out of DO loops. For example:

```
     GO TO 5
```

The program continues execution with statement number 5. Control is transferred unconditionally to statement number 5.

GO TO, Assigned

The assigned GO TO statement indicates which statement is the next to be executed. The assigned GO TO has the form:

GO TO i, $(k_1, k_2, \ldots, k_n)$
or
GO TO i

where

i is an integer switch variable
k_i are statement numbers

The k's are optional. If present, the variable i must have been assigned the value of one of the statement numbers in the parentheses. The next statement to be executed is the one whose statement number in the parentheses has the same value as the variable i. If a statement number has been assigned to i that is not in the list of k's, a compile time diagnostic is generated.

For example:

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

Statement number 17 is executed next.

GO TO, Computed

The computed GO TO indicates the statement that is to be executed next. This is determined by using a computed integer value. It has the following form:

GO TO $(k_1,k_2,\ldots,k_n)$,e

where the k_i are statement labels or switch variables. The expression e is truncated to an integer at the time of execution. The next statement to be executed will be k_i where i is the integral value of the expression e. If i is out of range, a message is outputed and execution is terminated. For example:

```
     J = 3
     GO TO (5,4,17,1),J
```

Statement 17 is executed next.

IF, ARITHMETIC

The arithmetic IF statement causes a change in the execution sequence of statements depending on the value of an arithmetic expression. It has the following form:

IF (e) k_1, k_2, k_3

where e is an arithmetic expression and the k_i are statement numbers, switch variables or are null (not supplied). When k_i is null, the statement referenced is the next executable statement in the program.

The arithmetic IF is a three-way branch. Execution of this statement causes a transfer to one of the statements k_1, k_2, or k_3. The statement identified by k_1, k_2, or k_3 is executed next depending on whether the value of e is less than zero, zero, or greater than zero, respectively. Any two of k_1, k_2, and k_3 are optional, and if null, cause the execution of the program to continue with the next sequential executable statement after the IF statement.

Example:

```
IF  (A(J,K)-B)  10,4,30

IF  (A(J,K)-B) <0 control goes to statement 10
IF  (A(J,K)-B) =0 control goes to statement 4
IF  (A(J,K)-B) >0 control goes to statement 30
```

IF, LOGICAL

The logical IF statement causes conditional execution of a certain statement depending on whether or not a logical expression is true or false. It has the following form:

```
IF(e)s
```

where e is a logical or relational expression and s is any executable statement except a DO statement or another logical IF statement. Upon execution of this statement, the logical or relational expression e is evaluated. If the value of e is true, statement s is executed. If the value of e is false, control is transferred to the next sequential statement.

Example:

```
IF(A.GT.B) GO TO 3
```

If A is arithmetically greater than B, the execution of the user program continues with the statement labeled with 3. Otherwise execution continues with the next sequential executable statement.

If e is true and s is a CALL statement that does not take a nonstandard return, control is transferred to the next sequential statement upon return from the subprogram.

The following examples illustrate several uses of the logical IF.

```
1.   IF (A.AND.B) F = SIN (R)

2.   IF (16.GT.L) GO TO 24

3.   IF (D.LE.Y.OR.X.LE.Y) GO TO (18,20),I

4.   IF (Q) CALL SUB
```

In example 1, if (A.AND.B) is true, compute F and return to the statement following IF.

In example 2, if (16.GT.L), control transfers to statement 24.

In example 3, if (D.LE.Y.OR.X.LE.Y) is true, control transfers to statement 18 or 20 depending upon whether I is 1 or 2.

In example 4, Q must have been previously typed as LOGICAL. If its current value is true, control goes to the subprogram SUB. Return is to the statement following the IF.

If the operator .NE. or .EQ. is contained in a logical IF expression and both operands are not type integer or character, a warning message appears at the end of the source listing. The error message indicates that the equality or non-equality relation between the operands may not be meaningful. This is due to the fact that floating point arithmetic is not exact for certain fractions.

If the relational expression compares two character strings of unequal length, the shorter string is left justified and filled with blanks to equal the length of the longer string before the comparison is made.

IMPLICIT

The IMPLICIT type statement has the following form:

IMPLICIT type*s(h_1,h_2,...),type*s(h_1,h_2,...)

where: each h_i is a letter or pair of letters (separated by a dash) of the alphabet.

Type can be any of the following operators: INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL, or CHARACTER.

*s is optional and designates a length specification for its associated data type. Length specifications are ignored if type is INTEGER, DOUBLE PRECISION, COMPLEX, or LOGICAL. When type is REAL, a length specification of eight or more implies DOUBLE PRECISION; when type is CHARACTER, the length specification is as defined for the CHARACTER statement.

The IMPLICIT statement is used to redefine the implicit typing. All variable and function names beginning with a letter specified in the list or included in the alphabetic interval defined by two letters separated by a dash are typed as specified in the "Type" field. An IMPLICIT statement supersedes previous IMPLICIT statements. The IMPLICIT statement must appear before any use of the variable name being typed. It does not override explicit type statements.

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, X, or Y, is implicitly typed INTEGER. This also applies to the lowercase letters a through f, x, and y.

```
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 also applies to the lowercase letters a through h and o through z.

NOTE: When the IMPLICIT statement immediately follows either a SUBROUTINE or FUNCTION statement, the dummy arguments are affected by the implicit typing. This statement syntax is not recommended.

INTEGER

The INTEGER statement is an explicit type statement with the following form:

INTEGER $a_1*s_1(k_1)/d_1/,a_2*s_2(k_2)/d_2/,...,a_n*s_n(k_n)/d_n/$

where

a_i is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement.

$*s_i$ is an optional size-in-bytes qualification and is ignored.

k_i supplies the dimension information necessary to allocate storage to arrays.

d_i represents initial data value.

The INTEGER statement is more fully described under the Type statement entry in this section.

LOGICAL

The LOGICAL statement is an explicit type statement with the following form:

LOGICAL $a_1*s_1(k_1)/d_1/,\ldots,a_n*s_n(k_n)/d_n/$

where

a_i is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement.

$*s_i$ is an optional size-in-bytes qualification, and is ignored.

k_i supplies the dimension information necessary to allocate storage to arrays.

d_i contains the initial data value.

The LOGICAL statement is more fully described under the Type statement in this section.

NAMELIST

The NAMELIST statement has the following form:

NAMELIST/n_1/k_1/n_1/k_1/.../n_n/k_n

where each n_i is a NAMELIST name and each k_i contains lists of variables and/or array names to be associated, for input/output purposes, with the corresponding NAMELIST names.

The following rules apply to the NAMELIST statement:

1. A NAMELIST name consists of one to eight alphanumeric characters; the first character must be alphabetic. The name must be unique within the first six characters.
2. A NAMELIST name is enclosed in slants. The field of entries belonging to a NAMELIST name ends either with a new NAMELIST name enclosed in slants or with the end of the NAMELIST statement.
3. A variable name or any array name can belong to one or more NAMELIST names. Such variable names can also be of one to eight characters providing they are unique within the first six.
4. A NAMELIST name must not be the same as any other name in the program.
5. A NAMELIST statement defining a NAMELIST name must precede any reference to the name in the program.
6. A dummy argument of a subprogram cannot be used as a variable in a NAMELIST statement.
7. The NAMELIST table can accommodate array variables of no more than three dimensions.

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
```

Additional information on NAMELIST input/output statements is contained in Section V.

PARAMETER

The PARAMETER statement has the following form:

PARAMETER v_1=e_1, v_2=e_2,...,v_n=e_n

where: v_i is a parameter symbol. e_i represents arithmetic expressions involving only constants and previously defined parameter symbols.

The PARAMETER statement is used to define program constants with the result of an expression at compilation time. The value of a parameter symbol cannot be redefined during the execution of a program. A parameter symbol cannot appear where a constant cannot appear and cannot appear in a FORMAT statement.

The appearance of a parameter symbol in some context is interpreted as if its equivalent value had appeared instead.

A parameter symbol v may be of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER depending on the type of its defining expression e. In the following examples, I and J are of type INTEGER, K is REAL, L is LOGICAL, and M is CHARACTER.

```
PARAMETER I=5/2, J=I*3, K = 3.14159, L=.T., M="060171"
```

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 the parameter symbol M is assigned a CHARACTER equivalence.

The significant difference between a parameter symbol and, say, an ordinary integer variable that can be initialized with a DATA statement is in the usage. For example, a parameter variable can be used to supply dimensionality information.

```
      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.
```

In this example, 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 DO statement description 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 causes a temporary halt in the execution of the program until the operator resumes execution. A line is transmitted to the operator console (or user terminal in time sharing mode) consisting of the word "PAUSE", and information derived from the PAUSE statement. When the user transmits a carriage return, execution is continued with the statement following the PAUSE. If the user transmits "STOP", (or a word beginning with the letter S) execution halts. It has the form:

```
     PAUSE
       or
     PAUSE n
```

where n is an integer or character constant or variable. Integer values are limited to five digits.

Examples:

```
     PAUSE
     PAUSE  77777
     PAUSE  I
     PAUSE "TØØ BAD"
```

For PAUSE and PAUSE n, where n is an integer, the message displayed is

```
PAUSE ...... SNUMB snumb - nn
```

The line number field (......) contains the line number of the PAUSE statement or the integer n; snumb is the SNUMB of the job, nn is the activity number. The SNUMB and activity number are omitted for TSS jobs running under FORTRAN or YFORTRAN.

For PAUSE n where n is character information, the message displayed is

```
PAUSE  .....
```

where the ..... field is the character information.

For example:

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

A call to the above subroutine of the form:

```
     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

```
     PRINT, list
```

This form of the PRINT statement is used for list directed formatted output to the standard system output device. For a complete discussion of list directed input/output, see Section V, "List Directed Input/Output statements."

```
     PRINT t, list
          or
     PRINT t
```

The formatted PRINT statement causes information (list) to be transmitted to the standard output device and converted according to the format specified in t. The first character of each record supplied is a control character.

To be classified as a formatted PRINT, t must be a FORMAT statement number, a character scalar, or an array name.

```
     PRINT x
```

The NAMELIST PRINT statement causes the printout of information at the standard output device in accordance with the NAMELIST group x. For a complete description of NAMELIST input/output, see Section V.

To be classified as a NAMELIST PRINT, x must be a NAMELIST name.

PUNCH

```
PUNCH t, list
    or
PUNCH t
```

The formatted PUNCH statement causes information in punchable form to be transmitted to the standard output device, converted according to the format specified in t. (See FORMAT Statement.) To be classified as a formatted PUNCH, t must be a FORMAT statement number, a character scalar, or an array name.

```
PUNCH, list
```

This form of the PUNCH statement is used to transmit list directed formatted output in punchable form to the standard output device. See Section V for a complete description of list directed input/output.

```
PUNCH x
```

This NAMELIST PUNCH statement, where x is a NAMELIST name, causes formatted punchable information to be directed to the standard output device. See Section V for a complete description of NAMELIST input/output.

READ

```
READ, list
```

This form of the READ statement is used for list-directed formatted input from the standard system input device. For a complete discussion of list-directed input/output, see Section V. A read after a write on the same file is illegal, but the sequence of WRITE, REWIND, and then READ is legal.

```
READ t, list
      or
READ t
```

This statement enables the user to read a list referencing format information (t) that describes the type of conversion to be performed. A request is sent to the standard input device. The input is converted according to the format specified in t. The t field can be an integer constant representing a FORMAT statement number, or a character scalar or array name containing the FORMAT information (see Section V).

```
READ x
   or
READ (f,x)
```

This is a NAMELIST input statement where x is a NAMELIST name and f is a file reference. The first statement causes a read request to be sent to the standard input device. Input in NAMELIST input format will be accepted. See Section V for a complete description of NAMELIST input/output.

```
READ (f,t,opt1,opt2) list
```

This statement, formatted file READ, includes a reference to format information (t) and a file reference (f). It can include either or both options (opt1 and opt2) and a list specification. The file reference (f) can be an integer constant, variable, or expression. A file designator of 5 or 41 implies reference to the standard system input device.

The end-of-file transfer (opt1) option is designated as END=S1, where S1 is the statement label that is to receive control when an end-of-file condition is encountered.

The error transfer (opt2) option is designated as ERR=S2, where S2 is the statement label or switch variable that is to receive control when any input/output error is encountered.

The options can appear in any order and S1 and S2 can be statement numbers or switch variables.

*

READ (f,opt1,opt2) list

The unformatted file READ statement is the same as the formatted file READ except that the FORMAT reference is omitted. This statement applies to word-oriented serial access files (binary sequential files).

READ (f'n,opt1,opt2)list

This unformatted file READ is for random binary files. The n is an integer constant, variable, or expression that specifies the sequence number of the logical record to be accessed.

READ (f,x,opt1,opt2)

The NAMELIST file READ statement has a reference to some NAMELIST name (x) and the list is omitted. This statement causes formatted input to be read in accordance with NAMELIST group(x).

REAL

The REAL statement is one of the explicit type statements with the following form:

REAL a_1*s_1(k_1)/d_1/,...,a_n*s_n(k_n)/d_n/

where

a_i	is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by this statement
*s_i	is the size specification. If it is greater than 7, the type is treated as DOUBLE PRECISION
k_i	supplies dimension information necessary to allocate storage to arrays
d_i	is the initial data value

The REAL statement is more fully described under the Type statement in this section.

RETURN

The logical termination of any subprogram is the RETURN statement, which returns control to the calling program. There can be any number of RETURN statements in the program.

A RETURN statement is of the form:

```
RETURN
  or
RETURN i
```

where i is an integer constant or variable that denotes the ith nonstandard return in the argument list, reading from left to right of the CALL statement that invoked this (returning) subroutine. The value of i must be a positive integer no greater than the number of nonstandard returns in that argument list. When i has the value zero, a normal return is taken (the first of the RETURN statements shown above).

REWIND

This statement refers only to sequential files. It causes the specified file to be positioned at its initial point. The file is closed if it is an output file. The statement has the following form:

```
     REWIND f
```

where f is the file reference. f can be an integer constant or variable.

If the file is an output file, an EOF is written before rewinding.

STOP

The STOP statement causes the object-program to halt and control to be returned to the operating system. It has the forms:

```
     STOP n
     STOP
```

where n is an integer or character constant or variable.

The action taken when a STOP statement is executed varies with batch and TSS execution, and the presence of n. STOP n prints on the standard system output device:

```
     STOP AT LINE n
        or
     STOP n
```

the former being displayed when n is an integer, the latter when n is character. STOP with no identification (n) goes unrecorded and the program simply terminates.

SUBROUTINE

The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram. The SUBROUTINE statement has the following form:

SUBROUTINE name (a_1, a_2, ...,a_n)
or
SUBROUTINE name

where

name is the symbolic name of a subprogram and must be unique within the first six characters.

Each a_i (if present) is a nonsubscripted variable or array name, the dummy name of a SUBROUTINE or FUNCTION subprogram, or an * or $ indicating a nonstandard return.

Examples:

```
SUBROUTINE COMP (X,Y,*,$,P)
SUBROUTINE QUADEQ (B,A,C,ROOT1, ROOT2)
SUBROUTINE OUTPUT
```

1. The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram.

2. The SUBROUTINE subprogram can use one or more of its arguments to return output. The arguments so used must appear in some definitional content within the subprogram other than a DATA statement (that is not allowed). See the EQUIVALENCE statement rule 5 for a definition of the contexts.

3. 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. The actual arguments must correspond in number, order, size and type with the dummy arguments.

4. When a dummy argument is an array name, a statement containing dimension information must appear in the SUBROUTINE subprogram; the corresponding actual argument in the CALL statement must be a dimensioned array name.

5. No argument in a SUBROUTINE statement can be included in COMMON, EQUIVALENCE, NAMELIST, or DATA statements in the subprogram.

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

7. The SUBROUTINE subprogram can contain any FORTRAN statements except FUNCTION, BLOCK DATA, or another SUBROUTINE statement.

8. The character * or $ appearing in an argument position denotes a nonstandard or alternate exit from the subroutine.

TYPE

The explicit type statements are of the form:

type *b a_1*s_1(k_1)/d_1/,a_2*s_2(k_2)/d_2/,...,a_n*s_n(k_n)/d_n/

where:

type is one of the following: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER.

b is an optional integer constant size specification and applies to all variables in the statement unless otherwise specified by s_i. For example:

```
CHARACTER *10 C1,C2,C3*12,C4
```

C1, C2, and C4 have a length of ten characters. C3 has a length of 12 characters.

a_i is a variable, array, or FUNCTION subprogram name whose characteristics are being qualified by the type statement.

*s_i is an optional, positive integer constant, size specification. This field is ignored for all types except:

1. For REAL type, a size specification, *s, greater than seven is treated as DOUBLE PRECISION.

2. For CHARACTER type, the size field, *s, is interpreted as the maximum number of characters that may be contained by the CHARACTER element being defined. When this field is omitted, the size is assumed to be six for BCD programs and eight for ASCII. S cannot be greater than 500 characters for ASCII or 511 characters for BCD. When s is less than or equal to six, then s represents the number of characters that are stored in each word.

   An adjustable size specification for s is permitted within a subprogram when both the character variable and its size parameter(s) are included as formal parameters. Additional information on adjustable size specifications is contained under the CHARACTER statement in this section.

(k_i), if present, consists of from one to seven integer constants, parameter symbols, or (for subprograms only) INTEGER variables separated by commas. This field supplies dimension information (information necessary to allocate storage to arrays). If this information does not appear in the type statement, it must appear elsewhere in the program (in a DIMENSION or COMMON statement).

NOTE: When a type statement immediately follows a SUBROUTINE or FUNCTION statement, the dummy arguments are affected by the implied typing.

$/d_i/$ represents initial data value. The form for d_i is as specified for the DATA statement.

Thus, meaningful permutations of the above permit:

INTEGER $a_1(k_1)/d_1/,a_2(k_2)/d_2/,\ldots,a_n(k_n)/d_n/$

REAL *b $a_1*s_1(k_1)/d_1/,a_2*s_2(k_2)/d_2/,\ldots,a_n*s_n(k_n)/d_n/$

DOUBLE PRECISION $a_1(k_1)/d_1/,a_2(k_2)/d_2/,\ldots,a_n(k_n)/d_n/$

COMPLEX $a_1(k_1)/d_1/,a_2(k_2)/d_2/,\ldots,a_n(k_n)/d_n/$

LOGICAL $a_1(k_1)/d_1/,a_2(k_2)/d_2/,\ldots,a_n(k_n)/d_n/$

CHARACTER *b $a_1*s_1(k_1)/d_1/,a_2*s_2(k_2)/d_2/,\ldots,a_n*s_n(k_n)/d_n/$

WRITE

WRITE (f,t,opt2) list

The formatted file WRITE statement must include a file reference (f) and a FORMAT reference (t). It can include the option opt2 and a list reference.

A write after a read on the same file is illegal but the sequence of WRITE, REWIND, and then READ is legal.

The file reference (f) can be an integer constant, variable, or expression. A designation of 6 or 42 implies the system standard output printing device. A designation of 43 implies the system standard output punching device.

The FORMAT reference (t) can be an integer constant representing the statement label of a FORMAT statement, or a character scalar or array element containing the FORMAT information (see Section V).

The error transfer (opt2) option is designated as ERR=S2, where S2 is the statement label or switch variable that is to receive control when an error condition is encountered.

WRITE (f,opt2) list

The unformatted file WRITE statement omits the format reference. It applies to the output of word oriented serial access files (binary sequential files). The f, opt2, and list fields are as specified for the formatted file WRITE.

WRITE (f'n,opt2) list

The random binary file WRITE statement contains a field (n) that specifies the sequence number of the logical record to be written. n may be a constant, variable, or expression and must be integer. The f, opt2, and list fields are as specified for the formatted file WRITE.

WRITE (f,x,opt2)

The namelist file WRITE statement includes a reference to the NAMELIST name (x). This statement causes character oriented records to be written on the indicated device. The f and opt2 fields are as specified for the formatted file WRITE; no list field is included in the namelist file WRITE. See Section V for a complete description of NAMELIST input/output.

SECTION V

INPUT AND OUTPUT

GENERAL DESCRIPTION

FORTRAN input/output statements specify the transmission of information between internal storage and input/output devices.

The ENCODE and DECODE statements, while not actually input/output statements, are of the same general form as formatted file WRITE and READ statements, respectively. They differ in that the file reference field of the READ/WRITE statements provides a storage reference for packed character information with DECODE/ENCODE. The information in this section contains general information for all of these statements.

The following notation is used in the description of input/output statements.

