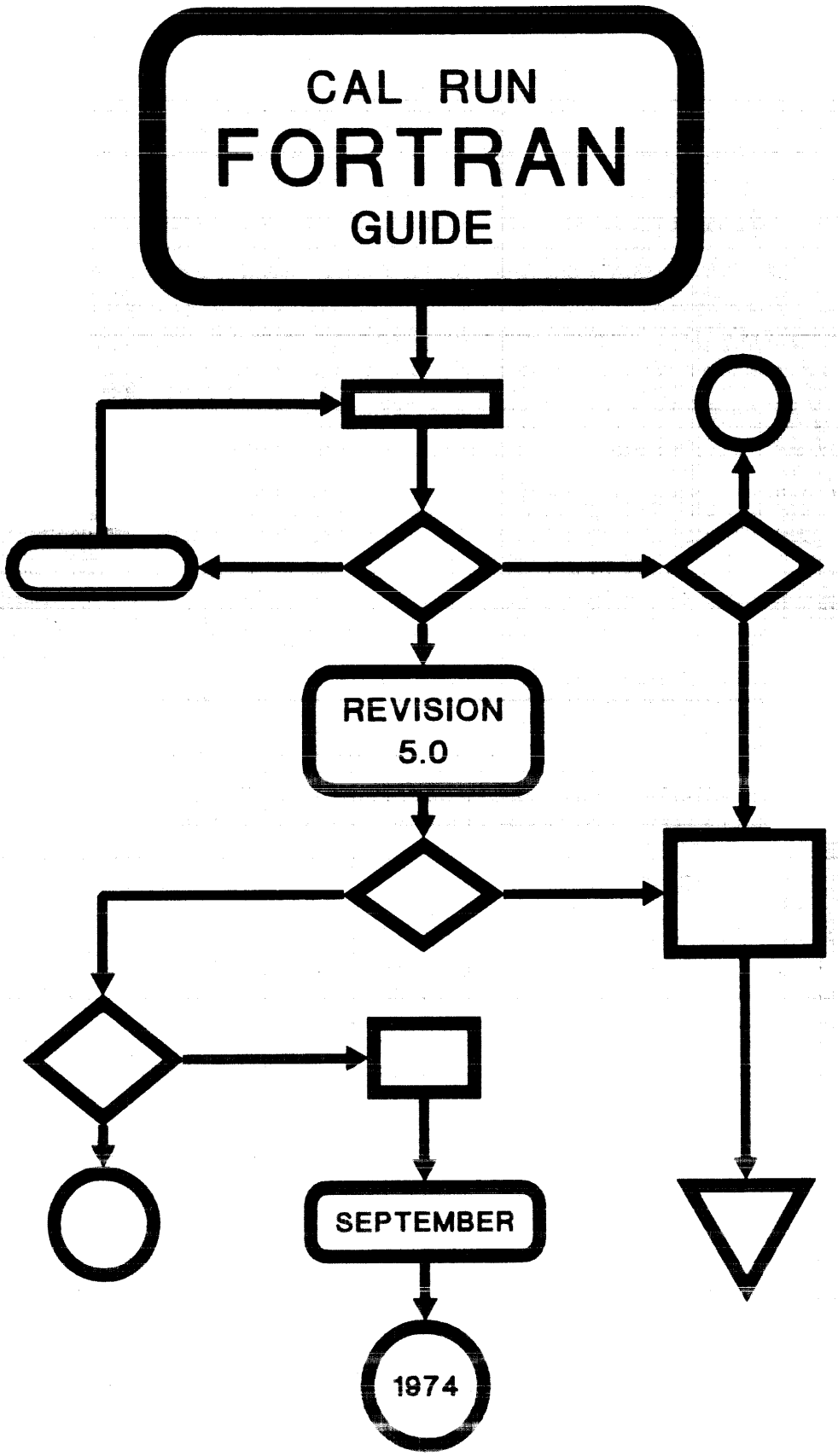
CAL RUN FORTRAN GUIDE



UNIVERSITY OF CALIFORNIA, COMPUTER CENTER, BERKELEY

CAL RUN Fortran Guide

University of California
Computer Center
Berkeley

Preface to Revision 4

This revision represents a change in the intended purpose of this document. Namely, material which is really relevant to the operating system instead of to Fortran, is to be published separately. Enough operating system information has been left here to enable simple Fortran runs to be set up and interpreted, but, for example, detailed descriptions of operating system control statements have been left out.

A second change in intention is that the Guide no longer purports to describe compilers other than RUN. The Guide to Computer Center Services contains pointers to documentation of other available compilers, such as FTN and AID.

As to the details of the changes, users familiar with the previous edition will find that chapters 1-10 and 12 contain only minor revisions and corrections. Chapter 11 has been greatly expanded to provide an index of all supported functions and subroutines available on the standard system library to users of RUN. Chapter 13 has been tailored more closely to the needs of Fortran users. Chapters 14-17 have been extensively reorganized into the new Chapters 14 and 15. The control statement descriptions and deck setups formerly contained in Chapter 15 are now published separately, and the information about FTN in Chapter 14 has been eliminated; otherwise, the former material has been retained, although in new locations. The old Appendix B (Fortran IV incompatabilities between the IBM 7094 and the CDC 6400) has been eliminated as have been all references to the 7094 and Fortran II. A new Appendix B lists known RUN bugs of a more or less permanent nature. Appendix C has been eliminated and the relevant material is now contained in Chapter 11.

Preface to Revision 5.0

This revision is intended to bring the manual up to date for fall 1974. Due to the exigencies of paper shortages, the use of colored sections has been eliminated.

Changes in this edition are numerous and distributed throughout the text. Most of the changes are by way of clarifications rather than changes of specification so that most of the information in the preceding edition is still applicable but must be used with caution. Substantive changes appear particularly in chapters 11 and 14.

TABLE OF CONTENTS

Introduction

Fortran in general and RUN Fortran in particular

The Fortran programming language is a widely available high-level computer language which is useful for expressing the solution to a wide variety of problems. A special translator program (known as a compiler) accepts, as input, programming statements written in Fortran and generates a corresponding sequence of instructions in the particular 'machine language' acceptable to the computer being used.

Ideally, Fortran exists independent of any particular computer or compiler. Unfortunately, many different Fortran compilers for different computers have been developed by different groups at different times, so that different versions of Fortran have inevitably evolved. Fortran programmers should be aware that the American National Standards Institute, (ANSI for short) has specified a 'standard' Fortran (see USA Standard FORTRAN X3.9-1966) which contains only those elements of the language common to most existing compilers. Thus, anyone developing a program which he may wish to operate in anything other than the current Computer Center environment should consider ANSI Fortran and use only ANSI standard features in his program. However, most compilers deviate in some respects from the standard with both extensions and restrictions being common, so that adherence to the ANSI standard does not eliminate, but only minimizes, the conversion problems encountered when going from one computer (or compiler or installation) to another.

This Guide describes the Fortran language as it is accepted by a particular compiler, the RUN Fortran Compiler.

Organization of this Guide

This Guide attempts to incorporate all information relevant to writing and running programs in RUN Fortran at the Computer Center. The approach is as tutorial as is practical, with many examples and much explanatory text. However, there are places where the desire for a complete reference document, combined with lack of time, have resulted in a presentation which is probably not accessible to the novice. Chapters 13 and 15 and the appendices are mostly of this type, and they are probably best regarded as reference material for the already knowledgeable. Even the more tutorial Chapters 1-12 do not constitute a text and a novice programmer may wish to consult one of the standard texts:

Organick, E.I., A FORTRAN IV Primer, Addison-Wesley, 1966

Lecht, C.P., The Programmer's FORTRAN II and IV, McGraw-Hill, 1966

Hull, T.E., Introduction to Computing, Prentice-Hall, 1966

Didday, R.L. and R.L. Page., FORTRAN for Humans, West, 1974.

The Guide is divided into several sections for easy reference. The first sections describe the RUN Fortran language and later sections give information relevant to actually running programs written in RUN Fortran.

Chapter 1-12 describe the RUN Fortran language and library.

Chapter 13 describes how information is structured in the 6400 computer by the operating system. Familiarity with this information is not necessary for most Fortran programs, but may be useful in cases where specialized input or output is performed and is essential for optimizing program performance when much I/O is to be done.

Chapter 14 describes how to set up an actual run on the computer and reproces and explains the output for a sample job. This information is mainly of use to the novice, but much useful information appears here which is not explicit anywhere else.

Chapter 15 gives miscellaneous information of a more or less esoteric nature about RUN Fortran. It also contains a food deal of 'folklore' which defied organization and for which no other place could be found.

Glossary gives definitions for some commonly used computer terms.

Appendices give details on how quantities are represented internally and how arithmetic is performed on the 6400 (Appendix A) and lists known bugs in the RUN compiler (Appendix B).

An Index is last.

Other documentation

A knowledge of material in the Guide to Computer Center Services is necessary for any utilization of the Center's facilities. It contains such useful items as where the computer is, how to get access to it, how much it costs, etc. It also contains a key to other Computer Center publications and facilities. These include other Fortran compilers, such as AID, Fortran Extended (FTN), and facilities of the operating system over and above those necessary for the straightforward use of RUN Fortran.

The other documents listed below also contain material of interest to users of RUN, but are not 'required reading':

CALIDOSCOPE Control Statements
OLDMSG
Computer Center Newsletters

These and other useful documents are available in the Computer Center Library, 216 Evans Hall.

Textual Notation

The following notation conventions have been adopted in the body of the text and in examples:

Numbers subscripted with 8, 10, and 2 indicate, respectively, octal, decimal, and binary representations, thus:

$$62_{10} = 76_8 = 111110_2 .$$

The symbol $\underline{0}$ (zero underlined) indicates, in internal character representation, a binary zero (all zero bits), not the BCD character zero.

Where necessary for the sake of clarity, the letter O is written as Ø to distinguish it from zero.

Where blanks need to be made explicit, they are indicated by the lower case letter b.

Updating

As errors are found and corrected or modifications are made to the system, the necessary changes are made to subsequent printings of the RUN Guide. Unless a major revision is made which obsoletes the previous edition, change lists or replacement pages will be available from the Computer Center Library. Points where revisions have been made are indicated by vertical bars in the margin of the page. Pages which have been revised since an edition was issued are identified with the date of the last revision.

1.1 WHAT IS FORTRAN?

FORTRAN is a problem-solving tool, a language which allows man to communicate with a computer without being forced to learn "machine language", the special code to which a computer responds. FORTRAN is a problem-oriented language; that is, it closely resembles the language naturally used in stating scientific and engineering problems. This allows the computer user to focus his attention on the definition of his problem and upon a plan for its solution.

FORTRAN is both a language and a compiler. As a language it is largely machine-independent. That is, it can be recognized by a large number of different computers regardless of their internal workings.

As a compiler, FORTRAN is a translator for a particular computer; that is, it is a computing procedure which will accept the FORTRAN language and convert it to a form usable directly by the computer. Because computers differ in internal characteristics, the interpretation of various specifications in the FORTRAN language may vary from computer to computer. Since a FORTRAN compiler is also a program, it may be designed to meet other criteria besides the language specifications, such as speed of translation, which may restrict the language. These reasons lead to differing versions of the FORTRAN language. This document describes a particular version of Fortran available at the Center, namely RUN Fortran.

One should remember that FORTRAN is not the natural language of the scientist or engineer, nor is it the natural language of the computer. Rather it is a compromise between the two, with a form composed of a set of symbols and rules that can be translated into the basic language of a computer. It is less ambiguous and more rigid in form than a natural language. It is also much smaller in scope, since the set of orders one is able to give a computer is small.

1.2 A FORTRAN SOURCE PROGRAM

The syntactic units of the Fortran language, the "sentences", are called statements. The rules for constructing these statements are described in Chapter 1-10.

A source program is a sequence of statements, arranged line after line, that specify a step-by-step procedure for solving a problem. This procedure is usually interpreted and acted on by the computer (i.e., executed) in sequential order, just as English sentences are read in sequence.

The types of statements which make up the program (and the language) can be grouped according to the components of the computer or compiler which they instruct.

The syntax ("grammatical rules") for each type of statement is described in the indicated chapters of this Guide:

Declarations instruct the compiler itself, providing information that enables it to translate other statements properly. They assign space in memory, structure groups of statements into subprogram units, and edit the information entering or leaving the computer (Chapters 5, 7, and 9).

The remaining statements describe a procedure the computer is to perform, and are called executable statements.

The arithmetic and logical statements define the basic computational steps for most programs and assign new values to variables in memory (Chapters 2-4).

Input-output (I/O) statements instruct one of the input-output devices attached to a computer. They usually call for data read into the computer from punched cards or for printing results (Chapter 10).

Execution Sequence Control Statements are used to alter the sequential mode of operation. Often values which have been computed are used in deciding which sequence of instructions is to be carried out (Chapter 6).

1.3 PROBLEM SOLUTION USING FORTRAN

To solve a problem on a computer by the use of the Fortran language is a multi-stage process. The first two stages are the problem definition and the analysis of a means of solving it. The computer must be provided with a step-by step method for reaching the answers. (A computer has no intuition; each decision or calculation must be described explicitly.)

In the third stage this procedure is written (coded) using the Fortran language statements. As the coding proceeds, more problem analysis may have to be done in order to clarify ambiguous areas.

The coding stage produces a source program which is usually presented to the computer in the form of punched cards or through a keyboard terminal.

In the fourth stage, the computer first executes the Fortran compiler with the source program as its input data. Next, the user's subprograms are combined with each other and with any required library subprograms by a program known as a loader. Finally, the computer may execute the procedure described by the source language statements, using any provided data. The calculated answers are usually printed by the computer.

In the final stage, the results must be scrutinized by the programmer. If errors are found, recoding and/or redefinition of the problem must be done. The execution and checking stages are known as testing. The program may be corrected and tested many times before a finished product is achieved.

A sample problem might be to find the real roots of the quadratic equation $ax^2+bx+c=0$.

Definition and Analysis: **The coefficients will be read from a data card. To solve the equation, the quadratic formula** $y = -b \pm \sqrt{b^2-4ac}$ **will be used. Since only real roots are to be found, a test will be included for a negative discriminant. The results will be printed along with the given coefficients.**

Coding: **The coding of the procedure could look as follows (the lines beginning with "C.." are explanatory comments):**

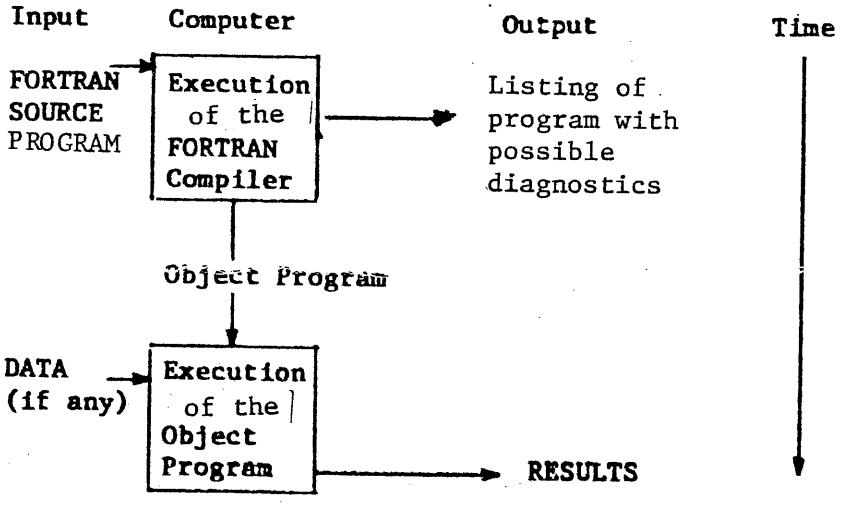
```
          PRØGRAM RØØTS(INPUT,ØUTPUT)
C..       CØMPUTE THE REAL RØØTS ØF THE EQUATIØN
C..             A*X**2+B*X+C=0
C..       PRINT HEADING
          PRINT 44
     44   FØRMAT(1H1,7X,1HA,16X,1HB,16X,1HC,17X,5HRØØT1,13X,5HRØØT2)
C..       READ DATA CARD
     10   READ 11,A,B,C
C..       CALCULATE DISCRIMINANT
          DISCRM=B**2-4.0*A*C
C..       TEST FØR NEGATIVE DISCRIMINANT
          IF(DISCRM)1,2,3
C..       NEGATIVE FØUND, PRINT MESSAGE
      1   PRINT 22, A,B,C
          STØP
C..       HAVE ZERØ DISCRIMINANT, RØOTS ARE EQUAL
      2   RØØT1=-B/(2.0*A)
          RØØT2 = RØØT1
          GØ TØ 4
C..       CALCULATE RØØTS
      3   RADICL=SQRT(DISCRM)
```

```
      RØØT1=(-B+RADICL)/(2.0*A)
      RØØT2=(-B-RADICL)/(2.0*A)
C..   PRINT RESULTS
    4 PRINT 33, A,B,C, RØØT1,RØØT2
      STØP
   11 FØRMAT(3F15.7)
   22 FØRMAT(1HO,3F17.7,*bRØØTSbAREbIMAGINARY*)
   33 FØRMAT(1HO,3F17.7,5X,2E18.10)
      END
```

Execution and Testing. The above program would be prepared as described in sections 1.5 and 1.6, and presented to the computer along with a set of operating system control statements. The program would be executed with various sets of data and the printed results would be checked against hand calculations. If there are any errors, changes in the source code or redefinition of the problem would be made.

1.4 COMPUTER PROCESSING OF A PROBLEM

The processing of the problem by the computer actually occurs in two parts. The first is compilation, the translation of the source program into an object program by the compiler. The second is the executing of this set of instructions by the computer to give the results specified by the source program and associated data. The stages are shown as blocks in the diagram below.



At each stage errors made by the programmer may be detected by the computer. If the individual statements, their order, or their reference to each other are inconsistent or have the wrong form (i.e., incorrect syntax), the compiler may issue a diagnostic report of the errors instead of allowing the program to continue to the next stage.

A program may still be faulty even when successfully compiled. The compiler can only check the syntax rules for FORTRAN statements described in this Guide. The meaning imparted by the programmer to these statements cannot be checked. Thus, a program which is syntactically correct may still fail to solve the given problem. It is not guaranteed that the compiler will detect all errors of syntax.

Errors occurring during execution are usually indicated by incorrect results or by error messages given in the printed output. But even if the results look correct, there is no guarantee that they are the desired results. They must be checked against known results, usually hand calculations, to insure that the program does indeed solve the stated problem.

1.5 FORTRAN CODING LINE

A FORTRAN coding line contains 80 columns* in which FORTRAN characters are written one per column. The three types of coding lines are listed below:

Line	Column	Content
Statement	1-5	statement number or blanks
	6	blank or zero
	7-72	FORTRAN statement
	73-80	identification field
Continuation	1-5	blank
	6	Any character other than blank or zero
	7-72	continued FORTRAN statement
	73-80	identification field
Comment	1	C
	2-80	comments

* Columns 73-80 are used only for labelling for the user's convenience. Their presence is optional and any Fortran source program may be entered through devices which allow only 72 characters per line.

1.5.1 Statement Number

Any statement may have an identifier called a statement number. A statement number is a string of 1 to 5 digits occupying any column positions 1 through 5. For usage, see Section 2.2.2.

1.5.2 Fortran Statement

The actual statement information is written in columns 7 through 72. Statements longer than 66 columns must be continued to the next line. Blanks in these columns are ignored by the Fortran compiler except in Hollerith fields or literal strings, and thus may be included wherever desired for clarity in reading a statement.

1.5.3 Statement Continuation

The first line of every statement must have a blank or zero in column 6. If statements occupy more than one line, all subsequent lines must have any characters other than blank or zero in column 6. Continuation lines may be separated by lines whose first 72 columns are blank. A statement may have up to 19 continuation lines.

1.5.4 Identification Field

Columns 73 through 80 are always ignored in the compilation process. Usually these columns are blank or contain sequencing information provided by the programmer, which acts as card identification when the program is to be punched on cards.

1.5.5 Comments

Each line of comment information is designated by a C in column 1. Comment information may be placed anywhere in the source program. It appears in the source program listing, but it is not translated into object code. The continuation character in column 6 is not applicable to comments.

1.6 INPUT MEDIA

It is necessary to transcribe the coded source program and data onto a medium which is easily interpretable by the computer. A detailed discussion of input and output media appears in Chapter 13.

1.6.1 Punched Cards

The most commonly-used medium is the punched card. Each coding line corresponds to one 80-column card; the terms "line" and "card" are often used interchangeably. A punch code is used in each column of the card to indicate the letter, digit, or special character which is represented by the column. This code is called Hollerith code and the information read into the computer from such cards is called display code. Both of these codes are detailed in Appendix A, Table H.

The format of a card used for statements in a source program is described in Section 1.5. When cards are being used for data, all 80 columns may be used.

1.6.2 Keyboard Terminals

The second most commonly-used input medium is a keyboard-actuated terminal. This is a device with a keyboard and a display mechanism connected to the computer by wire. The most common keyboard terminal is a teletype machine but other devices with displays similar to a television receiver are often used.

It is usual to restrict input and output to lines no longer than 72 characters when keyboard terminals are used.

1.6.3 Magnetic Tape

Magnetic tape is often used for the storage of large program or data files but not as an initial input medium for programs. See Chapter 13 for details of magnetic tape usage.

1.7 SYMBOL CONVENTIONS

Because of the similarity between symbols, certain handwriting conventions are established when coding a problem. Is are usually written as $\mathcal{I}$ and ones as / . Letter Os are often written with a slash, Ø. Zeros are seen as 0. Note that this convention (0 and Ø) is not universally adopted. Many documents exist which use the exact opposite convention. Beware! A preferred (and unambiguous) convention is to write a cursive letter $\mathcal{o}$ with the zero as 0.

Zs are usually written as Ƶ and twos as 2. A blank in a line of coding, especially in a Hollerith field, is represented by the symbol ƀ or b.

2.1 FORTRAN CHARACTER SET

Alphabetic:	A to Z	
Numeric:	0 to 9	
Special:	= equals	) right parenthesis
	+ plus	, commas
	- minus	. decimal point
	* asterisk	$ dollar sign
	/ slash	(space) blank
	(left parenthesis	

All characters appear internally in display code (Appendix A). A blank is ignored by the compiler except in Hollerith fields within DATA and FORMAT statements and in Hollerith constants; otherwise it may be used freely to improve program readability.

2.2 IDENTIFIERS

2.2.1 Symbolic Name

A symbolic name (also called an alphanumeric identifier) consists of one to seven* alphabetic or numeric characters beginning with a letter, with one exception. The combination of the letter Ø followed by 6 or more octal digits is recognized as an octal constant. Embedded blanks within identifiers are ignored.

Examples:

ALPHA	PEN IS UP (will be treated as PENISUP)
A1234	Ø123
HI THERE	Ø123456 (illegal as identifier)
Ø12KK3	M

Symbolic names are used as names of:

Formal parameters	Labelled common blocks
Variables	Filesets
Library subprograms	Function subprograms
Subroutine subprograms	Block data subprograms
Main programs	NAMELIST group names
Input/Output units	
Statement functions	

*The maximum allowed in standard Fortran is six characters. Seven character names should not be used in programswhich are also to be used on other Fortran systems.

2.2.2 Statement Identifiers

Statements are identified by unsigned numbers, 1-5 digits long, placed anywhere in columns 1-5 of the initial line of a statement. Blanks and leading zeros are ignored. Within one subprogram, statement numbers must be unique and can be any number from 1 to 99999. The values of the statement numbers in a subprogram do not affect the order in which statements are executed. Statement identifiers are optional for statements not referenced by other statements.

2.2.3 Data Types

Seven data types are used in Fortran: integer, octal,* real, double precision, complex, Hollerith, and logical. Complex and double precision data may be formed from real data. The computer word structure for each data type is listed in Appendix A. Both the range and precision of numeric data are system dependent. Both are greater in RUN Fortran than in most systems.

2.3 CONSTANTS

There are constants of each data type in Fortran. The type of a constant is determined by its form.

2.3.1 Integer Constants

An integer constant, N, is a string of up to 18 decimal digits in the range $-(2^{59} - 1) \leq N \leq (2^{59} - 1)$. The magnitude of the result of integer addition or subtraction must not exceed this. Subscript and DO-index calculations are limited to $2^{17} - 1$.

Examples:

63	3647631
247	464646464
314159265	574396517802457165

During execution, the maximum allowable magnitude when an integer is converted to real is $2^{48} - 1$. This maximum applies to the result or operands of integer multiplication or division. High order bits will be lost if a value is larger, but no error message is provided during execution. See Appendix A for range limits expressed in decimal.

2.3.2 Octal Constants

An octal constant consists of 6 to 20 octal digits preceded by the letter Ø or 1 to 20 octal digits suffixed with a B. The forms are:

$$Ø n_1 \ldots n_i$$

$$n_1 \ldots n_i B$$

---

* Octal is not a standard data type and should be avoided in programs which may be used in other Fortran environments.

Both forms of the constant are assigned logical mode.

If the constant exceeds 20 digits, or if a non-octal digit (8 or 9) appears, a compiler diagnostic is provided.

If there are less than 20 digits, they are right justified in the computer word. The high order bits are filled out with zeros.

Examples:

Ø00007777777700000000	2374216B
Ø7777700077777	777776B
Ø2323232323232323	777000777000777B
Ø000077	
Ø7777777777777700	

2.3.3 Real Constants

A real constant is represented by a string of digits; it may contain a decimal point, or an exponent representing a power of 10, or both. Real constants may be in the following forms:

n.n n. .n n.nE±s .nE±s n.E±s nE±s

where n is a string of digits, and s is the exponent to the base 10. The plus sign may be omitted if s is positive. The magnitude range of a non-zero constant is approximately 10^{-294} to 10^{+322} with approximately 15 significant digits. If the range is exceeded, a compiler diagnostic is provided. See Appendix A for exact range limits.

Examples:

3.E1 (means 3.0×10^1 : i.e., 30.)	
3.1415768	31.41592E-01
314.0749162	.31415E01
3.141592E+279	.31415E+01

2.3.4 Double Precision Constants

A double precision constant is a string of digits represented internally by two words. The forms are similar to real constants. The string is n; s is the exponent to the base 10.

.nD±s n.nD±s n.D±s nD±s

The D must always appear, but the plus sign may be omitted for positive s. The magnitude range of a nonzero constant is the same as for real constants, but with approximately 29 significant digits; if the range is exceeded, a compiler diagnostic is provided.

Examples:

3.1415927D	3141.593D3
3.1416D0	31416.D-04
3141.593D-03	

2.3.5 Complex Constants

A complex constant is represented by a pair of real constants separated by a comma and enclosed in parentheses (r_1,r_2); r_1 represents the real part of the complex number, r_2, the imaginary part. Either constant may be preceded by a minus sign.

If the real numbers comprising the constant exceed the allowed range, a compiler diagnostic is provided. A diagnostic also occurs when the pair contains integer constants, including (0,0).

Examples:

Fortran Representation	Complex Number
(1.,6.55)	1. + 655i
(15.,16.7)	15. + 16.7i
(-14.09,1.654E-4)	-14.09 + .0001654i
(0.,-1.)	0. - 1.0i

2.3.6 Hollerith Constants

A Hollerith constant is a string of display code characters* of the form nHf; n is an unsigned decimal integer representing the length of the field f . The maximum number of characters allowed in a Hollerith constant of H form depends upon its usage; n is limited to 10 characters when used in an expression. The limit of ten characters is a characteristic of this system. On other systems the limit may be different. In a properly formed DATA statement n is limited only by the number of characters than can be contained in up to 19 continuation lines. Blanks are significant in the field f . When n is not a multiple of 10, the last computed word is left justified with blank fill.

Alternate forms are nLf (left justified) and nRf (right justified) Hollerith constants with true zero fill (not the character zero) for incomplete words. These alternate forms are not defined in standard Fortran and should be avoided in programs to be used on other systems. The maximum number of

* Use of characters outside the Fortran character set (see section 2.1) must be tested for side effects in each context.

characters allowed for these forms in expressions is 10. If more than 10 characters are used in a DATA statement for such a constant, only the last word has the zero fill.

Hollerith constants may be used in an arithmetic replacement statement, such as I=5HABCDE. They are stored internally in display code. Hollerith data should never be stored in real varables; only integer variables should normally be used. Great care must be taken in comparing Hollerith data items because of the possibility of arithmetic overflow.

Examples:

Constants	Internal Form
6HCØGITØ	CØGITØbbbb
4HERGØ	ERGØbbbbbb
3HSUM	SUMbbbbbbb
5RSUMbb	00000SUMbb **
12HCØNTRØLDATA	CØNTRØLDAT Abbbbbbbbb
5LSUMbb	SUMbb00000 **
1H)	)bbbbbbbbb
3LbTT	bTT0000000 **

A Hollerith constant which is used as an actual parameter of a subroutine call or function reference has a word of all zeros following the last word of the constant. See: Hardware Representation of Data, Appendix A.

2.3.7 *Logical Constants*

Logical constants may be in the forms:

.TRUE.

.FALSE.

A false constant is stored internally as plus zero. A true constant is stored internally as minus zero (A word of all one bits).* These are the only proper logical values.

2.4 *VARIABLES*

Fortran recognizes simple and subscripted variables. A simple variable represents a single quantity; it references a storage location. The value specified by the identifier is always the current value stored in the location. A variable is identified by a symbolic name (Section 2.2.1).

---

* Note the difference between a logical constant and a logical variable, Section 2.4.5.