```
list = indication of an input/output list
   f = file reference (file code)
   t = FORMAT reference
   x = reference to a NAMELIST name
   a = reference to an internal storage buffer
       for packed character data
 opt = optional transfer condition
```

Each input/output statement can contain an implicit (NAMELIST) or explicit list of variable names, arrays, and array elements. The named elements are assigned values on input (or DECODE) and have their values transferred on output. With ENCODE, the values are converted to character information and stored in (a).

The list of a WRITE, PRINT, PUNCH, or ENCODE statement can also include constants and expressions of all types.

The file reference (f) can consist of an integer constant, variable, or expression that identifies the input/output unit. The value of the integer will be a two-digit file code, the value of which must be in the range $01 \le f \le 43$. A file is associated with a device by using the $ file and $ FFILE control cards or by using the 'fe' file descriptors of the RUN command described in Section III.

The FORMAT reference (t) can be an integer constant representing the statement number of a FORMAT statement, or the name of a character scalar or array element. If a statement number 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 (x) in the format reference position of the READ, WRITE, and PRINT statements. The NAMELIST statement(s) defining x and its associated list must appear before any input/output statements referencing x.

The internal storage buffer, a, applies only to the ENCODE and DECODE statements. Array names of any type or character variables (scalar or array element) can be used; however, the latter is preferred.

There are two optional transfer conditions: end-of-file and error. These 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 number or switch variable name can follow the equal sign (=). The order of the transfer conditions is not important. Conditions that can give rise to an error return include transmission errors or any of the error conditions described in the File and Record Control manual under "Error Procedures - User Supplied Routine".

The information 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. An unformatted record consists of a string of words.

List directed formatted input/output can be specified by a FORMAT statement of the form FORMAT(V) or can be implied by the form and content of the input/output statement.

Input/output statements are grouped as follows:

1. System device input/output statements.

   a. Formatted Input - Permits entering information from the standard input device with reference to a FORMAT statement.

   b. Formatted Output - Permits transfer of information to the standard output device with reference to a FORMAT statement.

   c. Formatted Punch - Permits transfer of information in punchable form to the standard output device with reference to a FORMAT statement.

   d. List Directed Input - Permits entering information from the standard input device without reference to a FORMAT statement.

   e. List Directed Output - Permits transfer of information to the standard output device without reference to a FORMAT statement.

   f. List Directed Punch - Permits transfer of punchable information to the standard output device without reference to a FORMAT statement.

   g. NAMELIST Input - Permits entering information from the standard input device with reference to a NAMELIST name.

   h. NAMELIST output - Permits transfer of information to the standard output device with reference to a NAMELIST name.

2. File input/output statements.

   a. Formatted Read/Write Statements - These statements include a FORMAT reference, the file reference, possibly an end-of-file option, an error return option, and/or 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.

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

3. 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.

4. 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 in "Data Input Referring to a FORMAT Statement" 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 effected 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 in "Data Input Referring to a NAMELIST Statement" in this section.

File Designation

In the source program, files can be designated by any integer expression, the value of which must be in the range of 1 to 43. The equation 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 (6,100) references file code 06 at run time.

Since the file designator 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 (these definitions can be overridden by the programmer): 05 designates the standard input file (file code I*); 06 refers to the standard output file (file code P*); and 41, 42, and 43 are referenced by the READ, PRINT, and PUNCH statements: 41 references the standard input file (I*); 42 the standard output file (P*) with printer destination; and 43 the standard output file (P*) with punch destination.

The default assignment of devices for the standard system file designators 05, 06, 41, 42, and 43 is as described above when the execution is done in a batch environment. In time sharing the default device of all of these file codes is the time sharing terminal. Allocation of actual files (versus the terminal) for one or more file designators is accomplished via the fe file list of the RUN command. In this list, the user supplies a descriptor for each file to be used by the object program. The names of all such files must be a 2-digit file code (nn) in the interval 01 ≤ nn ≤ 43. Unless the file has been created with a name which is the 2-digit file code, it will be necessary to specify the file code as an alternate name. Suppose, for example, a program has an input statement of the form: READ (5,100) I,J,K. Normal time sharing execution of this program will access the terminal for input values for I, J, and K. However, if the program is initiated with a RUN command such as

```
RUN PROGRAM # INPUT "05"
```

then the user's catalog is searched for the file named INPUT, that file is accessed, and the AFT name for the file will be 05. Execution of the above READ statement thus reads the file INPUT for its values of I, J, and K.

Conversely, given a statement of the form READ (10,100)... where typical operations would be on a file, the user can specify terminal input by omitting the catalog file descriptor. e.g.,

```
RUN PROGRAM # "10"
```

If any given file descriptor consists only of an unquoted 2-digit logical file code, a temporary file is created for the user 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.

For example:

```
RUN PROGRAM #10
```

If no file exists in the user's catalog of the name 10, a temporary file is created with that name, and I/O directed to file designator 10 is routed to the temporary file.

More detail on the fe list and file allocation for time sharing is given in Section III in the discussion of the RUN command.

List Specifications

When arrays or 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. A list item can be:

1. An expression (output only)
2. An implied DO
3. An array name
4. A scalar
5. A constant (output only)
6. An array element

An input/output list is processed from left to right. Parenthesized sublists are permitted only with implied DO's; redundant parentheses result in a fatal diagnostic.

Examples: A, B, C*D**E, 1.2, SQRT (14.6), F(K,K)

Consider 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)
```

This list implies that the information in the external input/ouptut 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 execution of an input/output implied DO list is exactly that of a DO loop, as though each left parenthesis (except expression and subscripting parenthesis) were a DO, with indexing given immediately before the matching right parentheses, 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 that 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 file.

Consider the following example:

```
          CHARACTER A*1 (50)
          READ (5,100) (A(I),I=1,50)
     100  FORMAT (50A1)
```

This 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.

By specifying an array name in the list of an input/output statement or a NAMELIST, an entire array can be designated for transmission between storage and an input/output device. Only the name of the array need be given and the indexing information can be omitted. For example:

```
     DIMENSION A(5,5)
          .
          .
     READ,A
```

In the above example, 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 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. The type of each variable in the list determines the conversion to be used.

```
     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
```

In all cases where a format reference (t) is supplied, the format must be of the form FORMAT (v). The t can be a FORMAT statement number, a character scalar, or an array name. The table of implied format conversions used for list directed formatted input/output is as follows:

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 (terminal input line, for example) is terminated and the list is not satisfied, another record (line) 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 considered 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 bracketted 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 the following table:

TYPE	VALUE
Integer	0
Real	0.0
Double Precision	0.D0
Complex	(0,0)
Logical	F
Character	all blanks

6. 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 models 33 and 35 teletypewriters, control, shift, L) as the only character of a line (other than the terminal carriage return).

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 some ASCII line and the next item in the list is an integer (implied I16 format), a new line is started.

Namelist Input/Output Statements

The NAMELIST statement and various forms of the NAMELIST input and output statements provide for the input and output of groups of variables and arrays by referring to a single name. NAMELIST names must conform with the same naming rules as normal variables and arrays except there is no type associated with the name and the name must be unique within six characters. A NAMELIST name must not be the same as any other variable, procedure, or array name in the subprogram defining it.

Each list that is mentioned in the NAMELIST statement is given a NAMELIST name. Therefore, only the NAMELIST name is needed in an input/output statement to refer to that list.

The NAMELIST statement has the general form:

NAMELIST $/n_1/k_1/n_2/k_2/..../n_n/k_n/$

where each n_i is a NAMELIST name and each k_i is a list of variable and/or array names to be associated, for input/output purposes, with the corresponding NAMELIST names. The NAMELIST statement is fully described in Section IV.

NAMELIST Input

This statement has the following forms:

```
READ (f,x,opt1,opt2)
READ x
```

where f is a file reference, and x is a NAMELIST name. The first form causes a read request to be sent to file f; the second issues a read request to the standard input device.

NAMELIST Output

This statement has the following forms:

```
WRITE(f,x,opt2)
PRINT x
PUNCH x
```

where f is a file referenced and x is a NAMELIST name. This statement causes printout of information on file f in the first form, or in the second on the standard output device, in accordance with the contents of the NAMELIST group x. The third form also directs output to the standard output device but in punchable format.

Data Input Referring to a NAMELIST Statement

When a READ statement refers to a NAMELIST name, the designated input device is readied and input of data is begun. The first input data record is searched for a $ immediately followed by the NAMELIST name, immediately 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 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.

Any combination of four types of data items, described in the following text, can be used in a data record. Empty fields (detected as one of the pairs (=,), (b̸,), or (,,)) cause an invalid word to be stored. The data items must be separated by commas. 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 if the record is card image (media code 2); otherwise, data can appear anywhere in the record. 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, thus deleting it from the input data file.

The form that data items can take is:

1. 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. Subscripts must be integer constants.

2. 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 be equal to the number of elements in the array.

3. 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.

4. 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 take any of the following forms:

1. Integers, e.g., 1,2,3
2. Real numbers, e.g., 1.,2.,3.3
3. Double precision numbers, e.g., -263D15
4. Complex numbers, that must be written in the usual form, (C1,C2), where C1 and C2 are real numbers.
5. Logical constants, that must be written as T or .TRUE., and F or .FALSE.
6. Character data written nH... or '...' 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.

Any selected set of variable or array names belonging to the NAMELIST name, 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.

```
                    1  456
                    ______________________________
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$
```

If the data cards are to be processed by System Input, the $ should not appear in column one. This results in an ambiguity with respect to control cards.

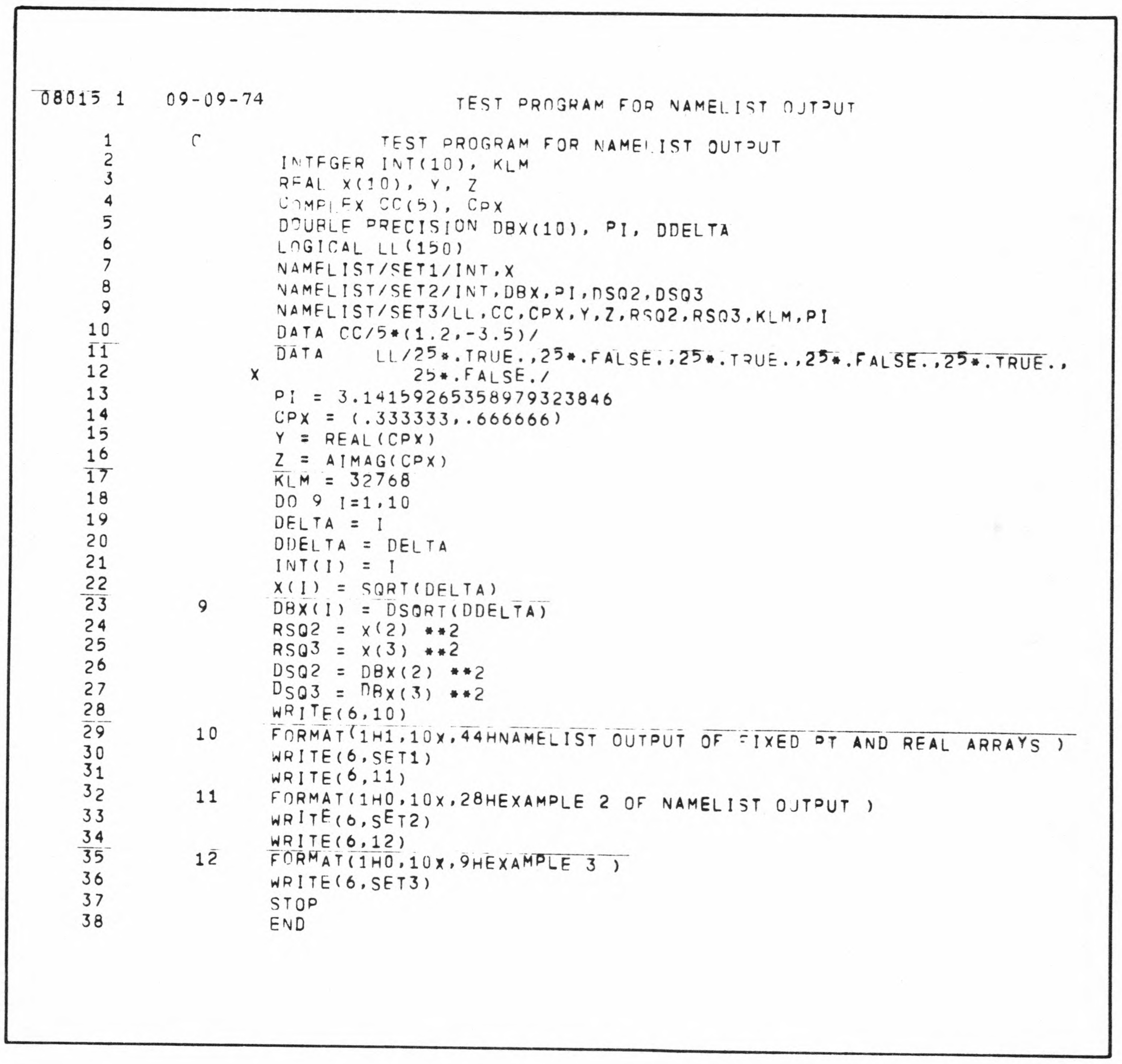
If this data is input to be used with the NAMELIST statement previously illustrated (in Section IV, NAMELIST statement) and with a READ statement, the following actions take place. The input file designated in the READ statement is prepared and the next record is read. The record is searched for a $ immediately followed by the NAMELIST name, NAM1. Since 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 and 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), and 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, the entire array is filled with the suceeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively, and the real constant 4.3 is converted to an integer and placed in L(3), L(4),..., L(10). The $ signals termination of the input for the READ operation.

Data Output Referring to a NAMELIST Statement

When data is output via NAMELIST, e.g., WRITE(6,NAM1), all variables associated with LIST, as specified in the NAMELIST statement, will be output. The output values are labeled with an appropriate variable name.

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 do not include comma separators and therefore cannot be processed by NAMELIST input. Figures 5-1 and 5-2 contain an example program and sample output from that program in the latter format.



```
08015 1    09-09-74                 TEST PROGRAM FOR NAMELIST OUTPUT

            C            TEST PROGRAM FOR NAMELIST OUTPUT
                  INTEGER INT(10), KLM
                  REAL X(10), Y, Z
                  COMPLEX CC(5), CPX
                  DOUBLE PRECISION DBX(10), PI, DDELTA
                  LOGICAL LL(150)
                  NAMELIST/SET1/INT,X
                  NAMELIST/SET2/INT,DBX,PI,DSQ2,DSQ3
                  NAMELIST/SET3/LL,CC,CPX,Y,Z,RSQ2,RSQ3,KLM,PI
                  DATA CC/5*(1.2,-3.5)/
                  DATA     LL/25*.TRUE.,25*.FALSE.,25*.TRUE.,25*.FALSE.,25*.TRUE.,
                 X         25*.FALSE./
                  PI = 3.14159265358979323846
                  CPX = (.333333,.666666)
                  Y = REAL(CPX)
                  Z = AIMAG(CPX)
                  KLM = 32768
                  DO 9 I=1,10
                  DELTA = I
                  DDELTA = DELTA
                  INT(I) = I
                  X(I) = SQRT(DELTA)
             9    DBX(I) = DSQRT(DDELTA)
                  RSQ2 = X(2) **2
                  RSQ3 = X(3) **2
                  DSQ2 = DBX(2) **2
                  DSQ3 = DBX(3) **2
                  WRITE(6,10)
             10   FORMAT(1H1,10X,44HNAMELIST OUTPUT OF FIXED PT AND REAL ARRAYS )
                  WRITE(6,SET1)
                  WRITE(6,11)
             11   FORMAT(1H0,10X,28HEXAMPLE 2 OF NAMELIST OUTPUT )
                  WRITE(6,SET2)
                  WRITE(6,12)
             12   FORMAT(1H0,10X,9HEXAMPLE 3 )
                  WRITE(6,SET3)
                  STOP
                  END
```

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   0.100000000000000000D 01    0.141421356237309505D 01    0.173205080756887729D 01
      4   0.200000000000000000D 01    0.223606797749978970D 01    0.244948974278317810D 01
      7   0.264575131106459059D 01    0.282842712474619010D 01    0.300000000000000000D 01
     10   0.316227766016837933D 01
PI        0.314159265358979324D 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        0.314159265358979324D 01
END     NAMELIST     SET3
```

Figure 5-2. NAMELIST Output of Fixed Point And Real Arrays

Formatted Input/Output Statements

These statements include a FORMAT reference, can include a file reference, either or both options 1 and 2, and a list specification. These statements pertain to character oriented sequential files. The formatted file statements have the following forms:

```
READ t, list
PRINT t, list
PUNCH t, list
READ (f,t,opt1,opt2)list
WRITE (f,t,opt2)list
```

where: t is the format reference.

f is the file designator and can be any integer expression.

A file designator of 5 or 41 for input, or 6, 42, or 43 for output implies a reference to the standard input/output devices.

Unformatted Sequential File Input/Output Statements

The unformatted sequential file input/output statements have the following forms:

```
READ (f,opt1,opt2) list
WRITE (f,opt2) list
```

The format reference is omitted and opt1, opt2, and list are optional. These statements apply to word oriented serial access files (binary sequential files).

Unformatted Random File Input/Output Statements

The forms for random binary file references are as follows:

```
READ (f'n,opt2)list
WRITE (f'n,opt2)list
```

where: n is an integer constant, variable, or expression that specifies the sequence number of the logical record to be accessed.

The major difference between the unformatted sequential and unformatted random file operations 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.

Linked files in time sharing can be accessed in a random mode using the ACCESS subsystem. For example, at the build mode level:

```
*ACCESS AF,/X"02",MODE/RANDOM/,R
*RUN#02
```

This is particularly useful when reading a FORTRAN created, standard system format, unformatted sequential file using random READ statements. Each record in the sequential file must be the same length.

Linked files in batch mode can be accessed in a random mode using a CALL ATTACH and specifying random mode.

Unformatted random files created by FORTRAN are normally recorded in standard system format (see <u>File and Record Control</u> reference manual).

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
        or
      CALL     RANSIZ(U,N,1)
```

U and N are the file unit number and logical record size parameters.

It is a requirement that FORTRAN random files have a constant record size. Further, 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 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. For example:

```
CALL RANSIZ(08,50)
```

This statement specifies that file code 08 has a constant record size of 50 and is in standard system format.

<u>File Properties</u>

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 replacment.
Record Sizes	- Random files have records, all of the same length.

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 IV:

REWIND
BACKSPACE
ENDFILE

INTERNAL DATA CONVERSION

The ENCODE and DECODE statements are similar to the formatted WRITE and READ statements respectively except 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)list
DECODE (a,t)list
```

where a is the internal buffer and t is a format designator.

The buffer area is designated by the first operand within the parentheses. It can be given as:

1. A character scalar
2. A character array element
3. An array

When the buffer area is designated as a scalar, it can be considered as 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 can be considered as 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 can be accomplished by letting the list and format specification 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 could 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. The following example demonstrates this feature.

```
          CHARACTER TEXT*20
          TEXT = "WOW IS THE TIME FOR "
          ENCODE (TEXT,10) "NOW"
     10   FORMAT (A3)
     20   PRINT, TEXT, "ALL GOOD MEN"
          STOP;END
```

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). For 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
```

CONSTRUCTION OF FORMATS WITH ENCODE

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 character set. The above subroutine will run in ASCII or BCD mode, without change.

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 is not printed but 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 effected. In any event, it is deleted when the line is printed. This control affects printers, terminals, and displays. When FORMAT (V) is used, either explicitly or implicitly, a first 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 question mark character or an 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.

Field Separators

The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or group of field descriptors are separated by a field separator.

The slash is used to separate field descriptors and to specify a demarcation of formatted records. The length of the strings for a given external medium are limited by the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not itself cause the introduction or inception of processing of the next record.

Repeat Specification

It is possible to repeat a field descriptor (except quoted strings, tabulation control, nH and nX) by writing a repetition number in front of it. Thus the field specification 3E12.4 is the same as writing E12.4, E12.4, E12.4.

It is also possible to repeat a group of field descriptors by enclosing the group in parentheses and preceding the left parenthesis with the repeat count. If no count is specified, a repeat count of one is assumed. For example, if four fields on a card are alternately described as F10.6 and E10.2, this can be written as 2(F10.6, E10.2). One additional level of grouping is permitted, using the same rules for representation. If, for example, the fields on a card could be described by (I3, F8.4, E8.2, F8.4, E8.2, I3, F8.4 E8.2, F8.4, E8.2, A10) then a more compact description would be (2(I3,2(F8.4,E8.2)),A10).

Scale Factors

To permit more general use of D-, E-, F-, and G-descriptors, a 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 desired to use it internally in the form .xxxxxx, then the FORMAT specification to effect this change is 2PF7.4. For output data, scale factors can be used with D-, E-, F-, and G-conversion.

For example, the statement FORMAT (I2,3F11.3) might output the following printed line:

```
27ȣȣȣȣ-93.209ȣȣȣȣȣ-0.008ȣȣȣȣȣȣ0.554
```

But the statement FORMAT (I2,1P3F11.3) used with the same data would output the following line:

```
27ȣȣȣ-932.094ȣȣȣȣȣ-0.076ȣȣȣȣȣȣ5.536
```

whereas, the statement FORMAT (I2,-1P3F11.3) would output the following line:

```
27ȣȣȣȣȣ-9.321ȣȣȣȣȣ-0.001ȣȣȣȣȣȣ0.055
```

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, output of numbers, whose absolute value is 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 a list of an input or output statement is used to transmit more than one record (card or line) and with different formats, a slash (/) is used to separate the format specifications for different lines. For example: if two cards 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 of a deck has an integer and a real number and all 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; for 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 following example illustrates the various levels of parentheses.

```
FORMAT (3E10.3,(I2,2(F12.4,F10.3)),D28.17)
       0       1   2            21       0
```

The parentheses labeled 0 are zero level parentheses; those labeled 1 are first level parentheses; and those labeled 2 are second level parentheses. 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; that is, the parenthesis preceding I2.

As these examples show, both the slash and the final right parenthesis of the FORMAT statement indicate a termination of a record.

Blank lines can be introduced into a multiline FORMAT statement by inserting consecutive slashes. When n consecutive slashes appear at the end of the FORMAT, they are treated as follows: for input, n records are skipped; for output, n-1 blank lines are written. When n consecutive slashes appear in the middle of the FORMAT, n-1 records are skipped for both input and output.