** 0 represents a true binary zero here, not the display code character for 0.

The type of a variable is determined in one of two ways:

EXPLICITLY — Variables may be declared a particular type with the type declarations. (Section 5.1)

IMPLICITLY — A variable not defined in a type declaration is assumed to be integer if the first character of its symbolic name is I, J, K, L, M, or N. All other variables not declared in a type declaration are assumed to be real.

Assuming no explicit typing affects these variables, we have:

INTEGER examples:

I15, JK26, KKK, NP326L, M

REAL examples:

TEMP, RØBIN, A55, R3P281

2.4.1 Integer Variables

Integer variables can be typed explicitly or implicitly and values may be in the range $-(2^{59} - 1) \leq I \leq (2^{59} - 1)$. The maximum allowable magnitude of an integer variable depends on usage. The magnitude of the result of conversion from integer to real, or of integer multiplication or division or an integer being printed may not exceed $2^{48} - 1$; the result of integer addition or subtraction can be as great as $2^{59} - 1$. Subscripts and DO-indexes are limited to $2^{17} - 1$. Each integer variable occupies one word in storage. See Appendix A for range limits expressed in decimal and internal formats.

Examples:

N	NEGATE
ITEM	K2S04
M58A	M58

2.4.2 Real Variables

Real variables may be typed explicitly or implicitly; a non-zero value must be in the approximate range $10^{-294} \leq |r| \leq 10^{+322}$ with approximately 15 significant digits. Each real variable is stored in 6000 Series floating-point format and occupies one word. See Appendix A for exact range limits and internal format.

Examples:

VECTOR	A62597	X
YBAR	BARMIN	X74A

The variable, r, may have any one of the following values:

$$-10^{+322} \leq r \leq -10^{-294}, \quad r = 0, \quad 10^{-294} \leq r \leq 10^{+322}$$

2.4.3 Double Precision Variables

Double precision variables must be typed explicitly by a type declaration. Each double precision variable occupies two words of storage and can assume values in the same range as real variables but with approximately 29 significant digits. See Appendix A for exact range limits and internal format.

2.4.4 Complex Variables

Complex variables must be explicitly typed by a type declaration. A complex variable occupies two words in storage (real part first). Each word contains a number in real format. The ordered pair of real values (C_1, C_2) represents the complex number: $C_1 + i\ C_2$.

2.4.5 Logical Variables

Logical variables must be typed explicitly by a type declaration. Each logical variable occupies one word of storage; it can assume the value true or false. A logical variable with a plus zero value (a binary 0) is false; any other value, including minus zero (a word of all one bits) is considered true.** (Caution: see Appendix A, Logical Operands). When a logical variable appears in an expression whose dominant mode is real, double, or complex, it is not converted prior to its use in the evaluation of the expression (as is the case with an integer variable).

2.5 SUBSCRIPTED VARIABLE

A subscripted variable may have one, two, or three subscripts enclosed in parentheses immediately following the variable name.* More than three subscripts produce a compiler diagnostic. The subscripts can be unparenthesized expressions in which operands are simple integer variables and integer constants and operators are addition, subtraction, multiplication, and division only. Subscripted variables may not appear in a subscript. If a program is to be used on other Fortran systems, subscript expressions should be limited to constants and linear functions of a single variable.

When a subscripted variable represents the entire array as in a DIMENSIØN statement, the subscripts are the dimensions of the array. When a subscripted variable references a single element in an array as in an expression, the subscripts describe the relative location of the element in the array. The number of subscripts should be the same as the number declared for the array.

---

* The use of the identifier, FØRMAT, for a dimensioned variable should be avoided because the compiler has difficulty distinguishings its use from a FØRMAT statement (Section 9.2).

** Thus, it is possible to have both a logical variable and its negation considered true if the variable does not contain a proper logical value, i.e., minus zero for true, plus zero for false.

Simple Variable	Subscripted Variable
FRAN	A(I,J)
P	B(I+2,J+3,2*K+1)
Z14	Q(14)
EVAL	STRING(3*K*ILIM+3)
I	GØING(5-I,50/J)

2.6 ARRAYS

An array is a block of successive storage locations which is used for the storage of all subscripted variables with a given name. The entire array may be referenced by the array name without subscripts (in I/O lists and subprogram references). Arrays may have one, two, or three dimensions; the array name and dimensions must be declared in a DIMENSIØN (Section 5.2), CØMMØN (Section 5.3), or type declaration (Section 5.1) prior to the first program reference to that array.

Each element in an array may be referenced by the array name with a subscript notation. Program execution errors (with or without an error message) may result if the value of a subscript is zero, negative, or larger than the corresponding dimension declared for the array. The maximum number of elements in an array is the product of the dimensions.

In discussing arrays the following definitions are used: The dimension of an array is the number of subscripts in its declaration. A column is a subset of the elements of a two-dimensional array which have the same value for the second subscript. Thus the first subscript is the row number and the second subscript is the column number within a two-dimensional array. A plane is the subset of the elements of a three-dimensional array which have a given value for the third subscript. Thus a column is effectively a one-dimensional array and a plane is effectively a two-dimensional array.

2.6.1 Array Structures

Arrays can be of three forms. A one-dimensional array has its elements stored in ascending memory locations; in the array declared as A(9) and stored beginning in location L in memory:

$A_1 \rightarrow L$	$A_4 \rightarrow L+3$	$A_7 \rightarrow L+6$
$A_2 \rightarrow L+1$	$A_5 \rightarrow L+4$	$A_8 \rightarrow L+7$
$A_3 \rightarrow L+2$	$A_6 \rightarrow L+5$	$A_9 \rightarrow L+8$

A two-dimensional array has its elements stored by columns in ascending locations; in the array declared as A(4,3):

$A_{11} \rightarrow L$	$A_{12} \rightarrow L+4$	$A_{13} \rightarrow L+8$
$A_{21} \rightarrow L+1$	$A_{22} \rightarrow L+5$	$A_{23} \rightarrow L+9$
$A_{31} \rightarrow L+2$	$A_{32} \rightarrow L+6$	$A_{33} \rightarrow L+10$
$A_{41} \rightarrow L+3$	$A_{42} \rightarrow L+7$	$A_{43} \rightarrow L+11$

A three-dimensional array has its elements stored by column, row, and finally plane; in the array declared as A(3,3,3):

$A_{111} \rightarrow L$	$A_{121} \rightarrow L+3$	. . .	$A_{133} \rightarrow L+24$
$A_{211} \rightarrow L+1$	$A_{221} \rightarrow L+4$	. . .	$A_{233} \rightarrow L+25$
$A_{311} \rightarrow L+2$	$A_{321} \rightarrow L+5$	. . .	$A_{333} \rightarrow L+26$

Thus, in order to step consecutively through the memory locations allocated to an array, we would begin with the element whose subscripts all equal one. We would then add one to the first subscript at each step until its maximum value (the first dimension) is reached. On the following step, we add one to the second subscript and reset the first to one. When the second reaches its maximum, we would similarly advance the third. When this algorithm would require advancing to a non-existent element, we are through.

The planes are stored in order, starting with the first, as follows:

A_{111}	A_{121}	A_{131}
A_{211}	A_{221}	A_{231}
A_{311}	A_{321}	A_{331}

A_{112}	A_{122}	A_{132}
A_{212}	A_{222}	A_{232}
A_{312}	A_{322}	A_{332}

A_{113}	A_{123}	A_{133}
A_{213}	A_{223}	A_{233}
A_{313}	A_{323}	A_{333}

Array allocation is discussed further under the CØMMØN and DIMENSIØN declarations. The memory location of an array element with respect to the first element is a function of the subscript values, the declared array dimensions, and the type of the array.

Given DIMENSIØN A(L,M,N), the locations of A(i,j,k), with respect to the first element A of the array, is given by the array element successor function*

$$A + (i-1 + L * (j-1 + M * (k-1))) * E .$$

The quantity enclosed by the outer parentheses is the evaluation of the subscript expression. E is the element length--the number of storage words required for each element of the array. For real, logical, and integer arrays, E = 1. For complex and double precision arrays, E = 2.

Example:

In an array defined by DIMENSIØN A(3,3,3), the location of A(2,2,3) with respect to A(1,1,1) is:

$$\text{Locn } A(2,2,3) = \text{Locn } A(1,1,1) + (2-1+3(1+3(2)))$$

$$= L + 22$$

The elements of a single-dimension array A may not be referred to as A(I,J,K) or A(I,J). Diagnostics occur if this is attempted.

* The array element successor function for the one and two dimensional arrays A(L) and A(L,M) are $A + (i-1) * E$ and $A + (i-1 + L * (j-1)) * E$ respectively, where the symbols are as above.

An expression is a constant, variable (simple or subscripted), function, or a combination of these separated by operators and parentheses. The four kinds of expressions in FORTRAN are: arithmetic and masking (Boolean) expressions, which have numerical values, and logical and relational expressions, which have truth values. Each type of expression is associated with a group of operators and operands.

3.1 ARITHMETIC EXPRESSIONS

An arithmetic expression can contain the following operators:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Operands may be:

Constants

Variables (simple or subscripted)

Evaluated functions

Expressions

Any unsigned constant, variable, or function is an arithmetic expression. If X is an expression, then (X) is an expression. If X and Y are expressions, then the following are expressions:

X + Y	X - Y
X * Y	X / Y
-X	X**Y

If op is one of the valid operators given above and X and Y are valid expressions, then X op op Y is never a valid expression.

Examples:

```
A
3.14159
B + 16.427
(XBAR+(B(I,J+I,K)/3))
-(C+DELTA*AERØ)
(B-SQRT(B**2-(4*A*C)))/(2.0*A)
GRØSS-(TAX*0.04)
(TEMP+V(M,MAXF(A,B))*Y**C)/(H-FACT(K+3))   (where V and
                                            MAXF are functions)
A+-B (erroneous expression - invalid sequence of operators)
A+(-B)
```

3.1.1 Arithmetic Evaluation

The hierarchy of arithmetic evaluation is:

**	exponentiation	class 1
/	division	class 2
*	multiplication	
+	addition	class 3
-	subtraction	

In an expression with no parentheses or within a pair of parentheses in which unlike classes of operators appear, evaluation proceeds in the above order. In expressions containing like classes of operators, evaluation proceeds from left to right. For example, A**B**C is evaluated as (A**B)**C.

Parenthetical and function expressions are evaluated first in a right-to-left scan of the entire statement. In parenthetical expressions within parenthetical expressions, evaluation begins with the innermost expression. Parenthetical expressions are evaluated as they are encountered in the right-to-left scanning process.

When writing an integer expression, it is important to remember not only the left-to-right evaluation process, but also that dividing an integer quantity by an integer quantity may yield a truncated result; thus 11/3=3. The expression I*J/K may yield a different result than the expression J/K*I. For example, 4*3/2 = 6 but 3/2*4 = 4.

Examples:

In the following examples, R indicates an intermediate result in evaluation:

A**B/C+D*E*F-G is evaluated:

$A^{**}B \rightarrow R_1$

$R_1/C \rightarrow R_2$

$D*E \rightarrow R_3$

$R_3*F \rightarrow R_4$

$R_4+R_2 \rightarrow R_5$

$R_5-G \rightarrow R_6$ evaluation completed

A**B/(C+D)*(E*F-G) is evaluated:

$E*F-G \rightarrow R_1$

$C+D \rightarrow R_2$

$A^{**}B \rightarrow R_3$

$R_3/R_2 \rightarrow R_4$

$R_4*R_1 \rightarrow R_5$ evaluation completed

H(13)+C(I,J+2)*(COS(Z))**2 is evaluated:

$COS(Z) \rightarrow R_1$

$R_1{}^{**}2 \rightarrow R_2$

$R_2*C(I,J+2) \rightarrow R_3$

$R_3+H(13) \rightarrow R_4$ evaluation completed

The following is an example of an expression with embedded parentheses.

A*(B+((C/D)-E)) is evaluated:

$C/D \rightarrow R_1$

$R_1-E \rightarrow R_2$

$R_2+B \rightarrow R_3$

$R_3*A \rightarrow R_4$ evaluation completed

(A*(SIN(X)+1.)-Z)/(C*(D-(E+F))) is evaluated:

$E+F \rightarrow R_1$

$D-R_1 \rightarrow R_2$

$C*R_2 \rightarrow R_3$

$SIN(X) \rightarrow R_4$

$R_4 + 1. \rightarrow R_5$

$R_5 * A \rightarrow R_6$

$R_6 - Z \rightarrow R_7$

$R_7/R_3 \rightarrow R_8$ evaluation completed

3.1.2 Mixed-Mode Arithmetic Expressions

Mixed-mode arithmetic with the exception of exponentiation is completely general*; however, most applications probably mix only the operand types: real and integer, real and double, or real and complex. The relationship between the mode of an evaluated expression and the types of operands it contains is established as follows below.

Order of dominance of the operand types within an expression from highest to lowest:

Complex

Double

Real

Integer

Logical

Arithmetic expressions, except exponentiation and functions, are evaluated by in-line arithmetic instructions.

The type of an evaluated arithmetic expression is the mode of the dominant operand type.

In expressions of the form A**B , the following rules apply:

If B is to be preceded by a unary minus operator, the form is A**(-B).

If A is preceded by a unary minus operator, it is equivalent to the form: -(A**B).

For the various operand types, the type relationships of A**B are:

Type of B

Type of A	I	R	D	C	L
I	I	n	n	n	n
R	R	R	D	n	n
D	D	D	D	n	n
C	C	n	n	n	n
L	I	n	n	n	n

mode of A**B

n indicates an invalid assumption

For example, if A is real and B is integer, the mode of A**B is real.

* Although this capability often provides more compact programs, its use can complicate the debugging process considerably.

Examples:

1) Given real A, B; integer I, J. The type of expression A*B-I+J is real because the dominant operand type is real.

   The expression is evaluated:

   Convert I to real

   Convert J to real

   $A*B \rightarrow R_1$ real

   $R_1-I \rightarrow R_2$ real

   $R_2+J \rightarrow R_3$ real

2) The use of parentheses can change the evaluation. A,B,I,J are defined as above. A*B-(I-J) is evaluated:

   $I-J \rightarrow R_1$ integer

   $A*B \rightarrow R_2$ real

   Convert R_1 to real

   $R_2-R_1 \rightarrow R_3$ real

3) Given complex C,D, real A,B. The type of the expression A*(C/D)+B is complex because the dominant operand type is complex. The expression is evaluated:

   $C/D \rightarrow R_1$ complex

   Convert A to complex

   $A*R_1 \rightarrow R_2$ complex

   Convert B to complex

   $R_2+B \rightarrow R_3$ complex

4) Consider the expression C/D+(A-B) where the operands are defined in 3 above. The expression is evaluated:

   $A-B \rightarrow R_1$ real

   $C/D \rightarrow R_2$ complex

   Convert R_1 to complex

   $R_1+R_2 \rightarrow R_3$ complex

5) Mixed-mode arithmetic with all types is illustrated by this example:

Given: the expression C*D+R/I-L

C	Complex
D	Double
R	Real
I	Integer
L	Logical

The dominant operand type in this expression is complex; therefore, the evaluated expression is complex.