Carriage Control

The WRITE (f,t), PRINT, and PRINT t, statements prepare formatted 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 effected. The interpretation of control characters is discussed under "Output Device Control" in this section. This control is usually obtained by beginning a FORMAT specification with 1H followed by the desired control character.

If carriage control information is not desired, see $ FFILE/NOSLEW in the Control Cards or General Loader manuals.

Data Input Referring to a FORMAT Statement

These specifications must be followed when data input to the object program is under format control:

1. The data must correspond in order, type, and field with the field specifications in the FORMAT statement; or the field can be shortened by using commas as delimiters. For example: for a format specification of 3I6, an input data card containing 1,bbbbb2bb3, is accepted. The values 1, 2, and 3 are input. Note that the second field is a full six characters wide and no comma appears; however, commas terminate the first and third fields. When using terminal input, the field can be shortened by using a carriage return as a delimiter.

2. Plus signs can be omitted or indicated by a +. Minus signs must be indicated.

3. Blanks in numeric fields are regarded as zeros.

4. Numbers for E- and F-conversion can contain any number of digits, but only the high-order eight digits of precision are retained. For D-conversion, the high-order 18 digits of precision are retained. In both cases, the number is rounded to eight or 18 digits of accuracy, as applicable.

5. Numeric data must be right-justified in the field.

To permit greater ease in input preparation, certain relaxations in input data format are permitted.

1. Numbers for D- and E-conversion need not have four columns allotted to the exponent field. The start of the exponent field must be marked by a D or an E or, if that is omitted, by a plus or minus sign (not a blank). For example, E2, E+2, +2, +02, and D+02 are all permissible exponent fields.

2. Numbers for D-, E-, and F-conversion need not contain a decimal point; the format specification suffices. 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:

1. D, E, F, G, I, and O represent the type of conversion.

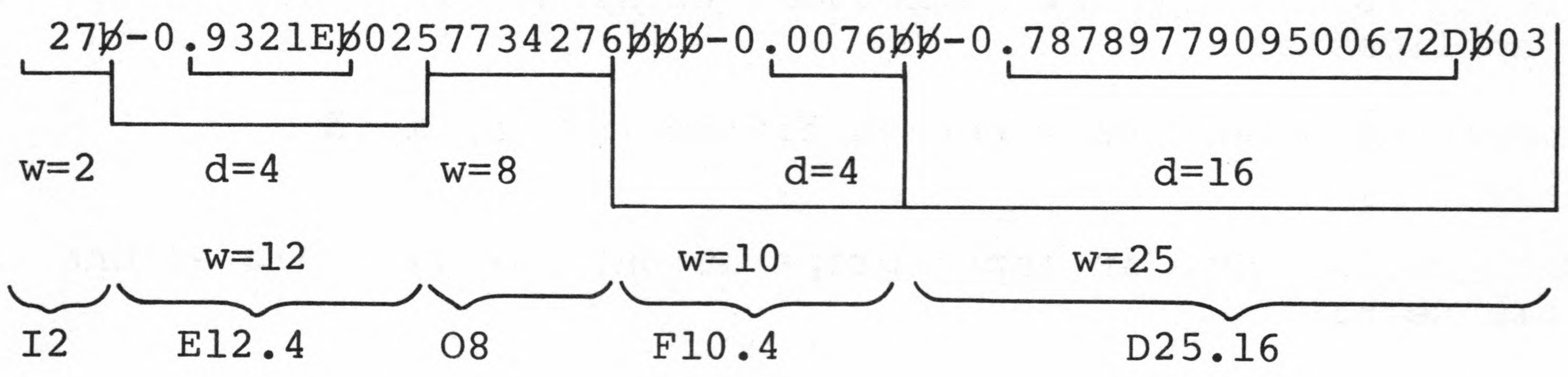
2. The w is an unsigned integer constant representing the field width for converted data; this field width can be greater than required to provide spacing between numbers.

3. The 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 F-conversion, if d is specified $\geq$ 9, it is truncated at 8. For E-conversion and D-conversion, if d is specified $\geq$ 19, it is truncated at 18 and right-justified in the field.

4. Each P is optional and represents a scale factor designator.

5. Each r is an optional nonzero integer constant indicating the number of occurrences of the numeric field descriptor that follows.

   For example, the statement FORMAT (I2,E12.4,O8,F10.4,D25.16) might cause the following line to be printed.



27b̸-0.9321Eb̸0257734276b̸b̸b̸-0.0076b̸b̸-0.7878977909500672Db̸03
w=2
d=4
w=8
d=4
d=16
w=12
w=10
w=25
I2
E12.4
O8
F10.4
D25.16

   where b̸ indicates a blank space.

The following notes apply to D-, E-, F-, G-, I-, and O-conversions:

1. No format specification should be given that provides for more characters than are permitted for a particular input/output record. Thus a format for a record to be printed should not provide for more characters (including blanks) than the capabilities of the relevant device.

2. Information transmitted with O-conversion can have real or integer names; information transmitted with G-conversion can have real or complex names; information transmitted with E-, and F-conversions must have real or complex names; information transmitted with I-conversion must have integer names; information transmitted with D-conversion must have double-precision 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 for the F-conversion except for scale processing.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be 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
. .	.
. .	.
. .	.
$10^{d-2} \leq N \leq 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leq N \leq 10^{d}$	F(w-4).0,4X
Otherwise	nPEw.d

Note that 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 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. For example, the number .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 an output number that is converted by D-, E-, F-, G-, or I-conversions requires more spaces than are allowed by the field width w, the field is filled with asterisks, unless subroutine NASTRK is invoked. (See 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 of 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 octal constant +377777777777 (noise word).

9. Specifications for successive fields are separated by commas and/or slashes. (See "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 supplies the real part. The second supplies the imaginary part.

The following is an example of a FORMAT statement that transmits an array of six complex numbers.

```
FORMAT (2E10.2, E8.3, E9.4, E10.2, F8.4, 3(E10.2, F8.2))
```

Alphanumeric Fields

Alphanumeric information can be transmitted in two ways. Both ways result in the storing of BCD or ASCII characters (as determined by an option in the $ FORTY or $ FORTRAN card or the YFORTRAN or FORTRAN Time Sharing System RUN commands).

1. The specifications rAw and rRw cause character data to be read into or written from a variable.

2. The specification nH, enclosing the string in quotation marks, or enclosing the string in apostrophes introduces alphanumeric information into a FORMAT statement.

The basic difference is that A and R conversions are given a variable name that can be referenced for processing or modification. The character constant notations do not "use" a list item.

The A and R format specifiers are used for the transfer of character variable data to and from input/output buffers.

If the field width (w) specified for A or R input is equal to or greater than the maximum length (as described in the CHARACTER type statement), the rightmost s characters are taken from the external field. The I/O pointer is advanced in accordance with the field width of the format specifier. If the field width is less than the maximum length described in the CHARACTER type statement, the w characters are taken from the external field. With A conversion, the data appears left adjusted with s-w trailing blanks in the internal representation. For R conversion, the internal representation is right justified with s-w leading zeros.

If the field width (w) specified for A or R output is greater than the maximum length described in the CHARACTER type statement, s characters are transmitted to the external field preceded by w-s blanks. If the field width is less than or equal to the maximum length as described in the CHARACTER type statement, the external 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 is used.

The R code is equivalent to A except that the characters are right-justified with leading zeros in the internal representation.

When the variable associated with an A or R format specifier is not of type CHARACTER then the variable is treated as a character variable with a size of one word of storage (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.

1. 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.

2. On output, a value of .TRUE. or .FALSE. in storage causes w minus 1 blanks, followed by a T or an F, respectively, to be written out.

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 character 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 purposes is Tt. The position in a FORTRAN record where the transfer of data is to begin is t. The t is an unsigned integer constant. Using this format code permits input or output to begin at any specified position. Tabbing can proceed backward as well as forward.

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; for example "(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 together 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)
```

SECTION VI

SUBROUTINES, FUNCTIONS, AND SUBPROGRAM 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, built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms. The major differences among the four classes of subroutines are as follows:

1. The first three classes can be grouped as functions. Arithmetic statement functions and built-in functions differ from the SUBROUTINE and FUNCTION subprograms in the following respects:

   a. A function has a single value in an expression.

   b. A function is referred to by an arithmetic expression containing its name; a SUBROUTINE subprogram is referred to by a CALL statement.

2. The arithmetic statement functions and built-in functions are open subroutines. An open subroutine is a subroutine that is incorporated into the object program each time it is referred to in the source program. SUBROUTINE and FUNCTION subprograms are closed; that is, they appear only once in the object program.

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:

1. A subroutine name consists of one to eight alphanumeric characters, the first of which must be alphabetic.

2. The type of the function, which determines the type of the result, is defined as follows:

   a. The type of an external function can be indicated by the name of the function or by placing the name in a Type statement.

b. 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 need not be typed elsewhere.

c. The type of a built-in function is indicated within the FORTRAN compiler and need not appear in a Type statement.

d. 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 dependent only on the type of the variables returned by that subroutine.

ARITHMETIC STATEMENT FUNCTIONS

An arithmetic function is defined internal to the program unit in which it is referenced. It is defined by a single statement similar in form to the arithmetic assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit. The name of a statement function must not appear in EXTERNAL, COMMON, EQUIVALENCE, NAMELIST, or ABNORMAL statements as a scalar name, or as an array name in the same program unit.

Defining Arithmetic Statement Functions

An arithmetic statement function definition has the form:

$f(a_1,a_2,\ldots,a_n) = e$

where f is the function name, the a_i are distinct symbolic names (called dummy arguments of the function), and e is an expression. Since the a_i are dummy arguments, their names, that serve only to indicate number, and order of arguments, can be the same as actual variable names appearing elsewhere in the program unit. The following is a list of exceptions, names of other program symbols that cannot appear as dummy names in the list if they have previously been defined as:

EXTERNAL names
ABNORMAL names
PARAMETER names
NAMELIST names
SUBROUTINE, FUNCTION or ENTRY names
Arithmetic statement function names

The expression, e , can contain:

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 e as a reference of the form:

```
a (list)
```

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.

By way of illustration, consider the following:

```
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)
```

In this example, 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 or an array element. For 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 as a primary 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 providing the corresponding dummy 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 resultant value is then made available to the expression that contained the function reference.

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.

Arithmetic Statement Function Example

A function can be defined to compute one root of the quadratic equation, $ax^2+bx+c=0$, given values a, b, and c as follows:

```
     ROOT (A,B,C)=(-B+SQRT(B**2-4*A*C))/(2*A)
```

This is the definition of the function. This definition can be used by supplying values for a, b, and c. An example of the use of the function using 16.9 for a, 20.5 for b, and T+30 for c follows:

```
     ANS = ROOT(16.9,20.5,T+30)
```

SUPPLIED INTRINSIC FUNCTIONS

The functions listed in Table 6-1 are the intrinsic or built-in functions supplied with FORTRAN. The intrinsic functions require only a few machine instructions and are inserted each time the function is used. To use these functions, it is necessary to only 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.

All functions in Table 6-1, except FLD, AND, OR, XOR, BOOL, and COMPL, are the standard FORTRAN intrinsic functions and their use is not described in this document. The use of FLD, AND, OR, XOR, BOOL, and COMPL are described in this section.

Table 6-1. Supplied Intrinsic Functions

Intrinsic Function	Definition	No. of Arg.	Call Name	Type of: Arg.	Type of: Function
Absolute Value	$\lvert a \rvert$	1	ABS IABS DABS[2] CABS[2]	Real Integer Double Complex	Real Integer Double Real
Truncation	Sign of a times largest integer $\leq \lvert a \rvert$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering[1]	a_1 (mod a_2)	2	AMOD MOD DMOD[2]	Real Integer Double	Real Integer Double
Choosing Largest Value	Max $(a_1, a_2, \ldots)$	≥ 2	AMAX0 AMAX1 MAX0 MAX1 DMAX1 MAX	Integer Real Integer Real Double I,R,D	Real Real Integer Integer Double I,R,D
Choosing Smallest Value	Min $(a_1, a_2, \ldots)$	≥ 2	AMIN0 AMIN1 MIN0 MIN1 DMIN1 MIN	Integer Real Integer Real Double I,R,D	Real Real Integer Integer Double I,R,D
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\lvert a_1 \rvert$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	a_1-Min (a_1, a_2)	2	DIM IDIM DDIM	Real Integer Double	Real Integer Double
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double

Table 6-1. (Cont). Supplied Intrinsic Functions

Intrinsic Function	Definition	No. of Arg.	Call Name	Type of: Arg.	Type of: Function
Express Two Real Arguments in Complex Form	$a_1+a_2 \sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex
Logical "and"	$a_1*a_2*\ldots$	≥ 2	AND	REAL, INTEGER, or TYPELESS	Typeless
Logical "or"	$a_1+a_2+\ldots$	≥ 2	OR	REAL, INTEGER, or TYPELESS	Typeless
Logical "exclusive or"	$a_1 \oplus a_2 \oplus \ldots$	≥ 2	XOR	REAL, INTEGER, or TYPELESS	Typeless
Ignore Type		1	BOOL	Any except LOGICAL	Typeless
Extract 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

[1]Remaindering (mod (a_1,a_2)) is defined as $a_1-[a_1/a_2]*a_2$, where the bracketed expression denotes the integer result of the expression a_1 divided by a_2.

[2]These functions are processed by external library subroutines.

Argument Checking and Conversion for Intrinsic Functions

A number of checks on arguments used in 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. Except as noted in Table 6-1, the argument type must agree with the type of the function. With the exception of the typeless functions (described in this section), 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, 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 argument types integer through complex.

Automatic Typing of Intrinsic Functions

Use of the generic forms of the mathematical intrinsic functions (see Table 6-1) allows for the type of the function's value to be determined automatically by the type of the actual arguments supplied. This subset of intrinsic functions contains:

1. Absolute value - ABS
2. Remaindering - MOD
3. Maximum value - MAX
4. Minimum value - MIN
5. Positive difference - DIM
6. 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 (functions allowing more than one argument), the type of the function itself is determined by the same rules that govern mixed mode expressions. See Table 4-1, Rules for Assignment of E to V.

FLD

FLD is used for bit string manipulation and has the following form:

FLD (i,k,e)

where:

i and k are integer expressions where $0 \le i \le 35$ and $1 \le k \le 36$; e is any integer, real, or typeless expression, a Hollerith word or one of the typeless functions listed in Table 6-1.

This function extracts a field of k bits from a 36-bit string represented by e starting 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.

For example:

```
I = 64

J = FLD (29,1,I)

PRINT, "I = ",I

PRINT, "J = ",J
```

This 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 ASF function. The FLD function is defined as follows:

```
FLD (i,j,a) = b
```

where:

> i and j are integer expressions ($0 \leq i \leq 35, 1 \leq j \leq 36$); a is a scalar or subscripted variable; and b is an expression. The j rightmost bits of expression 'b' will be inserted into 'a' beginning at bit position i.

For example, assuming ASCII characters:

```
CHARACTER*4 A,B

A = "ABCD"

B = "1234"

FLD (9,9,A) = B

PRINT, A

PRINT, B
```

This would result in the printing of

```
A4CD

1234
```

Typeless Intrinsic Functions

The five typeless functions are:

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, 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 the receiving variables and K were 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 were of type LOGICAL, the values of the variables would be as follows:

K	M1	M2	M3	M4	M5
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

(T = true)

(F = 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. A two-element array is employed to bypass the requirement of separating integer and alpha 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 that references it. The computation desired in a FUNCTION subprogram is defined by writing the necessary statements in a segment, writing the word FUNCTION and the name of the function before the segment, and writing the word END after the segment. The FUNCTION statement is of the form:

t FUNCTION f $(a_1,a_2,\ldots,a_n)$

where t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or null. The f is the symbolic name of the function to be defined. The a (called dummy arguments) are either variable names, array names, or external procedure names.

The symbolic name of the function must appear at least once in the subprogram as a variable name in some defining context (e.g., left of equals). The value of the variable at the time of execution of any RETURN statement in this subprogram is returned as the value of the function.

The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement or in a Type statement.

An abnormal FUNCTION subprogram can define or redefine one or more of its arguments to effectively return results in addition to the value of the function.

The FUNCTION subprogram can contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined. The FUNCTION subprogram must contain at least one RETURN statement.

If the function name appears in any of the following contexts, redefinition of the function result is effected.

1. Left of equals in assignment statement
2. In the list of a READ statement
3. In the list of a DECODE statement
4. As the buffer name in an ENCODE statement
5. As the induction variable of a DO loop

Redefinition can also occur if the function name appears in the argument list of a CALL statement or a reference to some abnormal external function, though not necessarily.

NOTE: A function cannot be referenced in an input/output list if such a reference causes any input/output operation to be executed.

Supplied FUNCTION Subprograms

The functions listed in Table 6-2 are the basic external FUNCTION mathematical subprograms supplied with the compiler. To use the functions, it is only necessary to write their name where needed and enter the desired expression(s) for argument(s). Except as indicated in Table 6-2, argument types must conform with the type of the function. The compiler does some checking as to type of arguments supplied and makes conversions in accordance with the following rules:

1. The hierarchy of argument types considered for conversion is: integer, real, double precision, complex.
2. A generic function call whose arguments do not conform as to type is transformed to the function type that supports the highest level argument supplied to it.
3. Integer arguments are converted to the type of the function being called.
4. Arguments to a non-generic form of external function are converted to conform to the function type specified. This is within the constraints of argument types integer through complex.

A generic name is assigned to the set of functions in Table 6-2.

When the mathematical library functions are referenced by their generic names, the type of the function is determined by the type of the argument(s) within the constraints of the types described in Table 6-2. The one exception is when an integer argument is specified to a generic function. In this case, the argument is converted and the real form of the function is called. Note that the type of ATAN2 is double precision if at least one of its arguments is double precision.

The functions listed in Table 6-3 are utilized in precisely the same manner as those listed in Table 6-2; they differ only in that they are nonmathematical.

Table 6-2. Supplied FUNCTION Subprograms, Mathematical

Function	Definition	No. of Arg.	Generic Name	Type of: Arg.	Type of: Function
Arccosine[1]	$\cos^{-1}(a)$	1 1	ARCOS DARCOS	Real Double	Real Double
Arccosine, Hyperbolic[1]	$\cosh^{-1}(a)$	1 1	ACOSH DACOSH	Real Double	Real Double
Arcsine[1]	$\sin^{-1}(a)$	1 1	ARSIN DARSIN	Real Double	Real Double
Arcsine, Hyperbolic[1]	$\sinh^{-1}(a)$	1 1	ASINH DASINH	Real Double	Real Double
Arctangent[1]	$\tan^{-1}(a)$	1 1	ATAN DATAN	Real Double	Real Double
	$\tan^{-1}(a/b)$	2 2	ATAN2 DATAN2	Real Double	Real Double
Arctangent, Hyperbolic[1]	$\tanh^{-1}(a)$	1 1	ATANH DATANH	Real Double	Real Double
Cosine[1]	$\cos(a)$	1 1 1	COS DCOS CCOS	Real Double Complex	Real Double Complex
Cosine, Hyperbolic[1]	$\cosh(a)$	1 1	COSH DCOSH	Real Double	Real Double
Cube Root	$(a)^{1/3}$	1 1	CBRT DCBRT	Real Double	Real Double
Exponential	e^{a}	1 1 1	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Exponential	10^{a}	1 1	EXP10 DEXP10	Real Double	Real Double
Exponential	2^{a}	1 1	EXP2 DEXP2	Real Double	Real Double

[1]Arguments expressed in radians.

Table 6-2 (cont). Supplied FUNCTION Subprograms, Mathematical

Function	Definition	No. of Arg.	Generic 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 DXPC2	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 FUNCTION Subprograms, Nonmathematical

Function	Usage	No. of Args.	Type of:	
			Arg.	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

The nonmathematical functions are as follows:

Shift Functions

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.

All are integer functions with integer arguments. (Refer to the Macro Assembler Program (GMAP) manual for a description of shifting functions.)

Set/Reset Program Switch Word

ISETSW(i)	Set program switch word.
IRETSW(i)	Reset program switch word.

The ones in the binary equivalent of the value of i determine the bit positions to be set/reset in the program switch word.

The function returns the new program switch word configuration.

Refer to the General Comprehensive Operating Supervisor (GCOS) manual for a description of the program switch word.

NOTE: Bits 0-17 of the program switch word cannot be changed when operating in the time sharing mode.

Mode Determination

MODE(i)

For i = 1: Function value = 0 for batch execution; function value = 1 for time sharing execution.

For i = 2: Function value = 0 for BCD character mode; function value = 1 for ASCII character mode.

For i $\neq$ 1 or 2: Function value = -1.

Character String Compare

KOMPCH (a,n,b,m,f)

This is a FUNCTION call where string b is compared, starting at position m, to string a, starting at position n; f characters are compared. The resulting value of the integer function is:

Function value = 0 when b = a
Function value = +1 when b is greater than a
Function value = -1 when b is less than a

a and b are character
f, m, and n are integer

See Appendix A for BCD and ASCII character collation (sort) values.

Random Number Generators

There are four separate calls provided for producing a sequence of random numbers. Each call 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 for any other number. The calling sequences are as follows:

RAND (range) $0 < A < \text{range}$

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.

RANDT (range) $0 < A < \text{range}$

The range must be a real constant or variable; the time register value is used as the seed. A different set of random numbers is generated each time the program unit is executed.

FLAT (seed) $0 < A < 1$

This version has a constant range but allows the seed to be varied. The seed must be a real constant or variable.

UNIFM2 (seed,mean,width) $[\text{mean-width}/2] < A < [\text{mean+width}/2]$

This version allows the seed and the range to be varied. For 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 generation of the sequence of random numbers. For all subsequent calls to the function, during the execution of same program unit, the value of the argument is ignored.

Arguments must be real constants or 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, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the FUNCTION subprogram definition. Actual arguments in the function reference can be one of the following:

1. A variable name
2. An array element name
3. An array name
4. An expression
5. Name of an external procedure
6. 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.

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 name, or an array element name.

Execution of a FUNCTION reference results in an association of actual arguments with all appearances of dummy arguments in the defining subprogram. If the actual argument is an expression, or constant, then this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram begins. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of a FUNCTION subprogram is an array name, the corresponding actual argument must be an array name or array element name.

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

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.

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 the correct numerical results, are returned to the calling program.

Example of FUNCTION Subprogram

Definition