Evaluation:

Round D to real and affix zero imaginary part.

Convert D to	complex
$C*D \rightarrow R_1$	complex
Convert R to	complex
Convert I to	complex
$R/I \rightarrow R_2$	complex
$R_2+R_1 \rightarrow R_3$	complex
$R_3-L \rightarrow R_4$	complex *

If the same expression is rewritten with parentheses as C*D+(R/I-L) the evaluation proceeds:

Convert I to	real
$R/I \rightarrow R_1$	real
$R_1-L \rightarrow R_2$	real *
Convert D to	complex
$C*D \rightarrow R_3$	complex
Convert R_2 to	complex
$R_2+R_3 \rightarrow R_4$	complex

3.2 RELATIONAL EXPRESSIONS

A relational expression has the form:

$$a_1 \text{ op } a_2$$

and its result has a logical value.

---

* Note that logical variables are not converted if used in an expression whose dominant mode is real, double, or complex.

The a's are arithmetic expressions; op is an operator belonging to the set:

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

A relation is true if a_1 and a_2 satisfy the relation specified by op; otherwise, it is false. A false relational expression is assigned the value plus zero; a true relational expression is assigned the value minus zero (all one bits).

Relations are evaluated by subtraction of the two expressions being related. Thus, for example, the evaluation of the relation p.EQ.q is equivalent to answering the question does p - q = 0? The difference is computed and tested for zero. If the difference is zero or minus zero, the answer is yes, and the relation is true. If the difference is not zero or minus zero, the relation is false.

The arithmetic values minus zero and plus zero are always considered equal in these comparisons.

Relational expressions are converted internally to arithmetic expressions according to the rules of mixed-mode arithmetic. These expressions are evaluated and compared with zero to determine the truth value of the corresponding relational expression. When complex (or double precision) expressions are tested for zero or minus zero, only the real part (or most significant part) is used in the comparison.

Take care when comparing non-integer values for equality. It is often more appropriate to test for an absolute difference less than some suitable tolerance value.

The operators of the above set cannot be used to relate logical variables and constants. The operators given under Logical Expressions, Section 3.3., should be used.

Order of dominance of the operand types within an expression is the order stated in mixed-mode arithmetic expressions. For example, relational expressions of the following forms are allowed:

```
I .LT. R

I .LT. D

I .LT. C
```

I is integer, R is real, D is double precision and C is complex. The I is converted appropriately before the comparison is made. If the expressions are not constants or single variables, they should be enclosed in parentheses to eliminate ambiguities in mode conversion.

Examples:

```
A  .GT.  16.              R(I).GE.R(I-1)
R-Q(I)*Z.LE.3.141592      K  .LT.  16
B-C .NE.  D+E             I  .EQ.  J(K)
                          (I).EQ.(J(K) )
```

3.3 LOGICAL EXPRESSIONS

A logical expression has the general form:

L_1 op L_2 op L_3 ...

The terms L_i are logical variables, logical constants, or relational expressions and op is either the logical operator .AND. indicating conjunction or .OR. indicating disjunction.

The logical operator .NØT. indicating negation appears in the form:

.NØT. L_1

If the value of the expression* is equal to plus zero, the logical expression has the value .FALSE. . All other values are considered true. Thus, it is possible to have both A and .NOT. A be true if A does not contain a proper logical value.

The hierarchy of logical operations is:

First	.NØT.
then	.AND.
then	.ØR.

A logical variable, logical constant, or a relational expression is, in itself, a logical expression. If L_1,L_2 are logical expressions, then the following are logical expressions:

.NØT.L_1

L_1.AND.L_2

L_1.ØR.L_2

If L is a logical expression, then (L) is a logical expression.

If L_1,L_2 are logical expressions and op is .AND. or .ØR., then L_1 op op L_2 is never legitimate.

---

* See Appendix A, Logical Operands.

.NØT. may appear in juxtaposition with .AND. or .ØR. only as follows:

L_1 .AND. .NØT. L_2

L_1 .ØR. .NØT. L_2

L_1 .AND.(.NØT. ...)

L_1 .ØR.(.NØT. ...)

.NØT. L_1 .ØR. ...

.NØT. L_1 .AND. ...

.NØT. may appear with itself only in the form .NØT.(.NØT. L); other combinations cause compilation diagnostics.

If L_1, L_2 are logical expressions, the logical operators are defined as follows:

.NØT.L_1	is true only if L_1 is false
L_1 .AND. L_2	is true only if L_1,L_2 are both true
L_1 .ØR. L_2	is true only is L_1 is true, L_2 is true, or both are true; otherwise it is false.

These relations are summarized in the table below, where T represents true and F is false.

L_1	L_2	L_1.AND.L_2	L_1.ØR.L_2	.NØT. L_1
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Examples:

1) B - C $\leq$ A $\leq$ B + C is written
B-C.LE.A.AND.A.LE.B+C

2) FICA greater than 176.0 and PAYNMB equal to 5889.0 is written
FICA.GT.176.0.AND.PAYNMB.EQ.5889.0

3) An expression equivalent to the logical relationship (P $\rightarrow$ Q) may be written as:
.NØT.P.ØR.Q

The masking expression is a generalized form of the logical expression in which the variables are of types other than logical.*

In a FORTRAN masking expression, 60-bit logical arithmetic is performed bit-by-bit on the operands within the expression. The operands may be any type variable, constant, or expression. No mode conversion is performed during evaluation. If the operand is complex, operations are performed on the real part. The masking operators are identical in appearance to the logical operators, their hierarchy is the same, and have the following definitions:

.NØT.	complement the operand
.AND.	form the bit-by-bit logical product of two operands
.ØR.	form the bit-by-bit logical sum of two operands.

The operations are described below:

p	v	p .AND. v	p .ØR. v	.NØT. p
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Let B_i be masking expressions, variables or constants of any type. The following are masking expressions:

.NØT. B_1

B_1 .AND. B_2

B_1 .ØR. B_2

If B is a masking expression, then (B) is a masking expression.

.NØT. may appear next to .AND. or .ØR. only as follows:

.AND. .NØT.

.ØR. .NØT.

.AND. (.NØT. ...)

.ØR. (.NØT. ...)

* Masking operations may also be performed by the AND, ØR, or CØMPL functions.

Masking expressions of the following forms are evaluated from left to right.

A .AND. B .AND. C . . .

A .ØR. B .ØR. C . . .

Arithmetic expressions contained in masking expressions should be enclosed in parentheses.

Examples:

A	**7777000000000000000**	**octal constant**
D	**0000000777777777777**	**octal constant**
B	**0000000000000001763**	**octal form of integer constant**
C	**2004500000000000000**	**octal form of real constant**

.NØT. A	is	0000777777777777777
A .AND. C	is	2004000000000000000
A .AND. .NØT. C	is	5773000000000000000
B .ØR. .NØT. D	is	7777777000000001763

4.1 ARITHMETIC REPLACEMENT

The general form of the arithmetic replacement statement is A = E , where E is an arithmetic expression and A is any variable name, simple or subscripted. The operator = means that A is replaced by the value of the evaluated expression, E, with conversion for mode if necessary.

Examples:

```
A = -A
B(N,4) = CALC(I+1)*BETA+2.3478
XTHETA=7.4*DELTA+(A(I,J,K)**BETA)
RESPSNE=SIN(ABAR(INV+2,JBAR)*ALPHA(J,KAPL(I)))
JMAX = 19
AREA = SIDE1 * SIDE2
PERIM = 2.*(SIDE1 + SIDE2)
C = (3.,1,)
```

4.2 MIXED-MODE REPLACEMENT

The type of an evaluated expression is determined by the type of the dominant operand. This, however, does not restrict the types that identifier A may assume. The following chart shows the A = E relationship for all the standard modes. The mode of A determines the mode of the statement.

When all the operands in the expression E are logical, the expression is evaluated as if all the logical operands were integers.

For example, if L_1, L_2, L_3, L_4 are logical variables, R is a real variable, and I is an integer variable, then $I = L_1 * L_2 + L_3 - L_4$ is evaluated as if the L_i were all integers and the resulting value is stored as an integer in I .

$R = L_1 * L_2 + L_3 - L_4$ is evaluated as stated above, but the result is converted to a real (a floating point quantity) before it is stored in R.

Warning: mixed-mode expressions are sometimes not compiled exactly as ANSI Fortran specifies. Dominant mode for sub-expression is determined by the compiler for each level of parenthesis grouping.

```
A = X .LT. Y .AND. I/J .EQ. K
```

is actually compiled by RUN as:

```
A = X .LT. Y .AND. FLOAT(I)/FLOAT(J) .EQ. FLOAT(K)
```

because the compiler determines the dominant mode as REAL, the integer division is not truncated as expected in standard Fortran.

To avoid this problem, and whenever in doubt, fully parenthesize the subexpressions in question. That is,

```
A = (X .LT. Y) .AND. (I/J .EQ. K)
```

will convince RUN to compile an ANSI standard interpretation without really altering the expression.

Type of A	Type of Expression E			
	Complex	Double Precision	Real	Integer
Complex	A = E	Set A = most significant half of E $A_{real} = E$ $A_{imag} = 0$	$A_{real} = E$ $A_{imag} = 0$	Convert E to Real $A_{real} = E$ $A_{imag} = 0$
Double Precision	$A = E_{real}$ less significant is set to zero	A = E	A = E less significant is set to zero	Convert E to Real A = E less significant is set to zero
Real	$A = E_{real}$	Set A = most significant half of E A = E	A = E	Convert E to Real A = E
Integer	Truncate E_{real} to Integer A = E	Truncate E to 48 bit integer A = E	Truncate E to Integer A = E	A = E
Logical	If $E_{real} \neq 0$, $A \neq 0$ If $E_{real} = 0$, A = 0	If $E \neq 0$, $A \neq 0$ If E = 0, A = 0	If $E \neq 0$, $A \neq 0$ If E = 0, A = 0	If $E \neq 0$, $A \neq 0$ If E = 0, A = 0

Examples:

Given:

C_i, A_1	Complex
D_i, A_2	Double
R_i, A_3	Real
I_i, A_4	Integer
L_i, A_5	Logical

1. $A_1 = C_1*C_2 - C_3/C_4$ (6.905, 15.393) = (4.4, 2.1)* (3.0, 2.0) - (3.3, 6.8)/(1.1, 3.4)

The expression is complex; the result of the expression is a two-word floating point quantity. A_1 is complex, and the result replaces the old value in A_1.

2. $A_3 = C_1$ 4.400 = (4.4,2.1)

The expression is complex. A_3 is real; therefore, the real part of C_1 replaces A_3.

3. $A_3 = C_1*(0.,-1.)$ 2.1000 = (4.4,2.1)*(0.,-1.)

The expression is complex. A_3 is real; the real part of the result of the complex multiplication replaces A_3.

4. $A_4 = R_1/R_2*(R_3-R_4)+I_1-(I_2*R_5)$ 13 = 8.4/4.2*(3.1-2.1)+14 -(1*2.3)

The expression is real. A_4 is integer; the result of the expression evaluation, a real, is converted to an integer replacing A_4.

5. $A_2 = D_1**2*(D_2+(D_3*D_4)) +(D_2*D_1*D_2)$ 4.96800000000000 = 2.0D**2*(3.2D+(4.1D*1.0D)) +(3.2D*2.0D*3.2D)

The expression is double precision. A_2 is double precision; the result of the expression evaluation, a double precision quantity, replaces A_2.

6. $A_5 = C_1*R_1-R_2+I_1$ 46.72=(4.4,2.1)*8.4-4.2+14

The expression is complex. Since A_5 is logical, the real part of the evaluated expression replaces A_5. If the real part is zero, zero replaces A_5.

4.3 LOGICAL REPLACEMENT

The general form of the logical replacement statement is L = E , where L is a logical variable and E may be a logical or relational expression. L is replaced by a minus zero (a word of all one bits) if the evaluated expression is true and by a plus zero (a binary 0) if the evaluated expression is false.

Examples:

```
LØGICAL A, B, C, D, E, LGA, LGB, LGC
REAL F, G, H
A = B .AND. C .AND. D
A = F .GT. G .ØR. F .GT. H
LGA = .NØT. LGB
LGC = E .ØR. LGC .ØR. LGB .ØR. LGA .ØR. (A .AND. B)
```

4.4 MASKING REPLACEMENT

The general form of the masking replacement statement is M = E. E is a masking expression, and M is a variable of any type except logical. No mode conversion is made during the replacement.

Examples:

```
INTEGER I,J,K,L,M,N(16)
REAL B,C,D,E,F(15)

N(2) = I .AND. J
B = C .AND. L
F(J) = I .ØR. .NØT. L .AND. F(J)
D = (B.LT. C) .AND. (C .LE. E) .AND. .NØT. I
```

Masking is used to form special bit patterns within a computer word and to manipulate bits and characters within a computer word.*

---

* A set of bit and character string manipulative functions is also available. The descriptions and binary decks for these subprograms may be obtained from the Computer Center Library.

TYPE DECLARATIONS AND STORAGE ALLOCATION 5

5.1 TYPE DECLARATION

The type declaration statements provide the compiler with information on the data type of variables and function values. They may also be used for array storage allocation.

Statement	Characteristics	
CØMPLEX list	2 words/element	Floating Point
DØUBLE PRECISIØN list or DØUBLE list	2 words/element	Floating Point
REAL list	1 word/element	Floating Point
INTEGER list	1 word/element	Integer
LØGICAL list	1 word/element	Logical

DØUBLE may replace DØUBLE PRECISIØN in any RUN Fortran statement in which the latter is allowed. This abbreviation is not acceptable to all Fortran systems.

The list is a string of names separated by commas; integer constant subscripts are permitted. For example:

A, B1, CAT, D36F, CAR (1, 2, 3)

Type declarations are non-executable and must precede the first reference to the typed variables or functions in a given subprogram. Type declarations should also precede the first executable statement in a given program; if not, a warning diagnostic will be given. Only one type may be declared for a name in any subprogram. If a name is typed more than once (even with the same type), the second and ensuing delcarations will result in warning diagnostics.

A name not declared in a type declaration is type INTEGER if the first letter of the name is I, J, K, L, M, or N; for any other letter, it is type REAL. (See Section 2.4.)

When subscripts appear in the list, the associated name is the name of an array, and the product of the subscripts determines the amount of storage to be reserved for that array. By this means, dimension (Section 5.2) and type information are given in the same statement. In this case no DIMENSIØN statement is needed; in fact, it is <u>not allowed</u>.

Examples:

```
CØMPLEX A412,DATA,DRIVE,IMPØRT
DØUBLE PRECISION PLATE,ALPHA(20,20),B2MAX,F60,JUNE
REAL I,J(20,50,2),LØGIC,MPH
INTEGER GAR(60),BETA,ZTANK,AGE,YEAR,DATE
LØGICAL DISJ,IMPL,STRØKE,EQUIV,MØDAL
DØUBLE RL,MASS(10,10)
```

5.2 DIMENSION DECLARATION

Storage is reserved for arrays by the non-executable statements DIMENSIØN, CØMMØN, or a type statement.