```
      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

```
      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; there may be any number of results.
2. A SUBROUTINE is not called into action simply by writing its name, since no value is associated with the name. A CALL statement brings it into operation. The CALL statement specifies the arguments, and results in storing all output values.
3. There is no type or convention associated with the SUBROUTINE name. The naming is otherwise the same as for the FUNCTION.

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 expected by the called routine. This applies for all subroutines and functions (library or other).

Defining SUBROUTINE Subprograms

A SUBROUTINE statement is of the form:

```
SUBROUTINE s (a1,a2,...,an)
            or
SUBROUTINE s
```

where s is the symbolic name of the SUBROUTINE to be defined.

a_i, called dummy arguments, are each a variable name, an array name, an external procedure name, or an alternate return.

The symbolic names of the dummy arguments cannot appear in an EQUIVALENCE, COMMON, NAMELIST or DATA statement.

The SUBROUTINE subprogram can define or redefine one or more of its arguments so as to effectively return results.

The SUBROUTINE subprogram can contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

The SUBROUTINE subprogram 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. An actual argument in the SUBROUTINE reference may be one of the following:

1. a constant
2. a variable name
3. an array element name
4. an array name
5. an expression
6. the name of an external procedure
7. an alternate return

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 name, or an array name.

Execution of a subroutine reference results in an association of actual arguments with all appearances of dummy arguments in the defining subprogram. This association is by name rather than by value.

Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument that is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as 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 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.

Unless it is a dummy argument, a SUBROUTINE is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

SUBROUTINE Subprogram Examples

Defining

```
      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
```

Referencing

```
      CALL LARGE (ZETA,N,VAL,NCOL)
```

Returns From Function And Subroutine Subprograms

The RETURN statement is used to terminate all subprograms. This statement causes control to be returned to the calling program. There may be any number of RETURN statements in a subprogram. The RETURN statement has the form:

```
      RETURN

      or RETURN i
```

where i is an integer constant or variable whose value denotes the nth * or $ in the argument list of the SUBROUTINE statement, reading from left to right.

The normal sequence of execution following the RETURN statement of a subprogram is to the next executable statement following the CALL or function name statement in the calling program. It is possible to return to any numbered executable statement in the calling program by using a special return from the called subprogram (for SUBROUTINE subprograms only). This return must not violate the transfer rules for DO loops. FUNCTION subprograms must not have nonstandard returns.

The following text describes the form of the FORTRAN statements that is required to return from the SUBROUTINE to a statement other than the next executable statement following the CALL.

The general form of the CALL statement in the calling program is:

CALL subr (a_1,a_2,...,a_n)

where subr is the name of the SUBROUTINE subprogram being called. Each a_i is an argument of the form described with the CALL statement or is of the form:

$n

where n is a statement number, or switch variable used for a nonstandard return.

The general form of the SUBROUTINE statement in the called subprogram is:

SUBROUTINE subr (a_1,a_2,...,a_n)

where subr is the name of the subprogram. Each a_i is a dummy argument of the form described under SUBROUTINE subprograms, or is of the form:

* or $

where the * or $ denotes a nonstandard return.

The general form of the RETURN statement in the called subprogram is:

RETURN i

where i is an integer constant or variable which denotes the ith nonstandard return in the argument list, reading from left to right. For example:

```
Calling Program                        Called Subprogram

                                       SUBROUTINE SUB(X,Y,Z,*,*)
      .
      .                                          .
      .                                          .
10 CALL SUB(A,B,C,$30,$40)                       .
20 - - - - -                                     .
      .
      .                            100 RETURN  N
30 - - - - -
      .
      .                                END
      .
40 - - - - -
      .
      .
   END
```

In the preceding example, execution of statement 10 in the calling program causes entry into 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.

Nonstandard returns may best be understood by considering that a CALL statement that uses the nonstandard return is equivalent to a CALL and a computed GO TO statement in sequence. For 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+1
```

where I is set to the value of the integer in the RETURN statement executed in the called subprogram. If the RETURN index is not specified or is zero, a normal (rather than nonstandard) return is made to the statement immediately following the GO TO.

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 are equivalent:

```
     CALL NAME (P,$20,Q,$35,R,$22)
     CALL NAME (P,Q,R,$20,$35,$22)
     CALL NAME ($20,$35,$22,P,Q,R)
```

as are the following:

```
     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. 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.

ENTRY statements are nonexecutable and, therefore, do not affect control sequencing during normal execution of a subprogram. The order, type, and number of arguments need not 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 refers to. No subprogram can refer to itself directly or through any of its entry points, nor can it refer to any other subprogram whose RETURN statement has not been satisfied.

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
```

In the preceding example, the execution of statement 1 causes entry into SUB1, starting with the first executable statement of the subroutine. Execution of statements 2 and 3 also cause entry into the called program, starting with the first executable statement following the ENTRY SUB2(T,U,V) and ENTRY SUB3 statements, respectively.

Dummy Argument

A dummy argument is used to make entities in a referencing program available to the referenced subprogram.

A dummy argument of a subprogram can be associated with an actual argument that can be a variable, array, array element, subroutine, external function, constant, expression, or in the case of subroutines, statement number to which a special return can be made from a subroutine program.

The dummy argument can be used in the subprogram as a scalar variable, an array, a subroutine, or function name.

When the use of a statement number is specified (to which a special return can be made from a subroutine subprogram), the use of the * or $ in a dummy argument position is required if a statement number is associated with that dummy argument.

When the use of 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.

When the use of 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.

Unless otherwise specified, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

Supplied SUBROUTINE Subprograms

Table 6-4 contains a list of FORTRAN supplied SUBROUTINE subprograms. Immediately following the table are explanations of the subprograms, in alphabetical order.

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 ...)
DVCHK	Divide check test.	DVCHK(j)
EXIT	Purge buffers and terminate current activity.	EXIT
FCLOSE	Close file 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 transileration	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 ($n)
LINK	Restore link and transfer to its entry point.	LINK (name)
LLINK	Restore link and return to next statement in calling subroutine.	LLINK (name)
MEMSIZ	Memory allocated.	MEMSIZ (j)
NASTRK	Disables asterisk fill for field overflow on formatted output.	NASTRK
OVERFL	Exponent register overflow or underflow test.	OVERFL (j)
PDUMP (BCD) PDUMPA(ASCII)	Dump designated area of memory in specified format, return.	PDUMP [PDUMPA] ($a_1,b_1,i_1,..$)
PTIME	Processor time.	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	Sort in ascending order.	SORT (array,nrec,lrs, $key_1,...$)
SORTD	Sort in descending order.	SORTD (array,nrec,lrs, $key_1,...$)
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)
YASTRK	Re-establishes asterisk fill for field overflow on formatted output.	YASTRK

Table 6-4 (cont). Supplied SUBROUTINE Subprograms

Subprogram	Use	Call
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)

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> is the catalog/file descriptor; <u>catfil</u> is a character constant, or variable, containing the catalog/file string. It must be terminated by a semicolon. Imbedded blanks are ignored. The master catalog password is not needed; subsequent passwords are required if part of the file description. Alternate names are permitted.

<u>iprmis</u> is an integer variable or constant and is the permission desired. Those are ORed with any permission in the <u>catfil</u>.

= 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 first status word returned by the File Management Supervisor (see the <u>TSS System Programmer's Reference Manual</u>), 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

<u>buffer</u> = Null: Get a file system buffer.
= Not null: Use this variable array as a buffer (at least 380 words).

The following is an example of a null argument:

```
          Call ATTACH (lgu, catfil, iprmis, mode, istat,  )
```

Upon successful return from ATTACH, 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 name is the four-character ASCII constant or variable that is the 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 at the system level.

string is the command to invoke the subsystem and is an ASCII character or variable containing a carriage return or a reverse slant as the terminating character.

For example:

```
CALL CALLSS ("RUN P3\","RUNY")
```

When this statement is executed, the YFORTRAN subsystem is invoked and program P3 is executed.

```
CALLSS ("RUNP1\","BASY")
```

When this statement is executed, the BASIC subsystem is invoked and program P1 is executed.

```
CALLSS ("CATALOG FILENAME\")
```

When this statement is executed, the specific attributes of the file FILENAME are printed.

```
CALLSS ("ABC\")
```

When this statement is executed, the ABACUS subsystem is invoked.

```
CALLSS ("FDUMP\")
```

When this statement is executed, the FDUMP subsystem is invoked.

Nesting to more than two levels using CALLSS is not permitted. 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 none is given, "0000T/" is assumed.

Message is an array containing the CHARACTER message to be output.

Nwords is an integer variable or constant, representing the number of words to be printed. Any value greater than 11 is set to 11.

Nreply is an integer variable or constant (optional) and is used when a reply is desired. When present, the reply (in BCD) will be stored beginning at location nreply.

Nrepws is an integer variable or constant (optional) and is used when a reply of more than six characters is desired. When omitted, a one-word reply will be stored in nreply. When provided, nrepws words (nrepws *6 characters) will be stored beginning at location nreply.

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 = character string to be replaced

n = leftmost character of a (0-3 for ASCII or 0-5 for BCD)

b = replacement character string

m = leftmost replacement character of b (0-3 for ASCII or 0-5 for BCD)

l = number of characters to be replaced; if l is not given, one replacement character is assumed

String a is replaced, starting at position n, by string b, starting at position m; l characters are replaced. The letters m, n, and l are integer variables or constants.

CORFL

This subroutine enables the user to move data from or to the ten-word memory file (see "DRL CORFIL", TSS System Programmer's Reference Manual).

Calling Sequence:

```
     CALL CORFL (loc,i,j,k)
```

where:

<u>loc</u> is the integer array into which or from which the data is to be moved

i is the number of words to be moved such that $1 \leq I \leq 10$

j is the relative location in the 10-word file at which the transfer is to begin.

k = 0, data is transferred into the 10-word file

k = 1, data is transferred from the 10-word file

i,j,k are integer variables or constants.

This call is ignored in batch.

CORSEC

This subroutine provides the user with the means of obtaining the product of a memory allocation and processor time.

Calling Sequence:

```
     CALL CORSEC (a)
```

where the real value returned in a, a real variable, is the product of 1024-word blocks currently allocated and processor time in seconds. This feature can also be used as a function. For 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 file wanted.

<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

istat 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.

DATIM

This subroutine allows the user to obtain the current date and time.

Calling Sequence:

```
CALL DATIM (d,t)
```

D is an eight-character variable and will contain the date in the form mm/dd/yy (with trailing blanks if in BCD mode). T is a real variable and will contain 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 an eight-character or less variable containing the ASCII name of the temporary file to be created.

links size of file to be created (in links)

mode = 0, sequential file is created

≠ 0, random file is created

istat is status indication 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 deaccess a file, close the file and release its buffer. 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)
```

If more memory is required (to deaccess the file) and the request is denied, the time sharing user is terminated.

```
     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.
```

DUMP [DUMPA], PDUMP [PDUMPA]

This subroutine subprogram dumps all of memory 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.

Calling Sequence:

CALL DUMP or DUMPA $(a_1,b_1,j_1,\ldots,a_n,b_n,j_n)$

CALL PDUMP or PDUMPA $(a_1,b_1,j_1,\ldots,a_n,b_n,j_n)$

where a_i and b_i are variables at the beginning and end of area to be dumped. A_i or b_i may represent the first and last variable in the program unit, in which case all memory allocated to variables is dumped. J is an integer specifying the dump format. If no arguments are given, all of memory is dumped in octal. The values for j are as follows:

```
     j = 0, Octal
     j = 1, Integer
     j = 2, Real
     j = 3, Double Precision
     j = 4, Complex
     j = 5, Logical
     j = 6, Character
```

DUMPA and PDUMPA are the ASCII versions.

These subroutine subprograms check logical fault vector locations in the slave program prefix (refer to the General Comprehensive Operating Supervisor (GCOS) manual).

Calling Sequences:

CALL DVCHK(j) to determine if a divide check has occurred.

CALL OVERFL(j) to determine if an exponent register overflow or underflow has occurred.

where: j is an integer variable that is set to one (1) if a divide check, exponent register overflow, or exponent register underflow has occurred; otherwise, j is set to two (2).

To allow another value to be returned after a divide check fault, the following calling sequence is used prior to the statement that might cause the fault:

CALL FXDVCK(r,m)

The value r, which must be double precision, is returned after a floating-point divide check. The value m is returned after an integer divide check. The second argument may be omitted.

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 the value returned as follows:

Fault	Value Returned	
Divide check (integer)	No change	Unless CALL FXDVCK is used.
Divide check (floating-point)	A large floating-point value[1]	
Overflow (integer)	No change	
Exponent overflow	A large floating-point value[1]	
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 purges all buffers and terminates the current activity. Control is returned to the General Comprehensive Operating Supervisor.

Calling Sequence:

```
     CALL EXIT
```

FCLOSE

This subroutine subprogram closes file i without rewind and releases the buffer(s) assigned. The buffer is released only 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 allow users generating multifile tapes to space from one file to another (only valid with tape files).

Calling Sequence:

```
     CALL FILBSP (lgu,n) backspace n files

     CALL FILFSP (lgu,n) forwardspace n files
```

where

<u>lgu</u> = integer variable or constant (file code)
<u>n</u> = number of files to skip (integer variable or constant)

To ensure proper positioning, the current file, if output, should be closed with ENDFILE statement and counted as one of the files to be backspaced over. Declare the files to be multifile and unlabeled by use of $ FFILE card. For example,

```
     $ FFILE  xx,MULTFIL,NSTDLB
```

FLGEOF

This subroutine subprogram provides a signal requesting a 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 constant.

J is an integer variable used to indicate an end-of-file condition (user must test j (for a nonzero) when an end-of-file condition could have occurred). J should not generally be used for any other purpose.

FLGERR

This subroutine subprogram provides a variable used in detecting the occurrence of erroneous data. 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 constant.

J is an integer variable used to indicate an input data error (user must test j (for a nonzero value) when an error condition could have occurred). J should not generally be used for any other purpose.

FLGFRC

This subroutine provides the user with some control of File and Record Control errors. The user can set his error routine address into the file control block. This subroutine should be called prior to the first I/O for this file.

Calling Sequence:

```
     CALL FLGFRC (lgu,ptr)
```

where

lgu = an integer variable or constant representing the numberic file code

ptr = 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 GMAP subroutine could obtain the status code by executing a LDXn 3,1.

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 media (integer variable or constant)
- = 0 for BCD no slew
- = 2 for BCD card images
- = 3 for BCD slew
- = 5 for TSS ASCII format (Obsolete)
- = 6 for standard system ASCII format (no slew)

Others are ignored.

fc = logical file code (integer variable or constant)

The legal combinations are as follows:

0 to 2	3 to 0
0 to 3	3 to 2
0 to 5	3 to 5
0 to 6	3 to 6
2 to 0	6 to 0
2 to 3	6 to 2
2 to 5	6 to 3
2 to 6	6 to 5

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 one (1).

3. Set the reflexive read characters that are sent to a terminal to request input. The default setting of this parameter is the ASCII CHARACTER constant 'carriage return', 'line feed', 'equal sign', 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.

Example of reflexive read signature changed to "??"; in which 015 is a carriage return, 012 is line feed, and the two 077s are question marks:

```
    DATA J/O015012077077/

    CALL FPARAM (3,J)
```

Example of terminal line length setting to 160 characters:

```
    CALL FPARAM (1,160)
```

FXDVCK (see DVCHK)

FXEM (FORTRAN EXECUTION ERROR MONITOR)

This subroutine performs the following functions:

1. Prints a trace of subroutine calls.
2. Prints execution error messages.
3. Terminates execution with a Q6 abort or does one of the following:
   a. Continues with execution of the program.
   b. Transfers to an alternate error routine.
4. 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. See Appendix E for FXEM examples.

Calling Sequences:

```
    CALL ANYERR(v)
```

where v is a 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.

CALL FXEM(ncode,msg,n)

This call causes the display on file 06 of an error trace and the message contained in msg which must be a character constant or variable. The number of words, n, to be printed must be within the limits $0 \leq n \leq 20$. If only the first argument is given, only the trace is printed. Ncode is the error code (see Table 6-5) expressed as an integer in the range $1 \leq n \leq 143$.

CALL FXOPT(ncode,i1,i2,i3)

This call to the Execution Error Monitor is used to alter the standard switch word settings listed in Table 6-5. Ncode is the error code. I1, i2, i3 refer to the switch words settings for termination, message printing and trace, and alternate error procedure respectively. If i1=1, continue execution; if=0, abort with a Q6 abort. If i2=1, suppress printing; if=0, print. If i3=1, use alternate error procedure; if=0, use normal return. This option overrides the termination option.

Examples:

1. CALL FXOPT(32,0,1,0)

2. CALL FXOPT(32,1,0,0)

3. CALL FXOPT(32,0,0,1)

Example 1 causes a Q6 abort, for error number 32, when the error occurs. No message or trace is printed. Example 2 causes execution to continue after message and trace are printed. Example 3 indicates that return is to an alternate error routine after trace and message are printed. The alternate return takes precedence over termination.

CALL FXALT(SR)

The FXALT call is used to set the alternate error procedure location. SR is the alternate error procedure subroutine; it is used as the address to which the error monitor transfers. An EXTERNAL SR must be included in the calling routine. FUNCTION subprograms and parameters are not allowed. If the alternate procedure option for an error code is indicated but no call to FXALT has been made, a Q5 abort occurs when the 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 must not invoke the routine in which the error was found; i.e., the alternate error procedure for a formatted input/output statement must not 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→QR	EXPONENTIATION ERROR 0**0	SET RESULT=0
2	C	I**J	I=0,J<0	(2**35)-2→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→EAQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
4	C	{A**J, DA**J}	{A=0,J<0, DA=0,J<0}	10**38→EAQ	EXPONENTIATION ERROR 0**(-J)	SET RESULT=10**38
5	C	B**C	B<0	0→EAQ	EXPONENTIATION ERROR (-B)**C	SET RESULT=0
6	C	A**B	A=0,B=0	0→EAQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
7	C	A**C	A=0,C<0	10**38→EAQ	EXPONENTIATION ERROR 0**(-C)	SET RESULT=10**38
8	C	e**B	B>88.028	10**38→EAQ	EXP(B),B GRT THAN 88.028 NOT ALLOWED	SET RESULT=10**38
9	C	LOG(A)	A=0	-(10**38)→EAQ	LOG(0) NOT ALLOWED	SET RESULT-(10**38)
10	C	LOG(B)	B>0	0→EAQ	LOG(-B) NOT ALLOWED	SET RESULT=0.0
11	C	ARCTAN(A/B)	A=0,B=0	0→EAQ	ATAN2(0,0) NOT ALLOWED	SET RESULT=0
12	C	{SIN(A), COS(A)}	$\|A\|>2^{27}$	0→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→AQ	EXPONENTIATION ERROR 0**0	SET RESULT=0
15	C	CA**J	CA=0,J<0	10**38→AR 0→QR	EXPONENTIATION ERROR 0**(-J)	SET RESULT=(10**38,0.0)
16	C	DA**DB	DB≠0,DA < 0	0→EAQ	EXPONENTIATION ERROR (-DA)**DB	SET RESULT=0
17	C	DA**DB	DA=0,DB=0	0→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	$10**38 \rightarrow$ EAQ	EXPONENTIATION ERROR 0**(-DB)	SET RESULT=10**38
19	C	e**DA	DA > 88.028	$10**38 \rightarrow$ EAQ	EXP(B),B GRT 88.028, NOT ALLOWED	SET RESULT=10**38
20	C	LOG(DA)	DA=0	$-(10**38) \rightarrow$ EAQ	DLOG(0) NOT ALLOWED	SET RESULT=-(10**38)
21	C	LOG(DA)	DA < 0	$0 \rightarrow$ EAQ	DLOG(-B) NOT ALLOWED	SET RESULT=0
22	C	$\sqrt{DA}$	DA < 0	$\sqrt{DA}=\sqrt{\lvert DA \rvert}$	SQRT(-B) NOT ALLOWED	EVALUATE FOR +B
23	C	$\left\{\begin{array}{l}\text{SIN DA}\\ \text{COS DA}\end{array}\right\}$	$\lvert DA \rvert > 2^{54}$	$0 \rightarrow$ EAQ	DSIN OR DCOS ARG GRT 2**54 NOT ALLOWED	SET RESULT=0
24	C	ARCTAN(DA/DB)	DA=0,DB=0	$0 \rightarrow$ EAQ	DATAN2(0,0) NOT ALLOWED	SET RESULT=0
25	C	CA/CB	CB=(0,0)	$10**38 \rightarrow$ AR $10**38 \rightarrow$ QR	COMPLEX Z/0 NOT ALLOWED	SET RESULT=(10**38, 10**38)
26	C	e**CA	REAL CA>88.028	$10**38 \rightarrow$ AR $10**38 \rightarrow$ QR	EXP(Z),REAL PART GRT 88.028 NOT ALLOWED	SET RESULT=(10**38, 10**38)
27	C	e**CA	$\lvert$IMAG CA$\rvert > 2^{27}$	$0 \rightarrow$ AR $0 \rightarrow$ QR	EXP(Z),IMAG PART GRT 2**27 NOT ALLOWED	SET RESULT=(0,0)
28	C	LOG(CA)	CA=(0,0)	$-(10**38) \rightarrow$ AR $0 \rightarrow$ QR	CLOG(0) NOT ALLOWED	SET RESULT (-(10**38),0.0)
29	C	$\left\{\begin{array}{l}\text{SIN(CA)}\\ \text{COS(CA)}\end{array}\right\}$	$\lvert$REAL(CA)$\rvert > 2^{27}$	$0 \rightarrow$ AQ	CSIN OR CCOS ARG WITH REAL PART GRT 2**27 NOT ALLOWED	SET RESULT=0
30	C	$\left\{\begin{array}{l}\text{COS(CA)}\\ \text{SIN(CA)}\end{array}\right\}$	$\lvert$IMAG(CA)$\rvert$ >88.028	$10**38 \rightarrow$ AR $10**38 \rightarrow$ 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	A	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
93-99		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

LINK AND LLINK

The LINK subroutine enables the programmer to call program overlays in the batch mode. See Section III for TSS overlay linking via the RUNL command where a special form of LINK is used. The following call is used to invoke a link and transfer control to it without returning to the calling program/overlay.