The standard form of the DIMENSIØN declaration is:

DIMENSIØN $v_1,v_2,\ldots,v_n$

The variable names v_i may have 1, 2, or 3 integer constant subscripts separated by commas, as in SPACE(5, 5, 5). Under certain conditions within subprograms only, the subscripts may be simple integer variables as well as constants (see Section 5.2.1).

The DIMENSIØN declaration is non-executable and it must precede the first reference to its declared arrays in a given subprogram. The DIMENSIØN statement should precede the first executable statement and will result in a warning diagnostic otherwise.

The number of computer words reserved for an array is determined by the product of the subscripts in its declaration and by the type of the variable.

A maximum of 131,071* elements may be reserved in any one array. If the maximum is exceeded, a diagnostic is provided.

Examples:

```
CØMPLEX ATØM
DIMENSIØN ATØM (10,20)
```

In the above declaration, the number of elements in the array ATØM is 200. Two words are used to contain a complex element; therefore, the number of computer words reserved is 400. This is also true for double precision arrays. For real, logical, and integer arrays, the number of words in an array equals the number of elements in the array. The same effect as the two statements above could be achieved by the statement:

```
CØMPLEX ATØM (10,20).
```

* This is the limit enforced by the compiler. The size of our computer limits storage even more. A total of 120000B (40960_{10}) words is the maximum central memory field length normally available.

An array may only be dimensioned in one declaration statement in any one subprogram.

Examples:

```
DIMENSIØN A(20,2,5)
DIMENSIØN MATRIX(10,10,10),VECTØR(100),ARRAY(16,27)
```

5.2.1 Variable Dimensions

When an array name and its dimensions appear as formal parameters in a function or subroutine, the dimensions may be assigned through the actual parameter list accompanying the function reference or subroutine call. The dimensions must agree with the array size specified in the calling program. (See Variable Dimensions in Subprograms, Section 7.11.)

Example:

```
SUBRØUTINE X(A,L,M)
DIMENSIØN A(L,10,M)
```

5.3 COMMON DECLARATION

The CØMMØN declaration provides up to 61 blocks of storage that may be shared with other subprograms. The declaration can reserve both blank and labeled blocks. Only labeled common blocks may be preset by DATA declarations (section 5.5). Data stored in labeled common blocks by the DATA declaration are available to any subprogram using the appropriate labeled block identifier.

The starting addresses for both blank and labeled blocks are indicated on the load map.

Areas of common information may be specified by the declaration:

CØMMØN/i_1/$list_1$/i_2/$list_2$ ...

where i_1,i_2, ... are labeled common block names. They have the form of symbolic names (Section 2.2.1).

Example:

```
CØMMØN/DATA1/A,B,C
```

A common statement without a block label, or with just blanks between the separating slashes is treated as a blank common assignment. Data may not be entered into blank common by a DATA declaration.

$List_i$ is a string of identifiers representing simple and subscripted variables. If a non-subscripted array name appears in the list, the dimensions must be defined by a type or DIMENSIØN declaration in that program. If an array is dimensioned in more than one declaration, a diagnostic is provided. The order of simple variables or array storage within a common block is determined by the sequence in which the variables appear in the CØMMØN statements.

CØMMØN is non-executable and can appear anywhere in the program but should precede the first executable statement or else a warning diagnostic is given. Any number of CØMMØN declarations may appear in a program subject to the restriction that no single subprogram may declare more than sixty labeled common blocks. If DIMENSIØN, CØMMØN, or type declarations appear together, the order is immaterial.

Since labeled common block names are used only within the compiler and loader, they may be used elsewhere in the program as other kinds of identifiers, except the subroutine name, or formal parameters in the same subroutine. A variable listed in one common block may not appear in another common block. (If it does, the variable is doubly defined.)

Examples:

```
CØMMØN A,B(10),C(5,5)
CØMMØN/ /E,F,G,H(10,5,2)        Use of blank common
CØMMØN/BLØCKA/A1(15),B1,C1/BLØCKD/DEL(5,2),ECHØ
CØMMØN/VECTØR/VECTØR(5),HECTØR,NECTØR
```

The length of a common block (in computer words) is determined from the number and type of the list variables. In the following statements, the length of common block A is 12 computer words. The origin of the common block is Q(1).

```
CØMMØN/A/Q(4),R(4),S(2)
REAL Q,R
CØMPLEX S
```

Block A

origin	Q(1)	
	Q(2)	
	Q(3)	
	Q(4)	
	R(1)	
	R(2)	
	R(3)	
	R(4)	
	S(1)	real part
	S(1)	imaginary part
	S(2)	real part
	S(2)	imaginary part

If a subprogram does not use all of the locations reserved in a common block, dummy variables may be necessary in the CØMMØN declaration to insure proper correspondence of common areas.

```
CØMMØN/SUM/A,B,C,D      (main program)
CØMMØN/SUM/E(3),D       (subprogram)
```

In the above example, only the variable D is used in the subprogram. The unused variable E is necessary to space over the area reserved by A, B, and C. This coding technique should not be used. It is much safer and easier to duplicate the necessary CØMMØN statements.

Each subprogram using a common block assigns the allocation of words in the block. The variables used within the block may differ as to name, type*, and number of elements; but the block name and total size must be the same for each occurrence of a particular labeled CØMMØN block.

Example:

```
PRØGRAM LINEAR (INPUT,ØUTPUT)
CØMPLEX C
CØMMØN/TEST/C(20)
:
:
```

The length of the block labeled TEST is 40 computer words.

---

* No type conversion is implied so only temporary storage should be treated this way.

The subprogram may rearrange the allocation of words as in:

```
SUBRØUTINE ØNE
CØMMØN/TEST/A(10),G(10),K(10)
CØMPLEX A
.
.
.
```

The length of TEST is 40 words. The first 10 elements (20 words) of the block represented by A are complex elements. Array G is the next 10 words, and array K is the last 10 words. Within the subprogram, elements of G are treated as floating point quantitites; elements of K are treated as integer quantities.

A labeled common block will be loaded with the first subprogram referencing that block. Therefore, the length of a labeled block must not be increased by subprograms subsequently loaded. Variable names used within the block may differ as shown above. Blank common normally follows in memory all user and library subprograms needed; its size may be different from one subprogram to another, and is determined by the largest blank common declaration encountered.

5.4 EQUIVALENCE DECLARATION

The EQUIVALENCE declaration permits variables to share locations in storage. The general form is:

EQUIVALENCE $(A_1,B_1,\ldots),(A_2,B_2,\ldots),\ldots$

$(A_1,B_1,\ldots)$ is an equivalence group of two or more simple or subscripted variable names.

EQUIVALENCE is most commonly used when two or more arrays can share the same storage locations.* The lengths need not be equal.

Example:

```
      DIMENSIØN A(10,10),I(90)
      EQUIVALENCE (A,I)
    5 READ 10, A
                .
                .
                .
    6 READ 20, I
```

* If two variables of different types are declared equivalent, no mode conversion is implied of the data stored in the shared locations.

The EQUIVALENCE declaration assigns the first element of array A and array I to the same storage location. The READ statement 5 stores the A array in consecutive locations. Before statement 6 is executed, all operations using A should be completed since the values of array I are read into the storage locations previously occupied by A.

Variables requiring two memory positions per element which appear in EQUIVALENCE statements must be declared to be CØMPLEX or DØUBLE PRECISION prior to their appearance in such statements.

Example:

```
CØMPLEX DAT,BAT
DIMENSIØN DAT(10,10),BAT(10,10),CAT(10,10)
DØUBLE PRECISIØN CAT
CØMMØN/IFAT/FAT(2,20 )
EQUIVALENCE (DAT(6,3),FAT(2,2) ),(CAT,BAT)
.
.
.
```

EQUIVALENCE is non-executable and can appear anywhere in the program or subprogram. However, if it appears after the first executable statement, a warning diagnostic is provided.

No element of the formal or dummy parameter list of a subprogram may appear in an EQUIVALENCE statement contained within the subprogram. Two variables which are both declared in CØMMØN statements may not be declared equivalent.

Any single or multiword variable may be made equivalent to any other single or multiword variable. The variables may be with or without subscripts.

The following example illustrates changes in block lengths caused by the EQUIVALENCE declaration.

Example:

Given: Arrays A and B and simple variable C. A and C are declared in a CØMMØN statement and B is not.

Declaration of /BLØCK1/	Allocation of /BLØCK1/		
CØMMØN/BLØCK1/A(4),C	origin	A(1)	
DIMENSIØN B(5)		A(2)	B(1)
EQUIVALENCE (A(3), B(2))		A(3)	B(2)
		A(4)	B(3)
		C	B(4)
			B(5)

Therefore /BLØCK1/ is six words long.

5.5 DATA DECLARATION

Initial values* may be assigned to program variables or labeled common variables** with the DATA declaration:

DATA $d_1,\ldots,d_n/a_1,k*a_2,\ldots,a_n/,d_1,\ldots,d_n/a_1,\ldots,a_n/,\ldots$

- d_i identifiers representing simple variables, array names, or variables with integer constant subscripts or integer variable subscripts (implied DØ-loop notation).
- a_i constants; they may be signed or unsigned.
- k integer constant repetition factor that causes the constant following the asterisk to be repeated k times. If k is non-integer, a compiler diagnostic occurs. Note that 5*-2.0 is legal in a DATA statements and specifies 5 repetitions of -2.0.

DATA is non-executable and can appear anywhere in the program or subprogram. When DATA appears with DIMENSIØN, CØMMØN, EQUIVALENCE, or a type declaration, the statement that dimensions any arrays used in the DATA statement must appear prior to the DATA statement. Variables in blank common or formal parameters may not be preset by a DATA declaration.

Single-subscript, DØ-loop-implying notation is permissible. This notation may be used for storing constant values in arrays. Multiple-subscripted arrays can be preset by listing each array element (specifying all subscripts with integer constants) or single-subscripted implied DØ-loop notation can be used to preset contiguous array elements which may be accessed according to the array element successor function (see section 2.6.1).

---

* If a variable is preset to a value by a DATA statement, executing a statement with a variable on the left-hand side of an equal sign or reading into the variable destroys that value; the original value will not be restored until the program is reloaded.

** Standard Fortran allows DATA statements which initialize variables in common storage to appear only in BLØCK DATA subprograms (see section 7.14).

Examples:

1. DIMENSIØN GIB(10)

   DATA (GIB(I), I=1, 10)/1.,2.,3.,7*4.32/

   Array GIB:

   1.
   2.
   3.
   4.32
   4.32
   4.32
   4.32
   4.32
   4.32
   4.32

2. DIMENSION TWØ(2,2)

   DATA TWØ(1,1),TWØ(1,2),TWØ(2,2),TWØ(2,1)/1.,2.,3.,4./

   Array TWØ:

TWØ(1,1)	1.
TWØ(2,1)	4.
TWØ(1,2)	2.
TWØ(2,2)	3.

3. DIMENSION SINGLE(3,2)

   DATA (SINGLE(I), I=1,6)/1.,2.,3.,1.,2.,3./

   Array SINGLE:

SINGLE(1,1)	1.
SINGLE(2,1)	2.
SINGLE(3,1)	3.
SINGLE(1,2)	1.
SINGLE(2,2)	2.
SINGLE(3,2)	3.

In the DATA declaration, the type of the constant stored is determined by the structure of the constant rather than by the variable type in the statement. In DATA A/2/, an integer 2 replaces A, not a real 2.0 as might be expected from the form of the symbolic name A.

There should be a one-one correspondence between the variable names and the list. This is particularly important in arrays in labeled common. For instance:

```
COMMON/BLK/A(3),B
DATA A/1.,2.,3.,4./
```

The constants 1.,2.,3., are stored in array locations A(1),A(2),A(3); the constant 4. is discarded, B is unmodified and a mysterious DATA RANGE diagnostic is issued. If this occurs unintentionally, errors may occur when B is referred to elsewhere in the program.

```
COMMON/TUP/C(3)
DATA C/1.,2./
```

The constants 1.,2. are stored in array locations C(1) and C(2); the content of C(3), is not defined.

When the number of list elements exceeds the range of the implied DO, the excess list elements are not stored, and a diagnostic is given.

```
DATA (A(I), I=1,5,1)/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./
```

The excess values 6. through 10. are discarded.

Examples:

1.
```
DATA LEDA, CASTOR, POLLUX/15,16.0,84.0/

    LEDA:       15
    CASTOR:     16.0
    POLLUX:     84.0
```

2.
```
DATA A(1,3)/16.239/

   Array A:

         A(1,3)    16.239
```

3.
```
DIMENSION B(10)
DATA B/77B, 64B, 3*5B, 5*200B/

   Array B;      77B
                 64B
                  5B
                  5B
                  5B
                200B
                200B
                200B
                200B
                200B
```

4. CØMMØN/HERA/C(4)

   DATA C/3.6,3*10.5/

Array C:	3.6
	10.5
	10.5
	10.5

5. CØMPLEX PRØTER (4)

   DATA PRØTER/4*(1.0,2.0)/

Array PRØTER:	1.0
	2.0
	1.0
	2.0
	1.0
	2.0
	1.0
	2.0

6. DIMENSIØN MESAGE (3)

   DATA MESAGE/9HSTATEMENT,2HIS,10HINCØMPLETE/

Array MESAGE:	STATEMENTb
	ISbbbbbbbb
	INCØMPLETE

7. DIMENSIØN MESSAGE (3)

   DATA MESSAGE/23HSTATEMENTbISbINCØMPLETE/

Array MESSAGE:	STATEMENTb
	ISbINCØMPL
	ETEbbbbbbb

Program execution normally proceeds from one executable statement to the next executable statement following it in the program. Execution sequence control statements can be used to alter this sequence or cause a number of iterations of a section of a program.

Control may be transferred to an executable statement only; a transfer to a non-executable statement usually results in a diagnostic message (but sometimes the diagnostic gets lost).

6.1 GO TO STATEMENTS

Program control is transferred to a statement other than the next statement in sequence by the GØ TØ statements.

6.1.1 Unconditional GO TO

GØ TØ n

where n is statement number. Executing this statement causes an unconditional transfer to the statement labeled n .

6.1.2 Assigned GO TO

GØ TØ m, $(n_1,n_2,\ldots,n_k)$

GØ TØ m

This statement acts as a many-branch GØ TØ; m is a simple integer variable assigned a statement label n_i in a preceding ASSIGN statement. The n_i are statement labels in the same subprogram. As shown, the parenthetical statement label list need not be present.

The comma after m is optional; however, when the list is omitted, the comma must be omitted. Although an integer variable in type, m must be defined by an ASSIGN statement and cannot be defined as the result of a computation. No compiler diagnostic is given if m is computed, but the object code is incorrect. If an assignment has not been made for an assigned GØ TØ statement before executing it, an error will occur.