```
     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 must be 5-8 characters if using the ASCII option. Explicit trailing blanks are included in the character count.

To load a link and return to the next sequential statement of the calling routine, the required statement is:

```
     CALL LLINK(name)
```

NOTE: Due to 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 activity requires the presence of a PRINT statement or another form of character input-output statement in the main program.

MEMSIZ

This subroutine provides the user with the means of obtaining the amount of memory allocated to the compilation.

Calling Sequence:

```
     CALL MEMSIZ(j)
```

where the value returned in j, an integer variable, is the number of 1024-word blocks currently allocated this job. This feature can also be used as a function.

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 user with the means of obtaining processor time used.

Calling Sequence:

```
     CALL PTIME(a)
```

where the value returned in a, a real variable, is the processor time used in hours. 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 for the current user.

RANSIZ

This subroutine permits the user 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 (first I/O to) the file is mandatory.

Calling Sequence:

```
     CALL RANSIZ (u,n,m)
```

where: u is the logical file designator.

n is the record size.

m is a file format indicator.

U, n, and m must be of type integer. They can be any legal arithmetic expression.

Note that a call to RANSIZ can also be used to override a $ FFILE size specification and that this is the preferred method of specification since its function works for both batch and time sharing use of FORTRAN.

The third argument (m) is optional. When not supplied, file u is processed in standard system format (blocked, variable length records, etc.). When supplied, zero indicates standard system format; nonzero indicates that block and record control words are not to be processed. This latter format provides compatibility with random files generated by time sharing FORTRAN. The total file space is available for data; records are not blocked, can begin anywhere in a sector and may span device boundaries.

SETBUF

This subroutine allows the user to assign space in storage for use as an input/output buffer(s). The size of the buffer(s) must be one greater than the actual record size. The standard buffer size is therefore 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 is the logical file designator, an integer variable, or constant.

A is the array name of the first buffer.

B is the array name of the second buffer, if required.

SETFCB

This subroutine allows the user 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 except for the following possible error conditions:

1. Abort with a Q2 if there is no logical file table.
2. Abort with a Q1 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 allows the user to define a logical unit table for use by the I/O library subprograms. Normal return is to the next executable statement of the calling program. This subroutine must be called before any input/output is requested. It is called when the user wants to suppress the logical file table generated by the General Loader and to place the table in his own portion of memory. The NOFCB option must be specified in the $ OPTION control card.

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 table 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 Sequences:

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 on

where i is an integer variable or constant.

J is an integer variable which is set to 1 if sense light i was ON; set to 2 if sense light i was OFF. J can not be the induction variable of a currently active DO loop.

SORT

This subroutine provides the user with the means of sorting integer or character arrays in an ascending order.

Calling Sequence:

CALL SORT (array,nrec,lrs,key_1,...,key_n)

where:

<u>array</u> is the name of the array to be sorted;

<u>nrec</u> is an integer variable or constant and is the number of items, or logical records, in the array;

<u>lrs</u> is an integer variable or constant and is the logical record size, or the size of each item in the array in words. For integer arrays, this is always one. For character arrays, the record size can be determined by realizing that there are four characters per word in ASCII and six characters per word in BCD. For example, a CHARACTER *12 item would have <u>lrs</u> of 2 in BCD, 3 in ASCII.

<u>key</u> is the relative word number of the ith sort key in each logical record and must be such that $0 \leq \text{key}_i < \text{lrs}$. 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 2-dimensional array sort.

```
0010 DIMENSION ARR(3,5)
0020 PRINT,"INPUT DATA"
0030 READ(5,10)ARR
0040 10 FORMAT(3A5)
0050 CALL SORT(ARR,5,3,0,1,2)
0060 PRINT,"SORTED DATA"
0070 WRITE(6,10)ARR
0080 STOP;END

ready

*RUN=(BCD)
INPUT DATA
=ELK FAST 1
=COW SLOW 2
=DOG FAST 3
=CAT FAST 4
=ELK FAST 5
SORTED DATA
CAT FAST 4
COW SLOW 2
DOG FAST 3
ELK FAST 1
ELK FAST 5
```

SORTD

This subroutine provides the user with the means of sorting integer or character arrays in a descending order.

Calling Sequence:

CALL SORTD (array,nrec,lrs,key_1,...,key_n)

where:

<u>array</u> is the name of the array to be sorted;

<u>nrec</u> is an integer variable or constant and is the number of items, or logical records, in the array;

<u>lrs</u> is an integer variable or constant and is the logical record size, or the size of each item in the array in words. For integer arrays, this is always one. For character arrays, the record size can be determined by realizing that there are four characters per word in ASCII and six characters per word in BCD. For example, a CHARACTER *12 item would have a <u>lrs</u> of 2 in BCD, 3 in ASCII.

<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 such that $0 \leq key_i < lrs$. 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.

The example following builds an array of 10 random numbers from 0 to 20 based on the timer register. This array is then sorted in ascending order and printed, followed by a descending sort and print.

```
0005 DIMENSION ARRAY(10)
0010 DO 10 I=1,10
0020 10 ARRAY(I)=RANDT(20.0)
0030 CALL SORT(ARRAY,10,1,0)
0040 WRITE(6,200)ARRAY
0042 CALL SORTD(ARRAY,10,1,0)
0043 WRITE(6,200)ARRAY
0050 200 FORMAT(10(2X,F5.2))
0060 STOP;END

ready

*RUN
 0.85   1.08   5.73   6.29   8.08  11.61  11.84  12.15  13.74  15.17
15.17  13.74  12.15  11.84  11.61   8.08   6.29   5.73   1.08   0.85
```

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 is set to 1 if the switch i is ON and is set to 2 if the switch is OFF. J can not be the induction variable of a currently active DO loop.

Bits 6-11 of the Program Switch Word, described in the _General Comprehensive Operating Supervisor_ reference manual, correspond to sense switches 1-6.

TERMNO

This subroutine provides the user with the means of obtaining station code.

Calling Sequence:

    CALL TERMNO (a)

where the value returned in a, a character variable, is a 2-character station code. In batch, the call returns blanks.

TERMTM

This subroutine provides the user with the means of obtaining log-on time. The call is applicable only for time sharing activities; it is ignored in the batch mode.

Calling Sequence:

```
     CALL TERMTM (a)
```

where the value returned in a, a real variable, is the hours since log-on.

TRACE

This subroutine is callable from a FORTRAN object program in the time sharing mode. It is useful in tracing and debugging an object module. See the Debug and Trace Routines manual.

USRCOD

This subroutine provides the user with the 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 the value returned in s, a character variable, is a 12-character user identification.

YASTRK

This subroutine may be called to override a 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. See subroutine NASTRK.

Calling Sequence:

```
     CALL YASTRK
```

APPENDIX A

ASCII/BCD CHARACTER SET

ASCII CHAR	Octal	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	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	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

DIAGNOSTIC ERROR COMMENTS

The error detection capabilities of FORTRAN are extensive, including over 500 unique compiler diagnostics. Each diagnostic has zero, one, or two "plug-in" fields, as appropriate to the message. 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 the one having a value that is less than or equal to the line number printed.

The general form of a diagnostic line is as follows:

```
*****S    nnnn    text
```

where S is a severity code, nnnn is an error identification code, and text is the diagnostic message. There are three severity codes as follows:

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 error identification code may be used to reference the tabulation of error codes given in this appendix; it has a number of from one to four digits.

A correspondence of error codes with the compiler module detecting the error is as follows:

Error Number	Compiler Module
1- 199	Executive
200- 299	Phase 2
300- 399	Phase 3
400- 499	Phase 4
1000-1499	Phase 1

In the subsequent tabulation of diagnostics, phase 1 errors are numbered 1 through 485. These correspond to error identification codes 1001 through 1462 respectively. Diagnostics pertinent to the other phases are shown by actual error identification code.

The text of the diagnostic messages may include "plug-in" information, to more fully detail the nature of the error. There are eleven types of plug-ins. In the subsequent tabulation, a text insertion is indicated by a number, one through eleven, enclosed in slants. The interpretation of that number is as follows:

/No./ in Manual listing	Actual value plugged in for error
1	Variable name, constant, or statement number
2	One of the operator codes from Table B-2
3	One of the statement classification types from Table B-3
4	Character
5	One of the type codes from Table B-4
6	Number
7	One of the statement types from Table B-1
8	One of the Lexical classifications from Table B-6
9	One of the program types from Table B-5
10	Four characters
11	Four or six characters

The following abbreviation is used in the listing:

ASF Arithmetic Statement Function

Table B-1. List of Statement Types for Code /7/

ABNORMAL	RETURN
EXTERNAL	BACKSPACE
CHARACTER	BLOCK DATA
DOUBLE PRECISION	CALL
COMPLEX	COMMON
INTEGER	CONTINUE
LOGICAL	DATA
REAL	DIMENSION
READ	DO
WRITE	END
PRINT	ENTRY
PUNCH	EQUIVALENCE
ENCODE	FUNCTION
DECODE	GO TO
PAUSE	FORMAT
STOP	IF
REWIND	IMPLICIT
ENDFILE	NAMELIST
ASSIGN	SUBROUTINE
PARAMETER	

Table B-2. List of Operator Codes for Code /2/

.OR.	*
.AND.	.EQ.
.NOT.	.NE.
+	.LT.
-	.GE.
/	.LE.
**	.GT.

Table B-3. List of Statement Classification Types Code /3/

block name
intrinsic function
array
scalar
arithmetic statement function
function
abnormal function
subroutine
external
parameter
entry name
namelist name
do loop index
adjustable dimension
bad insert

Table B-4. List of Types Code /5/

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
CHARACTER
LOGICAL
TYPELESS

Table B-5. List of Program Types Code /9/

MAIN
SUBROUTINE
BLOCK DATA

Table B-6. List of Lexical Classifications Code /8/

VARIABLE
CONSTANT
OPERATOR
(
)
,
END OF STATEMENT
=
'

PHASE1 ERROR MESSAGES

SECTION I - Abnormal Statement

1.	F	VARIABLE NAME IS MISSING BEFORE FIRST , IN ABNORMAL STATEMENT
2.	F	VARIABLE NAME IN /7/ IS MISSING, STARTS WITH DIGIT OR IS 8 CHARACTERS
3.	F	/8/ FOLLOWING , IN ABNORMAL STATEMENT IS ILLEGAL
4.	F	/8/ FOLLOWING /1/ IN ABNORMAL STATEMENT IS ILLEGAL
5.	F	/4/ FOLLOWING 'ABNORMAL' IS ILLEGAL

SECTION II - Assignment Statement

6.	F	UNEXPECTED /8/ ENCOUNTERED IN /7/ STATEMENT
7.	F	/5/ CANNOT BE ASSIGNED TO A LOGICAL VARIABLE
9.	F	LEFT HAND SIDE OF ASSIGNMENT STATEMENT HAS AN ERROR IN TYPING
10.	F	TYPE OF RIGHT HAND SIDE OF ASSIGNMENT IS INCOMPATIBLE WITH LEFT HAND SIDE
273.	W	TYPE OF RIGHT HAND SIDE OF ASSIGNMENT IS INCOMPATIBLE WITH LEFT HAND SIDE

SECTION III - Arithmetic Statement Function

11.	F	/1/ IS AN UNDIMENSIONED ARRAY OR MISPLACED ASF
12.	F	UNEXPECTED /4/ FOLLOWING BALANCING PARENTHESES
13.	F	ILLEGAL STATEMENT OR /1/ USED PREVIOUSLY IN ASF ARGUMENT LIST
14.	F	ILLEGAL STATEMENT OR /8/ USED ILLEGALLY IN ASF
15.	F	ILLEGAL STATEMENT OR RIGHT PAREN AFTER , IN ASF IS ILLEGAL
16.	F	ILLEGAL STATEMENT OR = IS MISSING IN /7/ STATEMENT
224.	F	ILLEGAL STATEMENT OR UNEXPECTED /8/ IN ASF

SECTION IV - Assigned GOTO Statement

17.	F	'TO' IS MISSPELLED OR MISSING IN 'ASSIGN' STATEMENT
18.	F	STATEMENT NUMBER IS MISSING IN 'ASSIGN' STATEMENT
19.	F	LABEL IN ASSIGN STATEMENT IS NOT BETWEEN 1 AND 99999
20.	F	'TO' AND SWITCH NAME ARE MISSING IN 'ASSIGN' STATEMENT
21.	F	SWITCH NAME IS MISSING IN 'ASSIGN' STATEMENT
22.	F	SWITCH NAME IN 'ASSIGN' STATEMENT CONTAINS MORE THAN 8 CHARACTERS

SECTION V - CALL Statement

23.	F	/8/ FOLLOWING ')' IN CALL IS ILLEGAL, EXPECTING END OF STATEMENT
24.	F	SUBROUTINE NAME IS MISSING IN CALL STATEMENT
25.	F	MORE THAN 8 CHARACTERS IN SUBROUTINE NAME IN CALL STATEMENT
26.	F	/1/ USED AS /3/ , INVALID AS A SUBROUTINE NAME

SECTION VI - DATA Statement

27.	F	FIRST VARIABLE NAME IS MISSING IN /7/ STATEMENT
28.	W	EXPECTING COMMA AFTER /
29.	F	/8/ FOLLOWING LEFT PAREN OR COMMA IN /7/ STATEMENT IS ILLEGAL
30.	F	NON-INTEGER CONSTANT IN ARRAY SUBSCRIPT
31.	F	IDENTIFIER /1/ ON DATA LIST HAS NOT BEEN DIMENSIONED
32.	F	/8/ IS ILLEGAL FOLLOWING LEFT PAREN IN DATA STATEMENT
33.	F	UNEXPECTED /8/ FOLLOWING /1/ IN DATA STATEMENT
34.	F	/8/ FOLLOWING * IN SUBSCRIPT OF DATA STATEMENT IS ILLEGAL
35.	F	/8/ FOLLOWING /1/ IN SUBSCRIPT OF DATA STATEMENT IS ILLEGAL
36.	F	UNEXPECTED /8/ FOLLOWING + IN SUBSCRIPT OF DATA STATEMENT
37.	F	NUMBER OF SUBSCRIPTS FOR /1/ DOES NOT MATCH NUMBER IN DECLARATION
38.	F	/8/ FOLLOWING RIGHT PAREN OF /1/ IN DATA STATEMENT IS ILLEGAL
39.	F	CONTROL INDEX /1/ IS OUTSIDE IMPLIED DO LOOP
40.	F	/8/ IS ILLEGAL FOLLOWING = IN DATA STATEMENT
41.	F	/8/ FOLLOWING FIRST , IN DATA STATEMENT IS ILLEGAL
42.	F	/8/ FOLLOWING SECOND , IN IMPLIED DO LOOP IS ILLEGAL
43.	F	UNEXPECTED /8/ FOLLOWING STEP IN IMPLIED DO LOOP IN DATA STATEMENT

44.	F	UNEXPECTED /8/ FOLLOWING RIGHT PARENTHESIS TERMINATING IMPLIED DO LIST
45.	F	UNEXPECTED /2/ FOLLOWING VARIABLE OR RIGHT PAREN ON DATA LIST
46.	F	IMPLIED DO SPECIFICATION IS MISSING
47.	F	/8/ FOLLOWING / IN DATA LIST IS ILLEGAL STATEMENT
48.	F	SUBSCRIPT IN DATA STATEMENT IS NOT OF FORM I*J+K OR I*J-K
49.	F	SUBSCRIPTS AND IMPLIED DO PARAMETERS MUST BE INTEGERS
445.	F	UNEXPECTED /8/ FOLLOWING /1/ IN DATA STATEMENT
446.	F	UNEXPECTED /8/ FOLLOWING COMMA IN DATA VARIABLE LIST
447.	F	UNEXPECTED /8/ FOLLOWING THE IMPLIED DO PARAMETER /1/
448.	F	VARIABLE /1/ ON IMPLIED DO LIST IS NOT A DO CONTROL
456.	F	ILLEGAL/8/ IN OR FOLLOWING DATA SUBSCRIPT EXPRESSION

SECTION VII - DO Statement

50.	F	RECORD NUMBER IN /7/ MUST BE TYPE INTEGER
51.	F	LABEL FOR DO-END IS NOT BETWEEN 1 AND 99999
53.	F	EXECUTABLE STATEMENTS ARE ILLEGAL HERE
55.	F	INDEX VARIABLE FOR DO STATEMENT IS MISSING, STARTS WITH DIGIT OR > 8 CHARACTERS
56.	F	UNEXPECTED /8/ FOLLOWING INITIAL PARAMETER OF /7/ STATEMENT
57.	F	DO END STATEMENT NUMBER /1/ IS MISSING
253.	F	UNEXPECTED /8/ AFTER INITIAL PARAMETER OF DO STATEMENT
306.	F	DO LOOP INDEX /1/ MUST BE AN INTEGER
307.	F	/3/ /1/ ILLEGAL AS CONTROL INDEX
308.	F	NESTED DO LOOPS ARE USING /1/ AS THE SAME INDEX
309.	F	ADJUSTABLE DIMENSION /1/ MAY NOT BE USED AS A DO LOOP INDEX

SECTION VIII - Entry, Function, Subroutine, Blockdata Statements

58.	F	BLOCKDATA STATEMENT IS OUT OF PLACE
59.	F	ENTRY STATEMENT IS ILLEGAL IN /9/
60.	F	ENTRY MAY NOT BE DECLARED INSIDE DO LOOP

61.	F	ILLEGAL CONTINUATION LINE FOLLOWING RIGHT PAREN
62.	F	THE ARGUMENT /1/ APPEARS TWICE IN ENTRY LIST
63.	F	/2/ IS ILLEGAL IN ARGUMENT LIST FOR /7/ STATEMENT
64.	F	OPTIONAL RETURN * OR $ IS ILLEGAL IN /7/ STATEMENT
65.	F	UNEXPECTED /8/ IN DUMMY ARGUMENT LIST
66.	F	/8/ AFTER ')' IS ILLEGAL IN ARGUMENT LIST OF /7/ STATEMENT
68.	F	MISSING NAME IN /7/ STATEMENT
69.	F	MORE THAN 8 CHARACTERS IN NAME OF /7/
70.	F	FUNCTION DEFINITION FOR /1/ MUST BE FIRST STATEMENT OF SUBPROGRAM
71.	F	FUNCTION HAS NO NAME
72.	F	'SUBROUTINE' STATEMENT MUST APPEAR FIRST IN SUBPROGRAM
454.	F	FUNCTION ENTRY MUST HAVE AN ARGUMENT LIST

SECTION IX - Format Statement

8.	F	UNEXPECTED /4/ FOLLOWING SIGNED NUMBER - EXPECTING A P
79.	F	FORMAT STATEMENT DOES NOT HAVE A STATEMENT LABEL
85.	F	ONLY TWO LEVELS OF NESTED PARENTHESES ALLOWED IN FORMAT STATEMENT
111.	F	X OR H FIELD IN FORMAT STATEMENT MUST HAVE A NUMERIC PREFIX
121.	F	'T' SPECIFICATION CANNOT HAVE A NUMERIC PREFIX
161.	F	THE LETTER /4/ IS ILLEGAL IN A 'P' SPECIFICATION
164.	F	NO WIDTH FIELD IN FORMAT SPECIFICATION
170.	F	'W' FIELD CANNOT BE ZERO IN FORMAT SPECIFICATION
180.	F	UNEXPECTED END OF STATEMENT IN /7/ SPECIFICATION
181.	F	TOO MANY CONTINUATION LINES IN FORMAT SPECIFICATION
185.	F	THE CHARACTER /4/ APPEARS ILLEGALLY IN A FORMAT SPECIFICATION
188.	F	COMPILER ERROR WHILE PROCESSING FORMAT STATEMENT
222.	F	THE CHARACTERS PRECEDING ABOVE FORMAT ERROR ARE /11/
257.	F	'P' SPECIFICATION IN FORMAT MUST HAVE NUMERIC PREFIX
282.	F	INCOMPATIBLE W. D FIELD IN /4/ SPECIFICATION

SECTION X - Intrinsic Functions

73.	F	AN ARGUMENT OF /1/ IS /5/
74.	F	TOO MANY ARGUMENTS FOR /1/ IN /7/ STATEMENT

75.	F	TOO FEW ARGUMENTS FOR /1/ IN /7/ STATEMENT
240.	F	AN ARGUMENT OF /1/ IS NOT TYPELESS OR INTEGER

SECTION XI - CALL or FUNCTION Arguments

76.	F	/3/ /1/ ILLEGAL AS AN ARGUMENT OF FUNCTION OR CALL

SECTION XII - Equivalence Statement

77.	F	UNEXPECTED /8/ IN EQUIVALENCE GROUP - EXPECTING VARIABLE NAME
78.	F	/8/ IS ILLEGAL AFTER A VARIABLE NAME IN EQUIVALENCE STATEMENT
80.	F	SUBSCRIPTS FOR /1/ MUST BE POSITIVE INTEGERS OR INTEGER PARAMETERS
81.	F	DIMENSION CONSTANT IS TOO LARGE
82.	F	MORE THAN 7 DIMENSIONS ARE SPECIFIED FOR /1/
83.	F	/8/ IS ILLEGAL AFTER PARAMETER OR CONSTANT IN /7/ STATEMENT
84.	F	/8/ AFTER ')' FOLLOWING DIMENSIONS IN EQUIVALENCE STATEMENT IS ILLEGAL
86.	F	/8/ IS ILLEGAL AFTER ')' CLOSING GROUP IN EQUIVALENCE STATEMENT
87.	F	EQUIVALENCE GROUP MUST START WITH LEFT PAREN
310.	W	COMMA MUST PRECEDE START OF EQUIVALENCE GROUP FOLLOWING A RIGHT PAREN

SECTION XIII - External Statement

88.	F	FIRST VARIABLE NAME IN EXTERNAL STATEMENT IS MISSING
89.	F	VARIABLE NAME IN /7/ STATEMENT IS > 8 CHARACTERS
90.	F	/8/ FOLLOWING , IN EXTERNAL STATEMENT IS ILLEGAL
91.	F	/8/ FOLLOWING /1/ IN EXTERNAL STATEMENT IS ILLEGAL
92.	F	/8/ FOLLOWING RIGHT PAREN IN EXTERNAL STATEMENT IS ILLEGAL

SECTION XIV - File Designators

93.	F	ILLEGAL CHARACTER AFTER FILE REFERENCE CONSTANT IN /7/ STATEMENT
94.	F	FILE REFERENCE /1/ IN /7/ STATEMENT IS NOT AN INTEGER
95.	F	FILE REFERENCE IS MISSING IN /7/ STATEMENT
96.	F	FILE REFERENCE IN /7/ STATEMENT IS > 8 CHARACTERS

SECTION XV - GOTO Statement

67.	F	UNEXPECTED /8/ IN GO TO LIST
97.	F	ILLEGAL CHARACTER IN STATEMENT NUMBER IN /7/ STATEMENT
98.	F	STATEMENT NUMBER OR SWITCH IN GO TO STATEMENT IS MISSING OR ILLEGAL
99.	F	ILLEGAL SWITCH VARIABLE OR LABEL IN GO TO STATEMENT
100.	F	INVALID SWITCH OR STATEMENT NUMBER IN COMPUTED GO TO
101.	F	/8/ FOLLOWING /1/ IN COMPUTED GO TO IS ILLEGAL
102.	F	/8/ FOLLOWING RIGHT PAREN IN /7/ STATEMENT IS ILLEGAL
103.	F	/8/ IS ILLEGAL AS TERMINATOR FOR COMPUTED GO TO STATEMENT
104.	F	COMPUTED GO TO EXPRESSION MUST BE TYPE INTEGER
105.	F	COMPILER ERROR IN CONTROL TEST
106.	F	SWITCH /1/ IN /7/ STATEMENT IS NOT TYPE INTEGER
107.	F	/8/ FOLLOWING /1/ IN COMPUTED OR ASSIGNED GO TO IS ILLEGAL
108.	F	/8/ FOLLOWING FIRST ',' IN ASSIGNED GO TO IS ILLEGAL, EXPECTING '('
109.	F	/8/ FOLLOWING STATEMENT NUMBER IN ASSIGNED GO TO IS ILLEGAL
110.	F	STATEMENT NUMBER /1/ IN /7/ STATEMENT IS NOT INTEGER
112.	F	LABEL IN GO TO STATEMENT IS NOT BETWEEN 1 AND 99999
450.	F	GO TO LIST HAS NON INTEGER LABEL

SECTION XVI - Arithmetic IF Statement

113.	F	MISSING RIGHT PAREN IN EXPRESSION SECTION OF /7/ STATEMENT
114.	F	/5/ IS ILLEGAL FOR THE EXPRESSION SECTION OF /7/ STATEMENT
115.	F	ONLY STATEMENT NUMBERS OR SWITCHES ARE LEGAL AFTER EXPRESSION SECTION OF IF
116.	F	ILLEGAL CONSTRUCT OF IF STATEMENT
117.	F	/8/ IS ILLEGAL AFTER A STATEMENT NUMBER OR SWITCH IN ARITHMETIC IF
118.	F	/8/ IS ILLEGAL AFTER THIRD STATEMENT NUMBER OR SWITCH IN ARITHMETIC IF
119.	F	TOO MANY STATEMENT NUMBERS, SWITCHES OR FIELDS IN AN ARITHMETIC IF STATEMENT
120.	F	MISSING A STATEMENT NUMBER, SWITCH OR FIELD IN AN ARITHMETIC IF STATEMENT

SECTION XVII - Logical IF Statement

52.	F	THE TRUTH CLAUSE OF A LOGICAL IF MAY NOT BE /7/ STATEMENT
122.	F	MISSING RIGHT PAREN IN THE EXPRESSION SECTION OF /7/ STATEMENT
123.	F	MISSING VARIABLE NAME BEFORE = IN THE TRUTH CLAUSE OF LOGICAL IF STATEMENT
443.	F	TRUTH CLAUSE OF A LOGICAL IF CANNOT BE NULL

SECTION XVIII - Implicit Statement

124.	F	/8/ FOLLOWING TYPE DESIGNATION IN IMPLICIT STATEMENT IS ILLEGAL
125.	F	/8/ AFTER * IN /7/ STATEMENT IS ILLEGAL
126.	F	/8/ AFTER SIZE OPTION IN /7/ STATEMENT IS ILLEGAL
127.	F	UNEXPECTED /8/ ENCOUNTERED IN /7/ STATEMENT
128.	F	DELIMITER FOLLOWING 'FROM' ENTRY IN IMPLICIT STATEMENT IS ILLEGAL
129.	F	/8/ FOLLOWING RIGHT PAREN IN /7/ STATEMENT IS ILLEGAL
130.	F	/8/ IS ILLEGAL, EXPECTING TYPE FOR IMPLICIT STATEMENT
131.	F	TYPE SPECIFICATION IN IMPLICIT STATEMENT IS MISSING OR MISSPELLED
132.	F	NON-ALPHABETIC CHARACTER FOUND IN IMPLICIT STATEMENT - EXPECTING LETTER
133.	F	CHARACTER IN IMPLICIT LIST STARTING WITH /10/ ILLEGAL - EXPECTING LETTER

SECTION XIX - PARAMETER Statement

135.	F	= IS MISSING IN /7/ STATEMENT
136.	F	LEFT HAND SIDE OF = IN PARAMETER STATEMENT MUST BE A VARIABLE NAME
137.	F	FIRST VARIABLE NAME IN PARAMETER STATEMENT IS MISSING
452.	F	RIGHT OF EQUALS IN PARAMETER STATEMENT IS NOT A CONSTANT

SECTION XX - Return Statement

139.	F	RETURN STATEMENT IS ILLEGAL IN /9/
140.	F	ILLEGAL CHARACTER AFTER CONSTANT IN RETURN STATEMENT
142.	F	VARIABLE NAME IS >8 CHARACTERS OR CONSTANT IS TOO LARGE IN /7/ STATEMENT
143.	F	ILLEGAL CHARACTER AFTER CONSTANT IN /7/ STATEMENT

134.	F	CAN ONLY HAVE ARRAY /1/ WITH ADJUSTABLE DIMENSIONS IN SUBPROGRAM
138.	F	THE ADJUSTABLE DIMENSION /1/ MUST BE TYPE INTEGER
141.	F	CONSTANT IN ARRAY SUBSCRIPT MUST BE INTEGER
145.	F	CONSTANT FOR ARRAY DIMENSION IS NOT BETWEEN 1 AND 131071
146.	F	USE OF * FOR SIZE OPTION IS ILLEGAL IN /7/ STATEMENT
147.	F	/4/ IS ILLEGAL IN /7/ STATEMENT
148.	F	/8/ IS ILLEGAL AFTER SIZE OPTION IN /7/ STATEMENT
149.	F	THE /3/ /1/ CANNOT BE USED AS AN ADJUSTABLE DIMENSION
150.	F	MORE THAN 7 DIMENSIONS ARE SPECIFIED FOR /1/ IN /7/ STATEMENT
151.	F	/8/ IS ILLEGAL AFTER / IN /7/ STATEMENT
152.	F	/2/ IS ILLEGAL AFTER SPECIFYING DIMENSIONS FOR /1/
153.	F	/8/ IS ILLEGAL AFTER '/' IN /7/ STATEMENT
154.	F	FIRST VARIABLE NAME IN /7/ IS MISSING OR STARTS WITH A DIGIT
155.	F	/4/ IS ILLEGAL IN /7/ STATEMENT
156.	F	A DELIMITER IS MISSING BEFORE /1/ IN /7/ STATEMENT
157.	F	A DELIMITER IS MISSING OR /8/ IS ILLEGALLY USED IN /7/ STATEMENT
158.	F	/8/ IS ILLEGAL AFTER VARIABLE NAME IN /7/ STATEMENT
159.	F	/8/ IS ILLEGAL AFTER * IN /7/ STATEMENT
160.	F	/7/ STATEMENT INCOMPLETE
162.	F	A DIMENSION IS MISSING IN /7/ STATEMENT
163.	F	UNEXPECTED /8/ ON DIMENSION LIST OF /7/ STATEMENT
165.	F	TWO DELIMITERS IN A ROW IS ILLEGAL
166.	F	/8/ IS ILLEGAL AFTER DATA GENERATION OPTION IN /7/
167.	F	/8/ IS ILLEGAL AFTER A ',' , '/' OR SIZE OPTION
168.	F	NOT USED
169.	F	SIZE OPTION /1/ IS NOT A POSITIVE INTEGER CONSTANT
171.	F	INDEX INTO ARRAY /1/ IS > 131071
177.	F	DIMENSIONS ARE MISSING FOR /1/
192.	F	/3/ CANNOT BE TYPED

SECTION XXII - Expressions

172.	F	REAL PART OF COMPLEX EXPRESSION TOO LARGE
173.	F	REAL PART OF COMPLEX EXPRESSION TOO SMALL
174.	F	IMAGINARY PART OF COMPLEX EXPRESSION TOO LARGE
175.	F	IMAGINARY PART OF COMPLEX EXPRESSION TOO SMALL
176.	F	ILLEGAL COMBINATION OF TYPES IN RELATIONAL EXPRESSION

SECTION XXIII - Array Subscripts

178.	F	SUBSCRIPTS FOR /1/ MUST BE TYPE INTEGER
179.	F	SUBSCRIPTS FOR /1/ DO NOT MATCH DIMENSION SPECIFICATION

SECTION XXIV - Subscripted Identifier Use

182.	F	/1/ IS AN UNDIMENSIONED ARRAY OR AN INVALID FUNCTION
183.	F	SUBSCRIPTED ARRAY /1/ ILLEGAL AS A SUBSCRIPT IN /7/ STATEMENT
184.	F	FUNCTION /1/ ILLEGAL AS A SUBSCRIPT IN /7/ STATEMENT
186.	F	THE FUNCTION /1/ IS NOT ALLOWED IN DATA LISTS
187.	F	NUMBER OR ARGUMENTS IN /1/ ASF DOES NOT MATCH NUMBER IN DEFINITION
312.	F	ASF CALL HAS MORE THAN 100 ARGUMENTS
460.	F	ARRAY /1/ CANNOT BE USED AS A SCALAR WHEN A DO PARAMETER
461.	F	/1/ HAS PREVIOUSLY BEEN DIMENSIONED

SECTION XXV - Scalar Use

189.	F	/1/ MUST BE TYPED INTEGER BECAUSE IT IS USED AS A SUBSCRIPT IN /7/ STATEMENT
190.	F	THE IDENTIFIER /1/ MUST BE A PREVIOUSLY DEFINED PARAMETER SYMBOL
191.	F	THE VARIABLE /1/ HAS BEEN PREVIOUSLY USED AS /3/

SECTION XXVI - Expression Syntax

193.	W	ILLEGAL CHARACTER PRECEDING QUOTE IN /7/ STATEMENT
194.	F	= IS ILLEGAL IN ANY EXPRESSION
195.	F	/7/ STATEMENT INCOMPLETE
196.	F	UNEXPECTED /8/ ENCOUNTERED WHILE PROCESSING EXPRESSION

197.	F	UNEXPECTED /8/ IN EXPRESSION FOLLOWING RELATIONAL OPERATOR
198.	F	MISSING RIGHT PARENTHESIS OR COMMA
199.	F	UNEXPECTED /8/ FOLLOWING IMAGINARY COMPONENT OF COMPLEX CONSTANT
200.	F	UNEXPECTED /8/ IN SUBSCRIPT EXPRESSION
201.	F	UNEXPECTED /8/ AT BEGINNING OF FIRST SUBSCRIPT EXPRESSION
202.	F	UNEXPECTED /8/ -EXPECTING RIGHT PARENTHESIS OR COMMA
203.	F	UNEXPECTED /8/ IN ASF ARGUMENT-EXPECTING RIGHT PAREN OR COMMA
204.	F	UNEXPECTED /8/ WHILE PROCESSING ARGUMENTS OF CALL STATEMENT
205.	F	UNEXPECTED /8/ WHILE PROCESSING CALL STATEMENT-EXPECTING ')' OR ','
206.	F	UNEXPECTED /8/ WHILE PROCESSING DATA CONSTANT LIST
207.	F	UNEXPECTED /8/ WHILE PROCESSING THE SECOND OR THIRD INDEX PARAMETER
208.	F	UNEXPECTED /8/ WHILE PROCESSING FIRST INDEX PARAMETER OF [illegible] -EXPECTING ,
209.	F	UNEXPECTED /8/ WHILE PROCESSING LEFT HAND SIDE OF ASSIGNMENT STATEMENT-EXPECTING =
210.	F	/8/ ILLEGALLY USED IN /7/ STATEMENT EXPRESSION
211.	F	THE RELATIONAL OPERATOR /2/ IS ILLEGAL IN A RELATIONAL EXPRESSION
212.	F	ASF /1/ EXPANDS INCORRECTLY
341.	F	UNEXPECTED /8/ NEAR START OF I/O LIST
342.	F	UNEXPECTED /8/ IN ARRAY SUBSCRIPT-EXPECTING RIGHT PAREN OR COMMA

SECTION XXVII - Unary Operators

213.	F	/5/ 'S MAY NOT BE USED WITH .NOT. OPERATOR
214.	F	.NOT. MAY NOT BE USED WITH ARITHMETIC EXPRESSIONS
215.	F	TYPE /5/ MAY NOT BE USED WITH A UNARY + OPERATOR
216.	F	UNARY + MAY ONLY BE USED IN ARITHMETIC EXPRESSIONS
217.	F	UNEXPECTED /8/ TERMINATING CHARACTER STRING IN /3/ STATEMENT
218.	F	/5/ VARIABLE NAMES MAY NOT BE USED WITH UNARY -

SECTION XXVIII - Expression Semantics

219.	F	ILLEGAL COMBINATION OF LOGICAL, CHARACTER OR TYPELESS ENTITY IN EXPRESSION
220.	F	LOGICAL OPERATORS MUST BE USED WITH LOGICAL EXPRESSIONS OR VARIABLE NAMES

221.	F	CHARACTER, LOGICAL OR TYPELESS ILLEGAL IN EXPONENTIAL EXPRESSIONS
223.	F	ILLEGAL COMBINATION OF TYPES IN EXPONENTIAL EXPRESSION

SECTION XXIX - Constant Operations

225.	F	REAL CONSTANT IS TOO SMALL IN /7/ STATEMENT, ZERO ASSUMED
226.	F	REAL CONSTANT IS TOO LARGE IN /7/ STATEMENT
227.	F	NEGATIVE INTEGER CONSTANT IS TOO LARGE
228.	F	NEGATIVE REAL CONSTANT IS TOO LARGE
229.	F	NEGATIVE REAL CONSTANT IS TOO SMALL, ZERO ASSUMED
230.	F	NEGATIVE REAL PART OF A COMPLEX CONSTANT IS TOO LARGE
231.	F	NEGATIVE IMAGINARY PART OF A COMPLEX CONSTANT IS TOO LARGE
232.	F	NEGATIVE REAL PART OF A COMPLEX CONSTANT IS TOO SMALL, ZERO ASSUMED
233.	F	NEGATIVE IMAGINARY PART OF A COMPLEX CONSTANT IS TOO SMALL, ZERO ASSUMED
234.	F	COMPLEX CONSTANTS ARE ILLEGAL IN RELATIONAL EXPRESSIONS
235.	F	VALUE OF ARITHMETIC EXPRESSION WITH INTEGER CONSTANTS IS TOO LARGE
236.	F	ARITHMETIC EXPRESSION WITH REAL CONSTANTS IS TOO LARGE
237.	F	ARITHMETIC EXPRESSION WITH REAL CONSTANTS IS TOO SMALL, ZERO ASSUMED
238.	F	ARITHMETIC EXPRESSION WITH DOUBLE PRECISION CONSTANTS IS TOO LARGE
239.	F	ARITHMETIC EXPRESSION WITH DOUBLE PRECISION CONSTANTS IS TOO SMALL, ZERO ASSUMED
241.	F	0**0 IS ILLEGAL
242.	F	0**-J IS ILLEGAL
444.	F	INTEGER CONSTANT IS TOO LARGE - LARGEST INTEGER VALUE ASSUMED
449.	F	CHARACTER CONSTANT MAY NOT BE GREATER THAN 511 CHARACTERS

SECTION XXX - Constant List Processor

243.	F	/2/ IS ILLEGAL FOLLOWING / OR , IN DATA LIST FOR /7/ STATEMENT
244.	F	/8/ FOLLOWING / OR , IN DATA LIST FOR /7/ STATEMENT IS ILLEGAL
245.	F	/8/ FOLLOWING VARIABLE NAME OR CONSTANT IN DATA LIST IS ILLEGAL
246.	F	/8/ FOLLOWING VARIABLE NAME OR CONSTANT IN DATA LIST IS ILLEGAL
247.	F	REPEAT FACTOR IN /7/ STATEMENT MUST BE A POSITIVE INTEGER
248.	F	/2/ FOLLOWING * IN DATA LIST FOR /7/ STATEMENT IS ILLEGAL
249.	F	/8/ AFTER * IN DATA LIST FOR /7/ STATEMENT IS ILLEGAL

251.	F	THE IDENTIFIER /1/ IN DATA CONSTANT LIST IS NOT A CONSTANT

SECTION XXXI - Data List Constant

252.	F	COMPILER ERROR IN PROCESSING LABEL OR VARIABLE
254.	F	COMPLILER ERROR IN HANDLING ARGLIST
255.	F	MAY NOT REDEFINE /3/ /1/
256.	F	COMPILER ERROR IN PROCESSING ERROR
258.	F	COMPILER ERROR

SECTION XXXII - Initial Statement Analysis

259.	F	STATEMENT INCOMPLETE
260.	F	= FOLLOWING LEFT PARENTHESIS IS ILLEGAL
261.	F	DELIMITER MISSING
262.	F	CONSTANT FOLLOWED BY LEFT PAREN IS ILLEGAL
263.	F	UNEXPECTED /8/ FOLLOWING OPERATOR
264.	F	PARENTHESES DO NOT BALANCE IN AN ASF DEFINITION
265.	F	/8/ FOLLOWING RIGHT PAREN IS ILLEGAL
266.	F	ILLEGAL , IN ASF DEFINITION
267.	F	END OF STATEMENT OR = FOLLOWING , IS ILLEGAL
268.	F	PARENTHESES DO NOT BALANCE
269.	F	= OR ' IS USED ILLEGALLY IN STATEMENT

SECTION XXXIII - Identifier, Constant, and Label Formation

270.	F	VARIABLE IN STATEMENT HAS > 8 CHARACTERS
271.	F	ILLEGAL LETTER FOLLOWING DIGIT, EXPECTING 'D' OR 'E'
272.	F	MORE THAT 2 DIGITS IN THE EXPONENT IN /7/ STATEMENT
274.	F	LOGICAL OR RELATIONAL OPERATOR IS INCOMPLETE IN /7/ STATEMENT
275.	F	SOMETHING IS MISSING AFTER PERIOD IN /7/ STATEMENT
276.	F	LOGICAL CONSTANT IS INCOMPLETE IN /7/ STATEMENT
277.	F	CHARACTER CONSTANT IS INCOMPLETE IN /7/ STATEMENT
278.	F	EXPONENT INCOMPLETE IN /7/ STATEMENT
279.	F	NON-FORTRAN CHARACTER ILLEGAL IN STATEMENT