6.1.3 ASSIGN STATEMENT

ASSIGN n TØ m

This statement is used to define m for use with an assigned GØ TØ statement; n is a statement label in the same subprogram as the ASSIGN, m is a simple integer variable.

Example:

```
ASSIGN 10 TØ LSWTCH
    .
    .
    .
GØ TØ LSWTCH, (5,10,15,20)
```

Control transfers to statement 10.

6.1.4
Computed GO TO

GØ TØ $(n_1,n_2,\ldots,n_m),i$

This statement acts as a many-branch GØ TØ; i must be preset or computed prior to its use in the GØ TØ.

The n_i are statement labels and i is a simple integer variable. i is not a statement number; rather, the value of i points to a position in the list of m statement labels. If $1 \leq i \leq m$, a transfer is made to statement n_i. All other values of i produce execution aborts.

The comma separating the statement number list and the index is optional.

Example:

```
N=3
 .
 .
 .
GO TØ (100,101,102,103),N
```

The third statement number, 102, identifies the next statement to be executed.

For proper operation, i must not be specified by an ASSIGN statement. No compilation diagnostic is provided for this error, but program execution will be incorrect.

Example:

```
ISWICH = 1
GØ TØ (10,20,30), ISWICH
    .
    .
    .
10  JSWICH = ISWICH + 1
GØ TØ (11,21,31), JSWICH
```

Control is transferred to statement 10 by ISWICH and then to statement 21 by JSWICH.

6.2 IF STATEMENTS

Program control is transferred to a statement depending upon the condition of the computed results of the IF statements. Evaluating the condition of an IF statement, such as IF(A.GT.B) requires arithmetic. If exceptional values (such as indefinite in a real comparison or Hollerith data in an integer comparison) occur, then improper results may be obtained.

6.2.1 Three-Branch Arithmetic IF

IF (A) n_1,n_2,n_3

A is an arithmetic expression, and the n_i are statement labels. This statement tests the evaluated expression A and transfers accordingly as follows:

A < 0	transfer to statement n_1
A = 0	transfer to statement n_2 (including A = -0)
A > 0	transfer to statement n_3

In the test for zero, +0 = -0. When the mode of the evaluated expression is complex, only the real part is tested for zero. Likewise, when the mode of the evaluated expression is double precision, only the upper part (most significant half) is used.

Examples:

```
IF(A*B-SIN (X) )    10,20,10
IF (I)5,6,7
IF (A/B**2)3, 6, 6
```

6.2.2 One-Branch Logical IF

IF (L) s

L is a logical expression* and s is any executable statement except another logical IF or a DO statement. If L is true (not plus zero), the statement s is executed. Program control then passes to the next sequential statement unless s is a transfer statement. In that case, control is transferred as indicated by the transfer statement. If L is false (plus zero), s is not executed and the statement immediately following the IF statement is executed. If the IF statement is the last statement of a DO-loop, the looping continues until the DO-loop is satisfied.

Note that a logical variable is in itself a logical expression. Other types of variables, if used alone, may be treated as logical expressions (without mode conversion), or may return a diagnostic and be rejected by the compiler. (See Logical Operands in Appendix A.)

Note that the use of .EQ. to test the value of a logical variable, i.e., IF(SW1 .EQ. .TRUE.), will always succeed since 0 ≡ -0 in such a test. Therefore, a statement of the form IF(SW1) should be used instead, since SW1 is logical already.

---

* Double precision arithmetic is not used with double precision variables in a logical IF statement. An arithmetic IF statement handles double precision variables correctly.

Examples:

```
IF(A.LE.2.5) A=2.0
IF(VALUE*4.73.GT.PRICE.ØR.VALUE.LT.150.0)BUY=.TRUE.
IF(P.AND.Q)GØ TØ 427
```

6.2.3 Two-Branch Logical IF

IF (L) n_1,n_2

L is a logical expression; n_i are statement labels.

The evaluated expression is tested for true (not plus zero) or false (plus zero) condition. If L is true, transfer is to statement n_1. If L is false, transfer is to statement n_2.

Examples:

```
IF(K)5,6
IF(K.EQ.100)70,60
IF(IJUMP.LT.K)10,11
```

6.3 DO STATEMENT

DØ n i = m_1,m_2,m_3 or DØ n i = m_1,m_2

This statement makes it convenient to repeat groups of statements and to change the value of an integer variable during the repetition. The group of statements beginning with the DØ statement and ending with a statement with label n is called the DO loop. i is the index variable (simple integer). The values of i and the number of times the DO loop is executed are determined by the indexing parameters m_1, m_2, and m_3. They may be unsigned integer constants or simple integer variables. The initial value assigned to i is m_1; m_2 is the limit assigned to i , and m_3 is the amount added to i after each time the DØ loop is executed. If m_3 does not appear, it is assumed to have the value 1.

Statement n may not be an arithmetic IF, a RETURN, a STOP, a PAUSE, a two-branch logical IF, a GØ TØ, another DØ, a logical IF containing any of the preceding forms, nor a non-executable statement. It need not be a CØNTINUE statement.

The indexing parameters m_1,m_2,m_3 are either unsigned integer constants or simple integer variables. Subscripted variables and negative or zero integer constants cause a diagnostic.

The indexing parameters m_1 and m_2, if variable, may assume positive or negative values or zero, but the latter two may cause erroneous results.

If the values of i, m_1, m_2, and m_3 are changed during the execution of the DØ-loop, this may produce undefined results. If it is necessary to change these parameters during an iteration, the loop should be coded using an IF statement rather than a DØ.

6.3.1 DØ-Loop Execution

The initial value of i, m_1, is increased by m_3 and compared with m_2 after executing the DØ loop once, and if i does not exceed m_2, the loop is executed a second time. Then, i is again increased by m_3 and again compared with m_2; this process continues until i exceeds m_2. Control then passes to the statement immediately following statement n, and the DØ loop is said to be 'satisfied'.

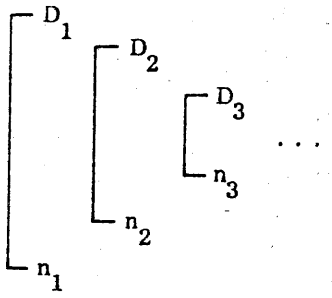
Should m_1 exceed m_2 on the initial entry to the loop, the loop is executed once (with i set to m_1) and the iteration ceases. If m_1 and m_2 are constants, this condition may produce an error diagnostic during compilation. When the DØ is satisfied, the index variable i is no longer well defined. If a transfer out of the DØ loop occurs before the DØ is satisfied, the value of i is preserved and may be used in subsequent statements.

DØ loop arithmetic is performed with 18-bit arithmetic, and the values involved must not at any time exceed 131071 (2^{17}-1) or else erroneous results may occur, e.g., the DØ may loop indefinitely.

6.3.2 DØ Nests

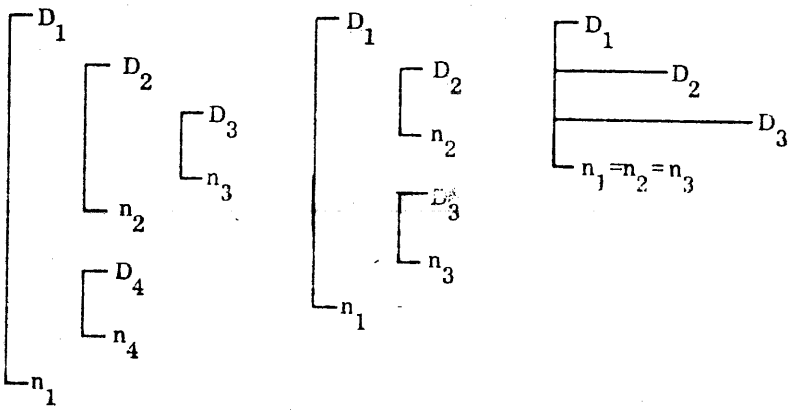
When a DØ loop contains another DØ loop, the grouping is called a DØ nest. Nesting may be to any level. Each loop must have a different index variable. The last statement of a nested DØ must either be the same as the last statement of the outer DØ loop or occur before it.

If $D_1, D_2, \ldots D_m$ represent DØ statements where the subscripts indicate that D_1 appears before D_2 appears before D_3 and $n_1, n_2, \ldots, n_m$ represent the corresponding limits of the D_i, then n_m must appear at or before n_{m-1}.



Examples:

DØ loops may be nested in common with other DØ loops:



```
      DØ 1 I=1,10,2
          .
          .
          .
      DØ 2 J=1,5
          .
          .
          .
      DØ 3 K=2,8
          .
          .
          .
3     CØNTINUE
          .
          .
          .
2     CØNTINUE
          .
          .
          .
      DØ 4 L=1,3
          .
          .
          .
4     CØNTINUE
          .
          .
          .
1     CØNTINUE
```

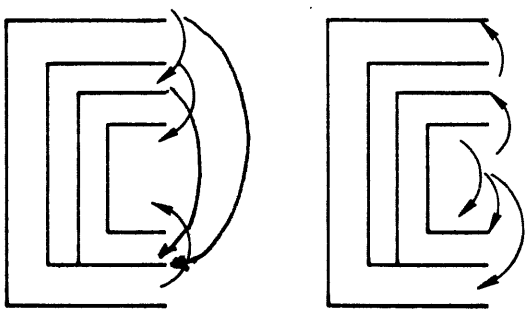
```
      DØ 100 L=2,LIMIT
          .
          .
          .
      DØ 10 I=1,10
      DØ 10 J=1,10
          .
          .
          .
10    CØNTINUE
          .
      DØ 20 K=K1,K2
          .
          .
20    CØNTINUE
          .
          .
          .
100   CØNTINUE
```

```
      DØ 5 I=1,5
      DØ 5 J=I,10
      DØ 5 K=J,15
          .
          .
          .
5     CØNTINUE
```

6.3.3
DØ Loop Transfer

A DØ loop should be entered by the execution of the DØ statement. One exception is allowed: once the DØ statement has been executed and before the loop is satisfied, control may be transferred out of the DØ range to perform some calculation and then transferred back into the range of the DØ.

In a DØ nest, a transfer may be made from an inner DØ loop into a DØ loop that contains it, but should not be made from the outer DØ loop to the inner DØ loop without first executing the DØ statement of the inner DØ loop. The compiler issues no diagnostics for some such transfers but the program will execute incorrectly.



Not Allowed Allowed

6.4 CONTINUE STATEMENT

n CØNTINUE

The usual function of the CØNTINUE statement is to provide the statement label n. It is most frequently used as the last statement of a DO loop to provide a label for the loop termination, particularly when a GØ TØ or IF would normally be the last statement of the loop. If CØNTINUE is used elsewhere in the source program, it acts as a do-nothing instruction and control passes to the next sequential program statement. The CØNTINUE statement must contain a statement number in columns 1-5.

Example:

```
    25 CØNTINUE
```

6.5 PAUSE STATEMENT

PAUSE

PAUSE n

$n \leq 5$ octal digits without an Ø prefix or B suffix. PAUSE n stops program execution with the words PAUSE n displayed as a message to the operator. An operator entry from the console can continue or terminate the program. Program continuation proceeds with the statement immediately following PAUSE. If n is omitted, it is understood to be blank.

6.6 STOP STATEMENT

STØP

STØP n

$n \leq 5$ octal digits without an Ø prefix or B suffix. STOP terminates the program execution and returns control to the monitor. The message STØP n is placed in the Job Log. If n is omitted, it is understood to be blank.

6.7 RETURN STATEMENT

RETURN

A subprogram normally contains one or more RETURN statements to indicate the end of logic flow within the subprogram and to return control to the calling program.

From function subprograms, control returns to the statement containing the function reference. From a subroutine subprogram, control returns to the next executable statement following the CALL. A RETURN statement in the main program causes execution to be terminated unless this is the pseudo-main program of an overlay (Chapter 8).

6.8 END STATEMENT

END

END must be the last physical line in every main program or subprogram. It is not executable. An attempt to execute it terminates the program. An END line in a subprogram does not act as a RETURN. The END line may not be numbered.

For compatibility the END line may include the name of the program or subprogram which it terminates, but this may cause errors and is not recommended.

7.1 SUBPROGRAM TYPES

A FORTRAN source program consists of a main program with or without subprograms. Subprograms are block data declarations or computational procedures returning zero, one, or more values as results. There are two kinds of procedural subprograms: subroutines and functions. In the following discussion, the term subprogram refers to both.* Subprograms may be compiled independently of one another. (Although used like other functions, a arithmetic statement function is compiled within the subprogram where it is referenced.) The object code for a subprogram is only loaded once for program execution, although the subprogram may be referred to in many places.

7.2 SUBPROGRAM COMMUNICATION

A calling program is a main program or subprogram that refers to another subprogram. The reference causes program control to be transferred to the subprogram called. A subprogram referenced (called) by a program may not have the same name as that program. Subprograms may call or be called by any other subprogram as long as the calls are nonrecursive; that is, when program A calls B, B may not call A, even indirectly.

The main program and subprograms communicate with each other via parameters and/or COMMON variables. The parameters appearing in a subroutine call or function reference are termed actual parameters. Corresponding arguments, called formal parameters, appear with the called subprogram as part of its definition. When a subprogram calls another, the actual parameters of the calling subprogram are associated with the corresponding formal parameters of the called subprogram; that is, the called subprogram acts as if it has been written using the actual parameters of the caller. If any values of the formal parameters are changed, the effect is as if the values of the actual parameters are changed; in this way new parameter values can be 'returned' from a called subprogram.

The actual parameter list in the calling program must contain the same number of parameters as the formal parameter list of the called subprogram. The corresponding parameters in the two lists must agree in type,** dimensionality, and intended use (i.e., whether the parameter will be altered by the subprogram or not).

---

* The term "subprogram" is also frequently used to include the main program as well. Also the term "program" may be used to refer to one subprogram or to a collection of subprograms. There are also library subprograms which are used without explicit CALL or function references.

** There is no automatic mode conversion if the types of the actual and formal parameters disagree.

CØMMØN variables (Section 5.3) may also be used to transmit parameters between subprograms, either from caller to called or vice versa. In this case those CØMMØN variables must also agree in type, dimensionality and intended use. The same variable should not occur both in a CØMMØN statement in the subprogram and as an actual parameter if it is altered under either guise.

7.2.1 Formal Parameters

Formal (dummy) parameters may represent the names of arrays, simple variables, functions, and subroutines. A name may not appear more than once in a formal parameter list. Since formal parameter names are local to the subprogram containing them, they may be the same as names appearing outside the subprogram in other contexts.

No element of a formal parameter list may appear in a CØMMØN, EQUIVALENCE or DATA statement within the subroutine. If it does, a compiler diagnostic results.

When a formal parameter represents an array, it must be dimensioned within the subprogram. If it is not declared, the array name must appear without subscripts and only the first element of the array is available to the subprogram.