280.	F	$ IS NOT FOLLOWED BY A STATEMENT NUMBER OR SWITCH IN /7/ STATEMENT
281.	F	/4/ FOLLOWING $ IS ILLEGAL IN /7/ STATEMENT
283.	F	UNEXPECTED /4/ WHILE PROCESSING IDENTIFIER
284.	F	ILLEGAL RELATIONAL OR LOGICAL OPERATOR /10/ AFTER PERIOD
285.	F	/4/ IS ILLEGAL IN RELATIONAL OR LOGICAL OPERATOR OR LOGICAL CONSTANT
287.	F	/4/ IS ILLEGAL AFTER A LOGICAL OR CHARACTER CONSTANT IN /7/ STATEMENT
288.	F	/4/ IS ILLEGAL IN AN ARITHMETIC CONSTANT IN /7/ STATEMENT
289.	F	/4/ IS ILLEGAL IN A SWITCH NAME IN /7/ STATEMENT
290.	F	/4/ IS ILLEGAL IN STATEMENT NUMBER IN /7/ STATEMENT
291.	F	/2/ IS NOT A RELATIONAL OPERATOR IN /7/ STATEMENT
292.	F	RECEIVED A /8/ FOLLOWING BLOCKNAME INSTEAD OF A'/'
293.	F	/1/ IS USED AS A SWITCH IN /7/ STATEMENT AND IS NOT TYPED INTEGER
294.	F	/1/ MUST BE A SCALAR VARIABLE TO BE A SWITCH
295.	F	TRUNCATION IS REQUIRED FOR ARITHMETIC CONSTANT IN /7/ STATEMENT
296.	F	= IS ILLEGAL IN /7/ STATEMENT
297.	F	STATEMENT NUMBER MUST BE BETWEEN 1 AND 99999
313.	F	NON-FORTRAN CHARACTER ENCOUNTERED
317.	F	FIRST LINE NUMBER MAY NOT BE FOLLOWED BY AN &

SECTION XXXIV - General Statement Formation

298.	F	/4/ IS ILLEGAL AS THE FIRST CHARACTER OF A STATEMENT
299.	F	NON-FORTRAN CHARACTER IS ILLEGAL AS FIRST LETTER OF STATEMENT
300.	F	STATEMENT UNRECOGNIZABLE OR FIRST VARIABLE IS > 8 CHARACTERS
301.	F	NON-FORTRAN CHARACTER ENCOUNTERED NEAR START OF STATEMENT
302.	F	. OR ' IS USED ILLEGALLY NEAR START OF STATEMENT
303.	F	FIRST WORD OF THE STATEMENT IS UNRECOGNIZABLE OR THE FIRST DELIMITER IS ILLEGAL
304.	F	FIRST DELIMITER OF /7/ STATEMENT IS ILLEGAL
305.	F	FIRST WORD OF STATEMENT IS FOLLOWED BY AN ILLEGAL DELIMITER

SECTION XXXV - Statement Legality Checks

314.	F	/7/ STATEMENT IS ILLEGAL AS A 'DO' END
315.	F	EXECUTABLE STATEMENTS ARE ILLEGAL IN BLOCKDATA SUBROUTINES

316.	F	THE TRUTH CLAUSE OF A LOGICAL IF MUST BE EXECUTABLE

SECTION XXXVI - Statement Labels

319.	F	STATEMENT NUMBER /1/ HAS PREVIOUSLY BEEN DEFINED
320.	F	/1/ HAS BEEN REFERENCED AS A FORMAT LABEL
321.	F	THE STATEMENT WITH STATEMENT LABEL /1/ IS ILLEGAL AS THE END OF A DO LOOP
322.	F	ILLEGAL REFERENCE TO NON-EXECUTABLE STATEMENT AT STATEMENT NUMBER /1/
323.	F	STATEMENT NUMBER /1/ PREVIOUSLY REFERENCED AS A NON-EXECUTABLE STATEMENT
336.	F	/1/ PREVIOUSLY USED AS A REFERENCE TO A FORMAT STATEMENT

SECTION XXXVII - Identifier Semantics

324.	F	/1/ USED AS A DUMMY ARGUMENT IN AN ENTRY SUBROUTINE OR FUNCTION STATEMENT
325.	F	/1/ HAS BEEN USED AS AN ABNORMAL FUNCTION OR HAS APPEARED IN A COMMON STATEMENT
326.	F	/1/ HAS PREVIOUSLY APPEARED IN AN EQUIVALENCE STATEMENT
327.	F	/1/ HAS PREVIOUSLY BEEN USED AS A VARIABLE NAME, ASF NAME OR BLOCKNAME
328.	F	/1/ HAS PREVIOUSLY BEEN REFERENCED AS EXTERNAL TO THIS PROGRAM
329.	F	/1/ HAS PREVIOUSLY BEEN USED IN A CONFLICTING WAY
330.	F	/1/ HAS PREVIOUSLY BEEN USED AS A SCALAR VARIABLE
331.	F	/1/ HAS PREVIOUSLY BEEN USED AS THE NAME OF THIS FUNCTION
332.	F	/1/ HAS PREVIOUSLY BEEN USED AS THE NAME OF AN ASF
333.	F	/1/ HAS PREVIOUSLY BEEN DIMENSIONED
334.	F	/1/ INCOMPATIBLE WITH BEING ADJUSTABLY DIMENSIONED OR USED AS SUBROUTINE, ENTRY OR FUNCTION ARGUMENT
335.	F	/1/ HAS PREVIOUSLY APPEARED IN A TYPE STATEMENT
337.	F	/1/ HAS PREVIOUSLY BEEN USED AS A SUBROUTINE NAME IN A CALL STATEMENT
338.	F	/1/ PREVIOUSLY IN AN ABNORMAL STATEMENT OR IS USED AS AN ABNORMAL FUNCTION
339.	F	/1/ HAS PREVIOUSLY BEEN USED AS A NORMAL FUNCTION
457.	W	/3/ /1/ MAY NOT BE REDEFINED IN CALL OR ABNORMAL FUNCTION
458.	F	MAY NOT REDEFINE /3/ /1/
459.	F	MAY NOT REDEFINE /3/ /1/ BY USE AS A BUFFER TERM

144.	F	VARIABLE NAME AFTER /7/ MUST BE TYPE INTEGER OR CHARACTER
250.	W	'END' STATEMENT MISSING - SIMULATED
286.	W	'STOP' STATEMENT MISSING - SIMULATED
318.	F	FILE REFERENCE IN /7/ STATEMENT IS NOT AN INTEGER
340.	F	STATEMENT NUMBER /1/ HAS BEEN DEFINED AS A FORMAT LABEL
343.	F	UNEXPECTED /8/ WHILE PROCESSING I/O LIST-EXPECTING RIGHT PAREN OR COMMA
344.	F	PROGRAM ILLEGALLY STARTS WITH CONTINUATION CARD OR NFORM NOT SPECIFIED
345.	F	THE DO-END STATEMENT NUMBER /1/ IS OUT OF PLACE
346.	F	/1/ USED OR REFERENCED AS THE LABEL OF A FORMAT STATEMENT
347.	W	STATEMENT NUMBER /1/ REFERENCED AS AN EXECUTABLE STATEMENT
348.	F	SINGLE OR DOUBLE QUOTE MISPLACED IN 'H' FIELD
349.	F	CHARACTERS IN 'H' FIELD EXCEED COUNT
350.	F	TOO MANY DIGITS IN OCTAL CONSTANT
351.	F	ILLEGAL CHARACTER IN A DATA CONSTANT
352.	F	ILLEGAL CHARACTER IN OCTAL CONSTANT
353.	F	UNEXPECTED END OF STATEMENT WHILE PROCESSING /7/ STATEMENT
354.	F	COMPILER ERROR-UNUSUAL CONSTRUCT ON I/O LIST
355.	F	I/O LIST ELEMENT HAS REDUNDANT PARENTHESIS OR MISPLACED IMPLIED DO LIST
356.	F	= FOLLOWING IDENTIFIER IS ILLEGAL
357.	F	UNEXPECTED /2/ AT START OF I/O LIST ELEMENT
358.	F	UNEXPECTED /8/ TERMINATING /7/ STATEMENT
359.	F	TYPE OF COMPONENT OF COMPLEX CONSTANT IS NOT INTEGER, REAL, DOUBLE-PRECISION
360.	F	THE REAL AND IMAGINARY PARTS OF A COMPLEX CONSTANT MUST BE CONSTANTS
361.	F	COMPILER ERROR-IMPROPER PROCESSING OF LOGICAL CONSTANT
362.	F	COMPILER ERROR-IMPROPER PROCESSING OF CONSTANTS
363.	F	ILLEGAL COMPLEX OPERATION
364.	F	/1/ HAS PREVIOUSLY BEEN USED IN A CONFLICTING WAY
365.	F	UNEXPECTED /4/ WHEN EXPECTING OPTION IN /7/ STATEMENT

366.	F	OPTION BEGINNING WITH /10/ ILLEGAL OR MISSPELLED
367.	F	ILLEGAL CONSTRUCT IN ASSIGN STATEMENT
368.	F	ILLEGAL /8/ IN OPTION FIELD OF /7/ STATEMENT
369.	F	UNEXPECTED /8/ FOLLOWING FIRST PARAMETER OF /7/ STATEMENT
370.	F	UNEXPECTED /8/ IN /7/ LIST
371.	F	FORMAT LABEL IS NON INTEGER OR NOT BETWEEN 1 AND 9999
372.	F	STATEMENT NUMBER IS TOO LARGE IN /7/ STATEMENT
373.	F	/8/ IS ILLEGAL AS AN OPTION IN /7/ STATEMENT
374.	F	THERE ARE TWO ERR OPTIONS IN /7/ STATEMENT
375.	F	THERE ARE TWO END OPTIONS IN /7/ STATEMENT
376.	F	UNRECOGNIZABLE OPTION IN /7/ STATEMENT
377.	F	AN I/O LIST SHOULD NOT BE GIVEN WHEN A NAMELIST VARIABLE IS USED
378.	F	CANNOT PERFORM I/O-ROUTINE NOT AVAILABLE IN LIBRARY
379.	F	THE PARAMETER SYMBOL /1/ CANNOT BE TYPED
380.	F	FIRST PARAMETER OF /7/ MUST BE A VARIABLE NAME TYPED CHARACTER
381.	F	/11/ IS NOT TYPE CHARACTER
382.	F	/3/ /1/ IS ILLEGAL AS A BUFFER TERM
383.	F	/1/ IS NOT A LEGAL BUFFER VARIABLE
384.	F	THE /3/ /1/ IMPROPERLY USED AS A VARIABLE FORMAT LABEL OR NAMELIST NAME
385.	F	VARIABLE FORMAT LABEL /1/ NOT DIMENSIONED
386.	F	/1/ MUST BE A NAMELIST NAME OR A VARIABLE FORMAT LABEL
387.	F	NAMELIST NAME MISSING IN /7/ STATEMENT
388.	F	FIRST VARIABLE NAME IS NOT A LEGAL VARIABLE NAME
389.	F	THE FORMAT STATEMENT NUMBER IN /7/ STATEMENT HAS ILLEGAL CHARACTER
390.	F	INPUT LIST ITEM MAY NOT BE A CONSTANT, EXPRESSION OR UNDIMENSIONED ARRAY
391.	F	EXPRESSION OR CONSTANT ILLEGAL IN /7/ LIST
392.	F	THE /3/ /1/ IS ILLEGAL ON I/O LIST
393.	F	/1/ MUST BE DIMENSIONED IN ORDER TO APPEAR ON I/O LIST
394.	F	UNEXPECTED /8/ IN /7/ STATEMENT, EXPECTING A '/'
395.	F	UNEXPECTED /8/ AFTER '/' IN /7/ STATEMENT
396.	F	UNEXPECTED /8/ AFTER NAMELIST NAME-EXPECTING A '/'
397.	F	UNEXPECTED /8/ AFTER /1/ IN NAMELIST STATEMENT

398.	F	UNEXPECTED /8/ AFTER COMMA IN /7/ STATEMENT
399.	F	UNEXPECTED /2/ FOLLOWING DECLARED VARIABLE NAME IN /1/ STATEMENT
400.	F	LENGTH OF ARRAY /1/ IS > 131071
401.	F	= IS USED ILLEGALLY IN /7/ STATEMENT
402.	F	' IS USED ILLEGALLY IN /7/ STATEMENT
403.	F	PREVIOUS TYPING OF PARAMETER SYMBOL /1/ WILL BE IGNORED
404.	F	DO END STATEMENT NUMBER /1/ REFERENCES A NON-EXECUTABLE STATEMENT
405.	F	UNEXPECTED /8/ WHILE PROCESSING I/O LIST
406.	F	/1/ PREVIOUSLY USED AS THE STATEMENT NUMBER OF AN EXECUTABLE STATEMENT
407.	F	ARGUMENT OF /1/ IS NOT INTEGER, REAL, LOGICAL, TYPELESS OR CHARACTER
408.	F	MAY NOT REDEFINE INDEX VARIABLE /1/
409.	F	MAY NOT REDEFINE ADJUSTABLE DIMENSION /1/
410.	F	/1/ MUST BE TYPE INTEGER
411.	F	/3/ /1/ CANNOT BE TYPED
412.	F	ARRAY SUBSCRIPT REPRESENTED BY PARAMETER SYMBOL /1/ IS NOT INTEGER
413.	F	COMPILER ERROR - TWO CONSECUTIVE IDENTIFIERS IN /7/
414.	F	/8/ IS ILLEGAL AFTER A BLOCKNAME OR BLOCKNAME IS MISSING
415.	F	UNEXPECTED /8/ AFTER RIGHT PARENTHESIS IN /7/ STATEMENT
416.	F	CONSTANT FOR SIZE OPTION IS NOT AN INTEGER
417.	W	/1/ PREVIOUSLY TYPED IN A CONFLICTING WAY - NEW TYPE IGNORED
418.	F	ARRAY /1/ HAS LOGICAL OR CHARACTER SUBSCRIPT
419.	F	A DIVIDE CHECK FAULT OCCURRED WHILE PROCESSING CONSTANT EXPRESSION
420.	W	ARRAY /1/ IS BEING USED AS A SCALAR
421.	F	/1/ USED AS AN ADJUSTABLE DIMENSION OR ADJUSTABLY DIMENSIONED ARRAY
422.	F	/1/ HAS PREVIOUSLY BEEN USED TO REFERENCE AN EXECUTABLE STATEMENT
423.	F	THE FILE REFERENCE IN /1/ IS NOT AN INTEGER
424.	F	THE TRUTH CLAUSE OF THE LOGICAL IF ILLEGAL STARTS WITH THE CHARACTER /4/
425.	F	/1/ PREVIOUSLY USED AS A STATEMENT NUMBER OF A NON-EXECUTABLE STATEMENT
426.	F	/2/ USED ILLEGALLY IN EXPRESSION
427.	F	.NOT. OPERATOR USED ILLEGALLY IN EXPRESSION
428.	W	UNEXPECTED /8/ AFTER RIGHT PARENTHESIS IN /7/ STATEMENT
429.	F	DO END STATEMENT LABEL /1/ IS MISPLACED

430.	F	/1/ HAS PREVIOUSLY BEEN DEFINED
431.	F	ONLY ONE VARIABLE NAME IN EQUIVALENCE GROUP
432.	W	RETURN STATEMENT MISSING-SIMULATED
433.	F	/1/ HAS PREVIOUSLY BEEN DEFINED, POSSIBLY BY USE IN COMMON OR EQUIVALENCE
434.	F	SWITCH NAME IN GO TO STATEMENT IS ILLEGAL
435.	W	'X' FIELD IN FORMAT MUST HAVE A NUMERIC PREFIX
436.	W	THE LOGICAL IF STATEMENT DOES NOT HAVE AN ASSOCIATED TRUTH CLAUSE
437.	F	ILLEGAL COMBINATION WITH CHARACTER TYPE IN RELATIONAL EXPRESSION
438.	F	UNEXPECTED /8/ ENCOUNTERED IN /7/ STATEMENT
439.	F	THE 'H' FIELD COUNT OF A LITERAL MAY NOT BE ZERO
440.	F	FILE NUMBER MUST BE TYPE INTEGER IN /7/ STATEMENT
441.	F	RECEIVED CHARACTER OR LOGICAL TYPE IN EXPRESSION
442.	F	CONSTANT FOR DO PARAMETER IS NOT BETWEEN 1 AND 262143
451.	F	THE VARIABLE FORMAT /1/ IS NOT TYPE CHARACTER
453.	F	VARIABLE NAME OR PARAMETER CONSTANT MUST BE A POSITIVE INTEGER
455.		UNUSED
462.	F	UNEXPECTED /8/ FOLLOWING RIGHT PARENTHESIS IN I/O LIST ELEMENT
463.	W	THE INITIAL DO PARAMETER /1/ IS ALSO THE CURRENT DO INDEX
464.	F	THE DO TERMINAL PARAMETER OR STEP /1/ IS ALSO THE CURRENT DO INDEX
465.	W	SIZE OPTION FOR THE CHARACTER VARIABLE /1/ EXCEEDS 511, 511 ASSUMED
466.	W	THE CHARACTER '&' APPEARS ILLEGALLY, IT HAS BEEN IGNORED
467.	W	SIZE OPTION IN /7/ STATEMENT IS TOO LARGE - 511 ASSUMED
468.	W	NON-BLANK CHARACTERS IN COLUMNS 1-5 ILLEGAL - CHECK FORM/NFORM OPTION
469.	F	ASF LEFT OF EQUALS MUST EXPAND INTO AN IDENTIFIER OR ARRAY ELEMENT
470.	W	EQUALITY OR NON-EQUALITY COMPARISON MAY NOT BE MEANINGFUL IN LOGICAL IF EXPRESSIONS
471.	W	SIZE OPTION IS MISSING OR IS ZERO - /7/ ASSUMED
472.	F	THE FUNCTION OR SUBROUTINE /1/ MAY NOT HAVE AN ADJUSTABLE SIZE SPECIFICATION
473.	F	THE ADJUSTABLE SIZE OPTION /1/ IS ILLEGAL IN AN IMPLICIT STATEMENT
474.	F	THE VARIABLE MODIFIED BY THE ADJUSTABLE SIZE OPTION /1/ IS NOT TYPE CHARACTER
475.	F	THE ADJUSTABLE SIZE OPTION /1/ IS LEGAL ONLY IN A SUBPROGRAM
476.	F	THE /3/ /1/ CANNOT BE USED AS AN ADJUSTABLE SIZE OPTION

477.	W	A ZERO SIZE OPTION IS ILLEGAL - STANDARD DEFAULT ASSUMED
478.	W	PREVIOUS USAGE OF /1/ CONFLICTS WITH BEING TYPED - TYPING IGNORED
479.	F	THE GENERIC FUNCTION /1/ DOES NOT SUPPORT /5/ ARGUMENTS
480.	F	TRANSFORMATION OF GENERIC FUNCTION /1/ ILLEGAL DUE TO PRIOR USE OF /1/
481.	W	TYPING OF ASF /1/ WILL BE IGNORED IN LEFT OF EQUALS APPEARANCE.
482.	F	1ST AND 2ND ARGS OF FLD LEFT OF EQUAL MUST BE POSITIVE INTEGERS WHOSE SUM IS < = 36
483.	F	3RD ARG OF FLD LEFT OF EQUAL IS NOT SCALAR OR ARRAY OR IS TYPE LOGICAL
484.	F	TOO MANY ARGS FOR FLD LEFT OF EQUAL
485.	F	EXPRESSION RIGHT OF EQUAL CANT BE TYPE LOGICAL

PHASE2 ERROR MESSAGES

201.	W	EQUATING /1/ WITH /1/ IS REDUNDANT
202.	F	EQUATING /1/ WITH /1/ IS INCONSISTENT
203.	F	EQUATING /1/ WITH /1/ CAUSES CONTRADICTORY ALIGNMENT
204.	F	PROGRAM CONTAINS MORE THAN 510 SYMREFS
205.	W	/1/ AND /1/ IN COMMON HAVE REDUNDANCY IN EQUIVALENCE
206.	F	/1/ AND /1/ IN COMMON HAVE INCONSISTENCY IN EQUIVALENCE
207.	F	/1/ IN EQUIVALENCE EXTENDS COMMON BLOCK /1/ BELOW ORIGIN
208.	F	EQUATING /1/ WITH /1/ CAUSES COMMON EXTENSION
209.	W	STATEMENT CANNOT BE REACHED
210.	W	STATEMENT IS NEVER REFERENCED
211.	F	BRANCH TO NON-EXISTENT LABEL /1/
212.	F	ILLEGAL TRANSFER TO /1/ RANGE OF DO
213.	F	ILLEGAL TRANSFER INTO PARALLEL DO AT /1/
214.	F	ILLEGAL TRANSFER INTO DO AT /1/ FROM NESTED DO
215.	F	ILLEGAL TRANSFER INTO OF NESTED DO AT /1/ FROM DO
216.	F	PROGRAM CONTAINS MORE THAN 1023 EQUATED NAMES
217.	F	/1/ IS USED AS AN ARRAY IN AN EQUIVALENCE STATEMENT BUT IS NOT DIMENSIONED
218.	F	/1/ AND /1/ IN DIFFERENT COMMON BLOCKS ILLEGALLY EQUIVALENCED
219.	W	/1/ IS REFERENCED AS AN ARRAY BUT IS NEVER DEFINED