7.2.2 Actual Parameters

Permissible forms:

Arithmetic expression*

Logical expression*

Constant*

Simple or subscripted variable

Array name **

FUNCTIØN subprogram name*

Library function or subroutine name* } See Section 7.13

SUBRØUTINE name*

A calling program statement label, identified by suffixing the label with the character S . *

---

* This form must not be a result parameter, i.e., there must be no stores into it in the subprogram.

** Normally the array dimensions of the actual parameter as declared in the calling program will agree exactly with the dimensions of the formal parameters in the subprogram (See Variable Dimensions, Section 5.2.1). If not, full account must be taken of the actual allocation of array storage (section 2.6.1) so the subprogram can properly locate values in the array.

7.3 MAIN PROGRAM

The first statement of a main program may be of the following form:

PRØGRAM name (f_1,...,f_n)

The choice of program name is restricted in that it must not be that of any subprogram which is to be used with it (including library routines), nor any of the reserved names discussed in Section 15.6.

If the PRØGRAM statement is omitted, a program name of "START." is assumed with filesets INPUT and ØUTPUT.

The parameters f_i are symbolic names naming all input/output filesets required by the main program and its subprograms.* These parameters must satisfy the following conditions:

1. The file name INPUT must appear if any READ i statement is included in the program or its subprograms.

2. The file name ØUTPUT must appear if any PRINT statement is included in the program or its subprograms.

3. The file name PUNCH must appear if any PUNCH statement is included in the program or its subprograms.

4. The file name TAPEi** must appear if a READ(i,n), WRITE(i,n), READ(i), or WRITE(i) statement is included in the program or its subprograms. (i is an integer from 1 to 99 with no leading zeros.)

5. If I is an integer variable name for a READ(I,n), WRITE(I,n), READ(I), or WRITE(I) statement which appears in the program or its subprograms, the file names TAPEI$_{i_1}$,..., TAPEi$_{i_k}$** must appear. The integers i_1,...,i_k (with no leading zeros) must include all values which are assumed by the variable I. A fileset name TAPEI bears no relationship to the file being referenced by I in the above input/output statements.***

---

* Input/output statements are discussed in Chapter 10. Fileset name substitution during program loading is discussed in section 13.0.2.

** The use of the word "tape" in forming these names does not imply the use of actual tape devices.

*** File names may be used as variables within a main program or subprogram and do not have any intrinsic value assigned them. That is, the statement 'READ(INPUT,n) list' will cause an execution error if INPUT has not been set to some integer value i within the program; the file TAPEi must also have been declared in the PRØGRAM statement.

6. If the program has been overlaid (see chapter 8) the overlay filesets must not appear.

Example:

```
      PRØGRAM SHØW(INPUT,ØUTPUT,TAPE6)
           .
           .
           .
      READ 10,X
      NUNIT=6
      WRITE (NUNIT)B
           .
           .
           .
      CALL ØUT(6,Y)
      PRINT 10,Z
10    FØRMAT(F6.6)
      RETURN
      END
      SUBRØUTINE ØUT(I,B)
           .
           .
           .
      WRITE(I)B
           .
           .
           .
      RETURN
      END
```

File names may be made equivalent and/or their buffer lengths may be specified at compile time by the f_i parameters in the PRØGRAM statement in the form:

filename = buffer length (octal)

or

$filename_2$ = $filename_1$

Example 1:

```
PRØGRAM name (INPUT=2001)
```

A file name INPUT is declared and it is to have a buffer length of 2001_8 words. The buffer length may not be specified to be less than 101_8 words. For example, PRØGRAM X (INPUT=20) will cause a buffer of 101_8 to be formed. If the buffer length is not indicated, a standard buffer size of 1001_8 (513_{10}) words is allocated.

I/O buffers are further discussed in section 15.2.

Example 2:

```
PRØGRAM name (ØUTPUT,TAPE6=ØUTPUT)
```

TAPE6 is equivalent to the ØUTPUT file. That is, WRITE(6,n) list will produce output on file ØUTPUT. The file name to which an equivalence is made must appear previously in the parameter list and must not have been defined by equivalence.

Example 3:

```
PRØGRAM name(INPUT,ØUTPUT=10000,TAPE1=INPUT,TAPE2=ØUTPUT)
```

All input read from logical I/O unit number one will be taken from INPUT and all output written on logical I/O unit number two will be transmitted to the ØUTPUT fileset. A buffer length is specified by ØUTPUT=10000 which establishes a buffer length of 10000_8. The file INPUT has a buffer length of 1001_8. Separate buffers are not set up for TAPE1 and TAPE2.

7.4 SUBROUTINE SUBPROGRAM

A subroutine subprogram is a computation procedure which may return zero, one, or more values. A value or type is not associated with the subroutine name itself.

The first statement of a subroutine subprogram must have one of the following forms:

```
SUBRØUTINE name (p1,...,pn)

SUBRØUTINE name
```

name is a symbolic name and p_i are formal parameters; n may be 1 to 60.

Restrictions on subprogram names are discussed in Section 15.6.

The parameter list is optional. If no parameters are specified, the second form is used.

No variable name used in a subroutine subprogram may be the same as the name in the SUBROUTINE statement.

7.5 CALL STATEMENT

The executable statement in the calling program for referring to a subroutine is:

```
CALL name
   or
CALL name (p1,...,pn)
```

where name is the name of the subroutine being called* and p_i are actual parameters; n is 1 to 60. The name should not appear in any declarative statement in the calling program, with the exception of the EXTERNAL statement when name is also an actual parameter.**

The CALL statement transfers control to the subroutine. When a RETURN statement is encountered in the subroutine, control is returned to the next executable statement following the CALL statement in the calling program. If the CALL statement is the last statement in a DO loop, looping continues until the DO loop is satisfied.

Examples:

1.
```
      SUBRØUTINE BLDX(A,B,W)
      W=2.*B/A
      RETURN
      END
```

Calls

```
      CALL BLDX(X(I),Y(I),W)
      SUM = X(I) + SUM        (control returns to this statement
                               after subroutine is executed)
      CALL BLDX(SIN(5.),ØVEC(L) + H/2.,EVEC(I+J))
```

2.
```
      SUBRØUTINE MATMULT

      CØMMØN/ITRARE/X(20,20),Y(20,20),Z(20,20)
      DØ 10 I=1,20
      DØ 10  J=1,20
      Z(I,J) = 0.
      DØ 10 K=1,20
   10 Z(I,J) = Z(I,J) + X(I,K)*Y(K,J)
      RETURN
      END
```

Operations in MATMULT are performed on variables contained in the common block ITRARE. This block must be defined in all programs calling MATMULT, for example:

```
      CØMMØN/ITRARE/AB(20,20),CD(20,20),EF(20,20)
      CALL MATMULT
```

---

* See Section 15.6 for a discussion of names that may not be used.

** Very obscure difficulties may occur if the same actual parameter is supplied for more than one formal parameter. Even more obscure versions of the same problems may occur if a subprogram has access to a particular variable through both a COMMON declaration and its parameter list. The difficulties arise because the variable may be altered in one guise without the program being aware that this also alters the other guise.

```
3.    SUBRØUTINE AGMT(SUB,ARG)
      CØMMØN/ABL/XP(100)
      ARG = 0.
      DØ 5 I=1,100
    5 ARG = ARG + XP(I)
      CALL SUB
      RETURN
      END
```

Here the formal parameter SUB is used to transmit another subprogram name. The call to subroutine AGMT might be CALL AGMT(MULT,FACTØR), where MULT is specified in an EXTERNAL statement (Section 7.13).

7.6 FUNCTION SUBPROGRAM

A function is a computational procedure which returns a value associated with the function name. The mode of the function is determined by a type indicator or the name of the function. See 15.6 for restrictions on subprogram names.

The first statement of a function subprogram must be one of the following forms where name is a symbolic name and the p_i are formal parameters.* A FUNCTIØN statement must have at least one parameter: $1 \leq n \leq 60$.

FUNCTIØN name $(p_1,\ldots,p_n)$

type FUNCTIØN name $(p_1,\ldots,p_n)$

Type is REAL, INTEGER, DØUBLE PRECISIØN, DØUBLE, CØMPLEX, or LØGICAL. When the type indicator is omitted, the mode is determined by the first character of the function name according to the rule used for the implicit typing of variable names. The name of the function must be the name of a simple variable appearing in the function subprogram. Its value at the time of the return from the function subprogram is the value of the function.

This variable must be assigned a value by appearing at least once in the function subprogram as any of the following:

- On the left-hand side of a replacement statement
- As an element of an input list
- As an actual parameter of a subroutine reference (in which it must be assigned a value)

---

* See footnote **, section 7.5

7.7 FUNCTION REFERENCE

In the general form

$$\text{name } (p_1,\ldots,p_n)$$

name identifies the function referenced. It is a symbolic name and its type is determined in the same way as a variable identifier. The p_i are actual parameters*, n is 1 to 60.

A function reference may appear in any expression where a subscripted variable may be used. The evaluated function has a single value associated with the function name.

When a function reference is encountered in an expression, control is transferred to the function subprogram.** A value for the function is computed and control is then returned to the statement containing the function reference.

The function name may not appear in any declarative statement in the calling program except a type statement or in an EXTERNAL statement when the name is used as an actual parameter.

Examples:

1.
```
      FUNCTIØN GRATER(A,B)
      IF(A.GT.B)1,2
    1 GRATER=A-B
      RETURN
    2 GRATER=A+B
      RETURN
      END
```

A reference to the function GRATER might be:

```
W(I,J)=FA+FB-GRATER(C-D,3.*AX/BX)
```

2.
```
      FUNCTIØN PHI(ALPHA,PHI2)
      PHI=PHI2(ALPHA)
      RETURN
      END
```

This function might be referenced:

```
      EXTERNAL SIN
      C=D-PHI(Q(K),SIN)
```

The replacement statement in the function PHI will be executed as if it had been written PHI=SIN(Q(K))

---

* See footnote **, section 7.5

** Except for statement functions (see section 7.8) and the Fortran intrinsic functions (see Chapter 11).

7.8 STATEMENT FUNCTION

A statement function is defined by a single expression and applies only to the subprogram containing the definition. The name of the statement function is a symbolic name; a single value is always associated with the name.

A statement function definition has the form:

name $(p_1,\ldots,p_n)$ = E

the formal parameters p_i are symbolic names and n is 1 to 60. The expression E may be any arithmetic or logical expression, which may contain references to any other functions. The nonparameter names appearing in the expression have the same values as they have outside the function.

A statement function reference has the form:

name $(p_1,\ldots,p_n)$

name is the name of the statement function; the actual parameters p_i may be any expressions.

During compilation, the statement function definition is compiled once in the subprogram and a transfer is made to this portion of the subprogram whenever a reference is made to the function. The value of the function is calculated using the actual parameters. Control is then returned to the statement containing the reference.

Actual and formal parameters must agree in number, order, and mode. The mode of the evaluated statement function is determined by the name of the function. However, the mode of the right-hand expression is determined by the highest mode of the formal parameters of the function.

The statement function name must not appear in a DIMENSIØN, EQUIVALENCE, CØMMØN, or EXTERNAL statement; the name of the function and its formal parameters may appear in a type declaration but cannot be dimensioned. Statement function names must not appear as actual or formal parameters.

A statement function definition must precede the first statement in which it is used, but it must follow all declarative statements (DIMENSIØN, type, etc.) which contain symbolic names referenced in the statement function, including formal parameters. All statement functions should precede the first executable statement; otherwise, a warning diagnostic is provided.

Omission of dimensioning information may cause later assignment statements to have the appearance of statement functions and lead to confusing diagnostics about statement functions.

A statement function may not reference itself and if such an attempt is made, a fatal diagnostic is provided.

Example definitions:

```
LØGICAL A,B,EQV
EQV(A,B)=(A.AND.B).ØR.(.NØT.A.AND..NØT.B)
CØMPLEX Z
Z(X,Y)=(1.,0.)*EXP(X)*CØS(Y)+(0.,1.)*EXP(X)*SIN(Y)
GRØPAY(RATE,HRS,ØTHRS)=RATE*HRS+RATE*.5*ØTHRS
```

Examples of use:

```
NETPAY=GRØPAY(1.25,HØURS(I),ØVTIME(I))-DEDUCT(I)-TAX
RESULT=(Z(BETA,GAMMA(I+K))**2-1.)/SQRT(TWØPIE)
```

7.9 LIBRARY SUBPROGRAMS

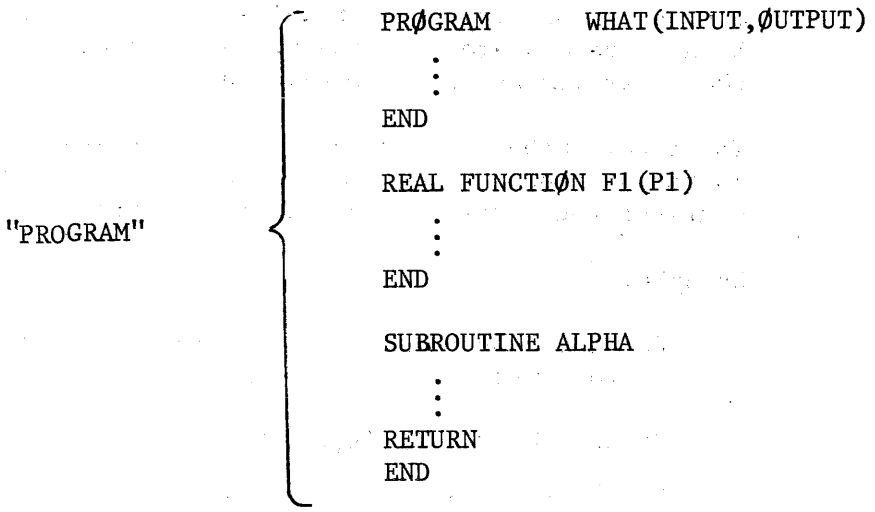
Subprograms that are used frequently have been stored in a reference file called a library. Library subroutine calls and function references may appear in the main program and/or subprograms in the same manner as do the programmer's own subprograms. The call acts as the request for the library program and causes a copy of its object code to be included during loading of the programmer's main program and any subprograms.

One exception to this procedure is for the intrinsic functions listed in Chapter 11. The object code for such a function is compiled directly into a subprogram at each place where it is referenced.

The names, parameters, and result type of standard library functions and subroutines are listed in Chapter 11. Errors detected by the library functions at execution time are listed in Chapter 15.

7.10 PROGRAM ARRANGEMENT

Frequently, the whole grouping of a main program, its subroutines and its functions are referred to as a "PROGRAM". A typical arrangement of a main program and a set of subprograms follows. The main program does not have to appear first.



See Chapter 14 for a discussion of a complete sample job. The ordering of subprograms is significant in case the program is to be overlaid (see chapter 8).

7.11 VARIABLE DIMENSIONS ON SUBPROGRAMS

In many subprograms, especially those performing matrix manipulation, the programmer may wish to use different array dimensions each time the subprogram is called.