220.	F	THE SUBPROGRAM DUMMY ARGUMENT /1/ IS EQUIVALENCED
221.	F	THE ARRAY /1/ HAS ADJUSTABLE DIMENSIONS BUT IT IS NOT AN ARGUMENT TO THIS SUBPROGRAM
222.	W	THIS ILLEGAL TRANSFER TO /1/ INSIDE A DO CANNOT BE REACHED BY ANY TRANSFER OUT OF THE DO
223.	W	/1/ IS AN ILLEGALLY DEFINED DO PARAMETER IN THE EXTENDED RANGE OF THE DO AT LINE /6/
224.	W	THIS DO MAY BE ILLEGALLY EXTENDED. IT IS CONTAINED IN THE EXTENDED RANGE OF THE DO AT LINE /6/
225.	W	THIS DO IS CONTAINED IN EXTENDED RANGE OF DO AT LINE /6/ AND SOME CONTROL PARAMETERS ARE THE SAME
226.	W	THERE IS AN ILLEGAL TRANSFER TO /1/ INSIDE A DO FROM THE EXTENDED RANGE OF THE DO AT LINE /6/
227.	W	COMPILER TABLES EXCEEDED DURING EXTENDED-DO FLOW ANALYSIS. ANALYSIS TERMINATED. COMPILATION CONTINUES
228.	F	RETURN /6/. THERE ARE ONLY /6/ RETURNS SPECIFIED
229.	W	/1/ IN EQUIVALENCE REALIGNS COMMON BLOCK /1/
230.	W	/1/ APPEARS IN A LABEL ASSIGNMENT STATEMENT
231.	W	/1/ NEVER APPEARS IN A LABEL ASSIGNMENT STATEMENT
232.	F	/1/ COMMON BLOCK SIZE IS GREATER THAN 131071 WORDS
233.	W	/1/ IS NEVER REFERENCED IN /1/
234.	F	/1/ COMMON BLOCK SIZE IS ZERO

PHASE4 ERROR MESSAGES

401.	W	DATA STATEMENT: /1/ IS INCONSISTENT WITH /1/
402.	W	DATA STATEMENT: VARIABLE LIST EXCEEDS LITERAL LIST
403.	W	DATA STATEMENT: LITERAL LIST EXCEEDS VARIABLE LIST
404.	F	DATA STATEMENT: /1/ IS IN BLANK COMMON
405.	F	DATA STATEMENT: /1/ IS NOT IN BLOCK COMMON
406.	F	DATA STATEMENT: /1/ IS IN BLOCK COMMON
407.	F	DATA STATEMENT: TOO MANY IMPLIED, NESTED DO'S
408.	T	COMPILER ABORT IN PHASE4
409.	F	TOO MANY ARGS
410.	F	LOGICAL TABLE OVERFLOW
411.	T	COMPILER ABORT IN PHASE4B

412.	W	/1/ IS NOT DEFINED
413.	W	THE CHARACTER ASSIGNMENT INVOLVES A TRUNCATION OF THE RIGHT OF EQUALS
414.	F	THE ADJUSTABLE DIMENSION /1/ IS NOT AN ARGUMENT TO THIS SUBPROGRAM
415.	F	FILE B* IS NOT BIG ENOUGH
416.	F	FORMAT NUMBER /1/ DOES NOT EXIST
417.	W	THE CHARACTER CONSTANT /1/ WAS TRUNCATED TO FIT /1/
418.	W	FORMAT NUMBER /1/ IS MISSING; FORMAT (V) SIMULATED
419.	W	SUBSCRIPT FOR ARRAY /1/ IS OUTSIDE RANGE
420.	F	LABEL /1/ IS NOT DEFINED
421.	F	COMPILER ERROR. CODE GENERATED MAY BE INCORRECT
422.	F	THE ADJUSTABLY DIMENSIONED CHARACTER SCALAR /1/ IS NOT AN ARGUMENT TO THIS SUBPROGRAM

EXECUTIVE ERROR MESSAGES

001.	W	ALTER FILE ERROR FOLLOWING $ ALTER /6/, /6/
002.	W	ALTER FILE PROCESSING STOPS WITH $ ALTER /6/, /6/
003.	W	$ UPDATE CARD MUST BE FOLLOWED BY $ ALTER CARD
004.	F	COMDK SEQUENCE NUMBER IS /6/, IT SHOULD BE /6/
005.	F	CHECKSUM ERROR ON COMDK CARD NUMBER /6/
006.	F	PREMATURE EOF WHILE READING COMDK
007.	W	MEMORY EXPANDED, USE $ LIMITS OR CORE = OPTION FOR NEXT RUN
008.	T	NO MORE MEMORY EXPANSION ALLOWED
009.	F	NOT ENOUGH MEMORY AT PRESENT, TRY AGAIN
010.	F	MEMORY EXPANSION IMPOSSIBLE, USE $ LIMITS CARD OR CORE = OPTION FOR NEXT RUN
011.	W	OPTZ CAN OPERATE ON NO MORE THAN /6/ SUBEXPRESSIONS
012.	T	OPTZ CAN OPERATE ON NO MORE THAN /6/ SUBEXPRESSIONS
013.	T	PREMATURE EOF WHILE READING S*. UNABLE TO COMPILE.
014.	T	COMPILER ABORT IN PHASE /6/
015.	W	INCORRECT OPTION ON $ FORTRAN CARD

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 LIBARIES 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 > 43

FORTRAN File codes can range from 1-43.

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 TASK derail.

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

This message appears when the "PSTR" load map file is random; it must be linked.

SAVE FILE(S) CANNOT BE SPECIFIED

This message appears when H* save file appears to the left of the equals sign.

M6 - CALL/RSTR CHECKSUM

This message appears when the H* save file is not sufficiently large enough (in current size) to contain the bound link/overlay structure.

ADDRESS OUTSIZE OF FILE LIMITS

This message appears when 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 non-zero 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 n 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 - PCO.

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.

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 will print following source statement causing abort. Rerun with DUMP option and $ SYSOUT card for file code *F. Return dump to Honeywell Field Support - PCO.

Execution Aborts

LK	No $ ENTRY card for this link.
Q1	Logical Unit Table overflow.
Q2	Missing Logical Unit Table.
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 C

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 his 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 his $ LIMITS card should be changed for subsequent recompilations.

ERROR DETECTION and DIAGNOSTICS

All diagnostics are classified as either Warning, Fatal, or Terminated. Warning does not cause compilation to be terminated; Fatal causes execution to be deleted; and Terminated causes a termination of compilation. See Appendix B.

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 C-1 shows the storage layout for two typical low-loaded FORTRAN object programs.

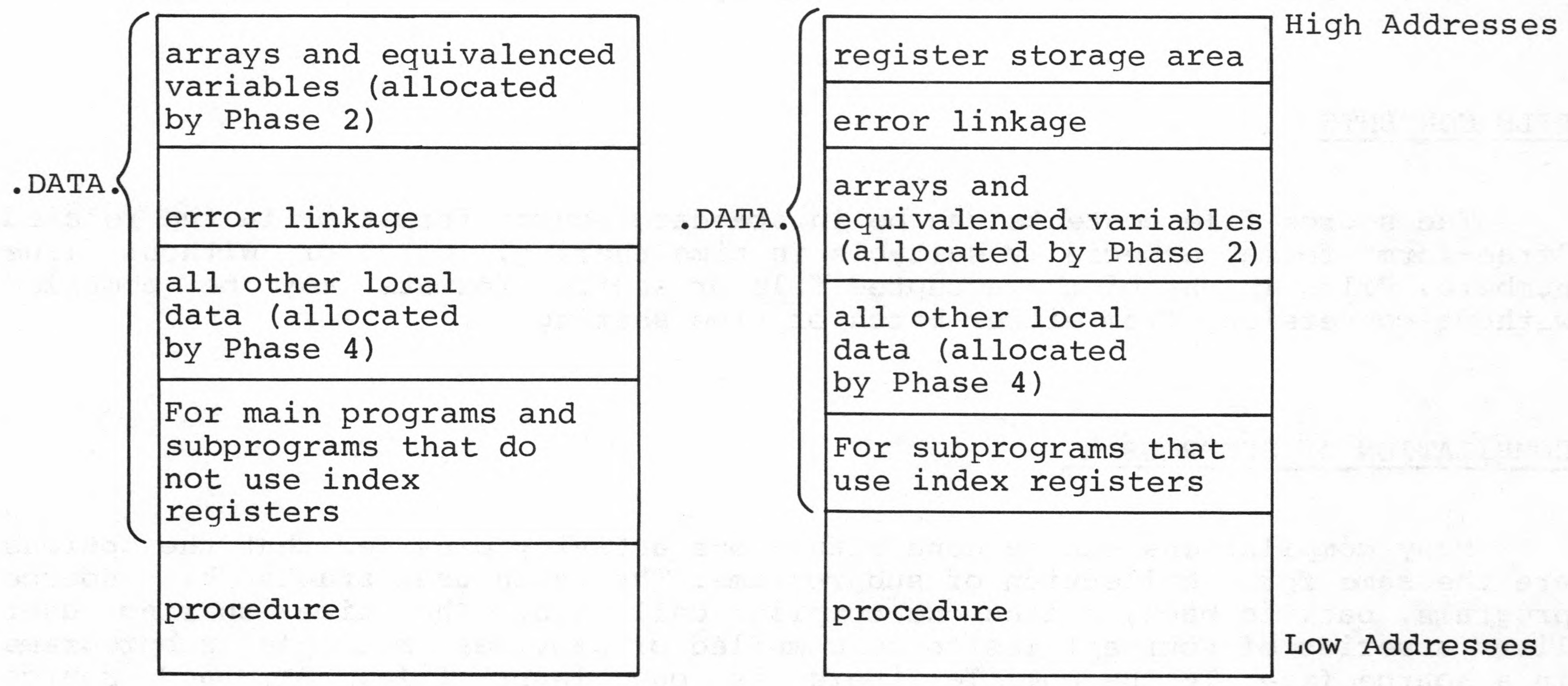

arrays and equivalenced variables (allocated by Phase 2)
.DATA.
error linkage
all other local data (allocated by Phase 4)
For main programs and subprograms that do not use index registers
procedure
High Addresses
register storage area
error linkage
.DATA.
arrays and equivalenced variables (allocated by Phase 2)
all other local data (allocated by Phase 4)
For subprograms that use index registers
procedure
Low Addresses

Figure C-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". 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) - that analysis generally done at the statement level.

2. Global Optimization (GO) - that 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 except for 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, 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 will increase compile time for most programs by a factor of about twenty percent. For many programs the specification of LSTOU will double 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 D

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 (including blanks).

Manual Mode

In manual mode, the user must provide (type) the line numbers for each line.

Automatic Mode

In automatic mode, the system provides the line numbers. They are printed as the build-mode request for input (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 for the user when he uses the command or response NEW. 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 (destructive) reinitialization of the current file.

Old File

An old file is a previously built and saved file which the user selects with the OLD command or response, naming the desired file. The old file is copied onto the current file where it is available to the user for processing or modification.

Current File

The current file is a temporary file assigned to the user, on which a new file is built or on which 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, if one existed, will not be altered until a RESAVE command naming the old file is executed.

Collector File

The collector file is a temporary file assigned to each user when he logs on. All input which is not a recognizable command is gathered onto this file -- for example, numbered statements. Then, when the file becomes full or a command is typed, depending upon the subsystem, 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. (The collector file is normally transparent to the user.)

Available File Table

An available file table (AFT) is provided for each time sharing system user. This table holds a finite number of file names (currently set at 20) which are entered in the AFT when the files are initially accessed (opened). The advantages of the AFT are:

1. Files requiring passwords or long catalog/file descriptions may be referended by file name alone, once they have been entered in the table.

2. Files used repeatedly remain readily available, thus reducing the overhead time and cost of accessing the file each time.

The following commands cause the named permanent files to be placed in the AFT.

LIST	<u>filename</u>(s)
OLD	<u>filename</u>(s)
SAVE/RESAVE	<u>filename</u>(s)
GET	<u>filename</u>(s)
PRINT	<u>filename</u>(s)
PERM	tempfile, <u>filename</u>

Because the AFT is of finite 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, the user must remove any unneeded files from the AFT in order to continue. The STATUS FILES command produces a listing of all of the user's files in the AFT. 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 not-busy.

DESCRIPTION OF FILES

File Specification

The specification of permanent files is provided for in the following formats:

1. filename — where the file name 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 not given (format a), the system will explicitly ask for the password.

If a required password is omitted in the string format (format c), a REQUEST DENIED message will be issued.

If the file was previously opened (e.g., with a GET), only the filename need be given regardless of its full description. If the requested file is not already open, it must emanate directly from the user's master catalog (quick-access type file) in order for formats a and b to be applicable.

Where desired-permissions and/or alternate-name are applicable, they are specified in the following format:

```
filedescr,permissions

                or

filedescr"altname",permissions
```

where:

<u>permissions</u> may be any one or combination of the following, separated by commas:

READ (or R)

WRITE (or W)

EXECUTE (or E)

APEND (or A)

<u>altname</u> may be a valid file name (one to eight characters), enclosed in double-quote signs.

Where a desired-permissions specification is applicable, a null <u>permissions</u> field implied 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 alternate-name, it must appear last in the specification string; e.g.:

filedescr,<u>permissions</u>(<u>i</u>,<u>j</u>)

or

filedescr "<u>altname</u>",<u>permissions</u>(<u>i</u>,<u>j</u>)

Examples:

```
OLD      FIL1$GOGO,R
SAVE     /CAT1CAT2$MAYI/FIL0$HERE
LIST     FILE2$HOHO(1,100)
PURGE    FIL3$ARIZ;FIL4;FIL5$SUN
GET      JJONES/DATACAT/BATCHWRLDFIL"INFILE"
```

<u>Categories of Files</u>

In the time sharing environment, distinctions are made between permanent files on two separate bases:

1. File-access type, which is a general time sharing, file-system-usage distinction and is not exclusive to FORTRAN; and,
2. File mode, which has primarily to do with the kinds of files produced under the FORTRAN system. (Both of these categories of distinctions apply to all files.)

FILE-ACCESS TYPES

There are three types of files, according to the method of creation and subsequent accessing of the file:

1. Quick-access files -- those permanent files that were automatically created (i.e., defined) by the system as a result of use of a SAVE filename or PERM tempfile; filename command as first reference to the named file. Quick-access files can also be created under ACCESS, provided 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 commands, and, in the case of data files (input or output), will be accessed automatically upon execution of a program reference to it -- i.e., it need not be pre-accessed by command.

   b. It has general READ permission assigned. It can be accessed with READ permission only by any other user who can describe it completely (creator's user ID/filename).

2. Quick-access files with password attached -- those permanent files that (normally) were automatically created (i.e., defined) by the system as a result of use of a SAVE filename$password command as first reference to a particular file name. This type of file is the same as the simple, quick-access type described above, except that it has the specified password attached. It has the following characteristics:

   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 will explicitly ask for the password. Also, in the case of data files, it will be accessed automatically upon execution of a program reference to it, but the system will explicitly ask for the password.

3. Nonquick-access files -- those permanent files that either do not "belong" to the user himself (i.e., where created by another user) or do not emanate directly from user's master catalog. In the latter case, the file is not completely described by user-ID and filename$password (intermediate catalogs exist), 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.

Note that quick-access files (with or without password) are only quick-access type relative to the file's creator. That is to say, a quick-access file for user A is by definition not a quick-access file for any other user.

NOTE: Once a type of file is initially accessed, whether by a GET or any other command, it can thenceforward be referred to simply by file name, unless explicitly removed from the AFT.

Three 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.
Binary	A linked (sequential) file of variable-length records in binary.
Random	A random file of fixed-length records in binary.

Source program files may be either time sharing format ASCII (type 5) or standard system format ASCII (type 6). ASCII input data files must be in type 5 format. Standard system format ASCII (type 6) may be converted to type 5 format using the time sharing command ASCASC. ASCII output data files written in FORTRAN are written as type 5 files. To use these output files in other time sharing systems, they must be converted to type 6, using the ASCASC command. Refer to the TSS General Information manual for a description of the ASCASC command.

FORTRAN object programs can produce linked binary-mode files as output.

FORTRAN object programs can produce random-mode files as output. Random is always the mode of object-program files themselves; i.e., compiler output is always in random mode.

Files created under execution and written must be converted to type 6 (with the ASCASC command) if the files are to be run under the Series 60 FORTRAN.

All files, of any mode, must be explicitly saved by use of the SAVE or PERM commands in order to be retained as permanent files. If the specified permanent file does not already exist, it will be implicitly created with the correct linked or random characteristic, as required by the file mode. (Linked is the standard, or default, type of file created.) If, however, the specified permanent file was explicitly created (predefined by the user, 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 true particularly for the file specified as 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 -- implicitly or explicitly -- as a random file. See the TSS General Information manual, for a description of the ACCESS subsystem.

ALTERNATE NAMING OF FILES

Permanent files may be temporarily renamed, with the altname capability of the time sharing command language, when necessary or desirable. Alternate naming is effective only during the terminal session in which the altname is assigned and the original file name in the file system is not altered. Two cases in which alternate naming would be required are as follows:

1. When a file created in the batch environment (e.g., a data file) with a name longer than eight characters is to be referred to by a FORTRAN program, it must be given an alternate name of eight characters or less.

2. When two or more files with identical file names are to be referred to in one time-sharing session, whether by commands or by the FORTRAN program, one or more must be differentiated by alternate names. (If the user is working only with his own quick-access files -- the "normal" case -- this problem does not arise.)

Alternate naming may conveniently be employed in the case where the file name used in FORTRAN program and the name of the actual permanent file to be referred to do not agree. Here the file may be given an altname, rather than changing the program reference. This case might frequently arise when working with common data files.

An alternate name can be assigned with the GET command, when "pre-accessing" data (or source) files, or can be assigned with extended forms of most other commands. Briefly, the syntax of alternate naming is:

filename"altname"

(or)

filename$password"altname"

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 IV.

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 will be 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 depressing 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 CALL request will cause a RECURSIVE CALL error message to be printed and the request will be ignored. Otherwise, the results of interpreting the CALL request will be unpredictable. The results will usually be either an abnormal program termination or, in time sharing, a loop that can be resolved only by entering a DONE, QUIT, or STOP request. (It may be necessary to depress the interrupt key to invoke FDEBUG to accept an input request.)

n CONTINUE

The CONTINUE request causes all debugging requests inserted at 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 label '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 will be interpreted will be that of the FORTRAN program unit that is 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 will be interpreted until another identifier request is encountered. When FDEBUG is invoked, the default identification in which subsequent requests will be interpreted will be that of the FORTRAN program unit that is 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 will be 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 will be 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 will be interpreted will be that of the FORTRAN program unit that is 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 will be 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 will be 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 depressed 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 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 Tables 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.

Table F-1. FDS Example in the Batch Mode

```
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
```

In the batch mode example described in Table 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.

Table 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 the Debugging Linked Overlay Programs paragraph 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.

Table F-2. FDS Example in the Batch Mode with Linked Overlays

```
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</u> <u>OF</u> <u>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
```

Table F-3. FDS Example in the Time Sharing Mode

```
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

```
RUN=(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

Table 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 will be displayed on the terminal and will include 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 Table F-4.

Table F-4. Example of a Symbolic Dump

```
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
```

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 will terminate 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 will terminate 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

Table 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 Table F-6. Each variable and array in Table F-6 is displayed by type.

Table F-5. Example of FDS Program and Subroutine used with FDUMP

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
```

Table F-5 (cont). Example of FDS Program and Subroutine used with FDUMP

Subroutine Referenced in Line 14

```
 1      SUBROUTINE CALC(X,Y,ANSW)
 2      X=X*Y+X
 3      ANSW=Y+Y*X
 4      INDX=INDX+1
 5       CALL FDUMP
 6      RETURN;END
```

Table F-6. Example of FDUMP Output

```
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
```

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

Table F-7 contains an example of the listing that is printed when the subprogram timing measurement system is invoked. Table 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.

Table F-7. Timing Measurement System Parameters

	NO. OF CALLS	TOT. MS. GLOBAL	GLOBAL % OF RUN	AVG. MS. PER CALL	TOT. MS. LOCAL	LOCAL % OF RUN	AVG. MS. 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				

Table F-8. Timing Measurement System in Time Sharing

```
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
```

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 will be called whenever the program terminates abnormally.

If an external subprogram is added to the wrapup list by including the CALL NTCALL statement, it will be called whenever the program terminates in a normal manner.

If a CALL NOCALL(subr) statement is included, all occurrences of 'subr' will be 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 will be 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 will be 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 will not be 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 will be 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 Table 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 will also be 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 will begin to process requests immediately following the most recently processed RETURN request. A CALL FCLOSE (fc) statement will 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 will be printed and the call or invocation will be 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 will be printed.

9. The FDEBUG module will 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 will begin 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 will be printed. A carriage return following a new identifier request will also produce a traceback.

13. When the wrapup list is inspected, a traceback will include 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 will lose control if the time sharing command !RUN=PROG is entered at the terminal, since the program named PROG would then be executed.

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 66)/6000 FORTRAN	ORDER NO.	DD02, REV. 0
		DATED	JANUARY 1975

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

CUT ALONG LI

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
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

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60(LEVEL 66)/6000 FORTRAN ADDENDUM B
ORDER NO.	DD02B, REV. 0
DATED	SEPTEMBER 1976

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 ______________________

____

CUT ALONG LINE

PLEASE FOLD AND TAPE —
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
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

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

20828, 2578, Printed in U.S.A.

DD02, Rev. 0