This is accomplished by specifying the array name and its dimensions as formal parameters in the FUNCTION or SUBROUTINE statement. The corresponding actual parameters specified in the calling program are used by the called subprogram. The dimensions that may be transmitted to a subprogram should be the same as the dimensions of the array in the calling subprogram.

The use of variable dimensions does not control storage allocation; it merely informs the subprogram of the dimensions in use in the calling program.

The formal parameters representing the array dimensions must be simple integer variables. The array name must also be a formal parameter. The actual parameters representing the array dimensions must have positive integer values.

The total number of elements of the corresponding array in the subprogram must not exceed the total number of elements of the array in the calling program.

Example:

Consider a simple matrix add routine written as a subroutine:

```
      SUBRØUTINE MATADD (X,Y,Z,M,N)
      DIMENSIØN X(M,N),Y(M,N),Z(M,N)
      DØ 10 I=1,M
      DØ 10 J=1,N
   10 Z(I,J)=X(I,J) + Y(I,J)
      RETURN
      END
```

The arrays X, Y, Z and the variable dimensions M,N must all appear as formal parameters in the SUBRØUTINE statement and also in the DIMENSIØN statement as shown. If the calling program contains the array allocation declaration

```
DIMENSIØN A(10,10),B(10,10),C(10,10),E(5,5),F(5,5),G(5,5),H(10,10)
```

the program may call the subroutine MATADD from several places within the main program as follows:

```
      CALL MATADD(A,B,C,10,10)
      CALL MATADD(E,F,G,5,5)
      CALL MATADD(B,C,A,10,10)
      CALL MATADD(B,C,H,10,10)
```

The compiler does not check to see if the limits of the array established by the DIMENSIØN statement in the main program are exceeded.

7.12 ENTRY STATEMENT

This statement has a single purpose. It provides an alternate entry point to a function or subroutine subprogram.

ENTRY name

Name is a symbolic name and may appear within the subprogram only in the ENTRY statement. The entry name may not be followed by a formal parameter list. Formal parameters, if there are any, are the same as those which appear in the FUNCTIØN or SUBRØUTINE statement. Each entry name must appear in a separate ENTRY statement.

ENTRY may appear anywhere within the subprogram except it should not appear within a DO; the ENTRY statement cannot be labeled. The first executable statement following ENTRY becomes an alternate entry point to the subprogram.

In the calling program, the reference to the entry name is made just as if reference were being made to the function or subroutine in which the ENTRY is embedded. That is, an equivalent parameter list must be used. The name may appear in an EXTERNAL statement and, if a function entry name, in a type statement.

In a function subprogram, an ENTRY name assumes the same type as the name in the FUNCTIØN Statement. The ENTRY name may not be given type explicitly in the defining program.

Examples:

```
      FUNCTIØN JØE(X,Y)
 10   JØE=X+Y
      RETURN
      ENTRY JAM
      IF (X.GT.Y) 10,20
 20   JØE=X- y
      RETURN
      END
```

Note that the formal parameters are X and Y, and that the function returned is that value stored in JØE even when entry is via JAM. This function could be called from the main program as follows:

```
      .
      .
      .
Z = A + B - JØE(3.*B,Q-1)
      .
      .
      .
R = S + JAM(Q,2.*P)
      .
      .
```

With the JAM call, the values of Q and 2.*P are used as X and Y. The argument list matches that of the function name JØE, and the function value JØE is returned and added to S.

7.13 EXTERNAL STATEMENT

When an actual parameter is the name of a function or a subroutine subprogram name, that name must be declared in an EXTERNAL statement in the calling program.

EXTERNAL $name_1$, $name_2$, ...

The EXTERNAL statement must precede the first statement which calls a function or subroutine subprogram using the EXTERNAL name. When it is used, EXTERNAL always appears in the calling program; it may not be used with statement functions. If it is a compiler diagnostic is provided.

Examples:

1: A function name used as an actual parameter requires an EXTERNAL statement.

Calling Program Reference

```
      .
      .
EXTERNAL SIN
CALL PULL(SIN,R,Q)
      .
      .
```

Called Subprogram

```
SUBRØUTINE PULL(X,Y,Z)
      :
Z=X(Y)
      :
END
```

But a function reference used as an actual parameter does not need an EXTERNAL statement.

Calling Program Reference

```
      :
CALL PULL(SIN(R),Q)
      :
```

Called Subprogram

```
SUBRØUTINE PULL(X,Z)
      :
Z = X
      :
END
```

2: A subroutine used as an actual parameter must have its name declared in an EXTERNAL statement in the calling program.

```
CØMMØN/ABL/ALST(100)
EXTERNAL RTENTA, RTENTB
CALL AGMT(RTENTA,V1)
CALL AGMT(RTENTB,V1)
```

When a subprogram name appears as an actual parameter, any parameters to be associated with a call of this subprogram must appear as separate actual parameters.

Example:

Calling Program

```
EXTERNAL ADDER
      :
CALL SUB (ADDER,A,B)
      :
```

Called Subprogram

```
SUBRØUTINE SUB(X,Y,Z)
      :
CALL X(Y,Z)
      :
END
```

CALL SUB(ADDER(A,B)) would imply that ADDER is a function value, not a subroutine name.

7.14 BLOCK DATA SUBPROGRAM

A block data subprogram may be used in place of DATA declarations in the procedural subprograms to enter data into labeled common blocks prior to program execution. The form is:

```
BLØCK DATA name
  .
  .
  .
FORTRAN declaration* statements only
  .
  .
  .
END
```

All elements in the common blocks must appear in a CØMMØN declaration in the subprogram even if they are not in the DATA declaration.

Example:

```
BLØCK DATA FIRST
CØMMØN/ABC/A(5),B,C,/DEF/D,E,F
CØMPLEX D,E
DØUBLE PRECISIØN F
DATA (A(L),L=1,5)/2.3,3.4,3*7.1/,B/2034.756/,D,E,F/2*
1(1.0,2.5),7.86972415872E30/
END
```

The BLØCK DATA name may not be the same as that of any other subprogram (including main and library subprograms) to be loaded with it.

Standard Fortran requires that all DATA statements which set initial values of variables in labeled common blocks appear in BLØCK DATA subprograms.

---

* That is CØMMØN, DATA, DIMENSIØN, EQUIVALENCE and type statements.

OVERLAY STRUCTURES 8

8.1 Overlays

When a program is too large for all parts of it to fit in the available central memory of the computer simultaneously as a linear load (the ordinary case), it must be divided into pieces which can be fetched as needed to fulfill the program function. The technique employed in this division is called overlaying, and the pieces into which the program is divided are called links. The overall structure of the overlaid program is a tree consisting of a root (the first link) and a set of overlays (the remaining links). Each link is a collection of the subprograms defined by the source program or selected from a library. The grouping of the subprograms into links is determined by the programmer on the basis of the logical and temporal structure of the program.

Even in cases where a program will fit in the memory available, it may be advisable to use overlays to reduce the cost or to improve turnaround by permitting the program to execute in a smaller central memory field.

Each overlay is a group of subprograms plus a pseudo-main program introduced to define the initial entry to the overlay. Overlays are in absolute (as opposed to relocatable) form which is loaded into central memory at the request of the program being executed. The process of loading a link during execution is very fast, consisting essentially of one read operation. Since loading a link is such a simple operation, the subprogram required to perform the load is quite small compared with a full-scale loader which must be able to relocate and link together subprograms, search libraries, create memory maps, etc.

The distinction between overlay generation (which requires the full-scale loader), and overlay execution (which requires only a small library routine to load the overlays which have already been created) must be kept in mind throughout the following discussions. This point is a frequent source of confusion in the use of overlays. For more information see CALIDOSCOPE Control Statements and L3 CAL CLDR which describes an alternate method of defining overlays with increased flexibility, and usually without requiring modification of the source program.

When a link is loaded during execution it is in the form in which it was written by the loader and loading it does not alter the other links which remain in memory. In particular, if a link is loaded, executed, and overlaid, and is then loaded again, all storage contained in it is reset to contain the original values regardless of any changes that may have been made during the first execution. On the other hand, common blocks which are initialized by DATA statements contained in the programs making up the link will be reset only if the common blocks

themselves are contained in the link. DATA statements are effective only at the time the overlays are generated and the values given are incorporated into the link which actually contains the common block in question.

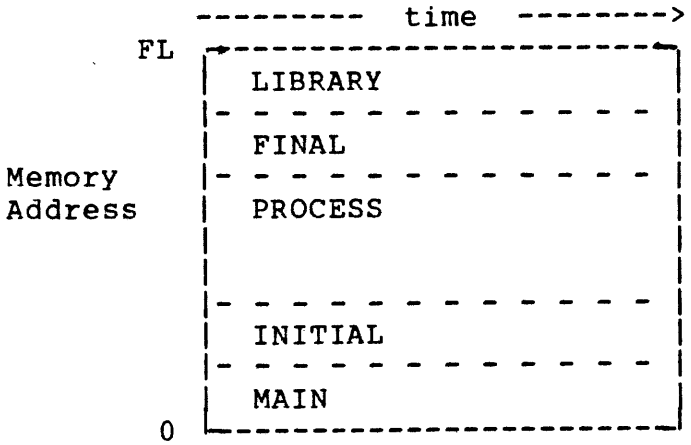
Blank common storage is normally allocated at the end of the lowest level link which declares it. If it is not declared in the root, its contents will be lost whenever the link it follows is overlaid.

A library search is made for unsatisfied externals after the decks have been read for each overlay. An external will be unsatisfied at this point if it is referenced by a deck in the current overlay but not defined by either the root or one of the decks just read or by the associated primary, if the current overlay is a secondary (see next section).

8.2 Overlay Execution

When an overlaid program is to be executed, the root is loaded first and remains in memory throughout the program execution. The root may call in primaries which may in turn call in their associated secondaries.

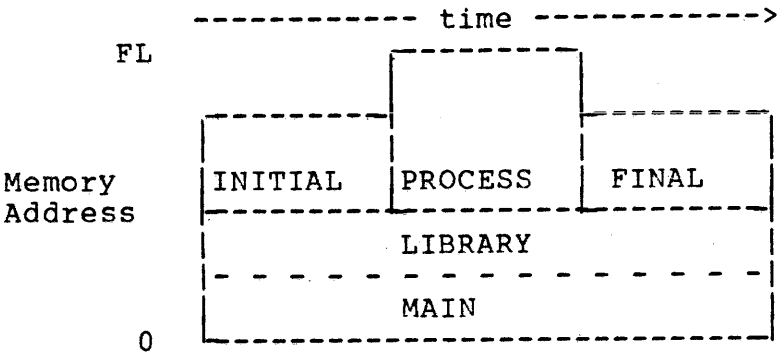
As an example, consider a program which which is divided into three phases which are executed consecutively. In an ordinary linear load this program might occupy memory in the following fashion during execution:



In this case the memory occupied by PROCESS and FINAL is idle while the INITIAL phase is being executed. Once this phase is complete, the space it occupies will be idle for the rest of the time of execution. Similar considerations apply to the PROCESS and FINAL phases. In

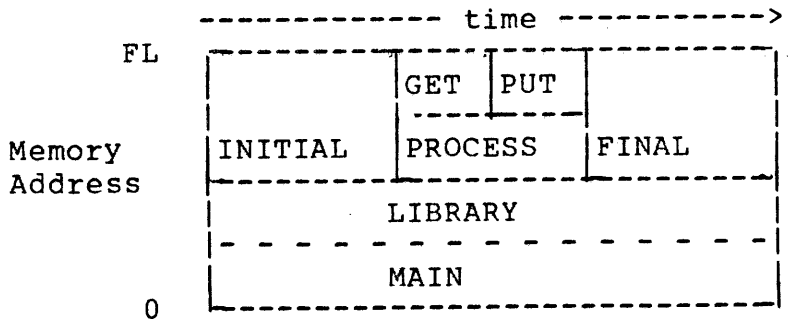
addition, parts of the library may be used by only one or two phases, but this will be ignored here to keep the example simple.

If the program were converted to an overlay structure, the above map could be replaced by one like this:



In this arrangement MAIN and LIBRARY are loaded first as the root, which then calls the link INITIAL. Once the processing performed by INITIAL is completed, the link PROCESS is loaded into the memory space previously occupied by INITIAL. Finally, PROCESS is replaced by the link FINAL to complete the processing required.

The process illustrated above may be continued so that, for example, the PROCESS overlay itself might be subdivided in a manner similar to that used for the entire program to give the following form which is still more compact:



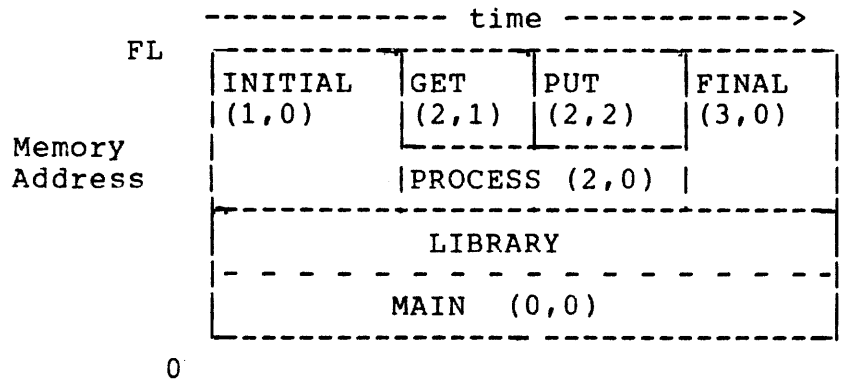
Clearly this process of subdivision cannot be carried on indefinitely without the overhead for additional overlays becoming too great. The program designer should ensure that excessive overlay requests are avoided by grouping interrelated subprograms together in a link where possible.

The overlays INITIAL, PROCESS, and FINAL may call subroutines contained in the root or reference labeled common blocks in the root with no more overhead than if they were in the same linearly loaded program (i.e., no overlay structure). Of course such a subroutine may not cause the

routine which called it to be overlaid if there is to be return to the calling routine. In general, external references from overlays must be directed toward the root. For example, a primary overlay may contain references to subprograms and common blocks in the root but not to those in a secondary overlay. A seconday may contain references to the root or to its associated primary.

The identification assigned to each link consists of a pair of numbers in the range 0-63 (0 to 77B) called the primary and secondary identifiers. The root is distinguished by having both these identifiers equal to zero. An overlay whose primary identifier is not zero but whose secondary identifier is zero is a primary. An overlay whose identifiers are both non-zero is a secondary. An overlay whose primary identifier is zero and whose secondary identifier is non-zero is an error.

The program may have only the root and at most one primary and one secondary (for a total of three links) in memory at one time. In the preceding example, MAIN and LIBRARY would constitute the root, and INITIAL, PROCESS, and FINAL would all be primaries. If PROCESS were subdivided as suggested, then the secondaries GET and PUT would have been introduced and the identifiers might be assigned as follows (where the parenthesized numbers are the identifier pairs for the links):



Since the maximum value an identifier may have is 63 (77B) this limits the number of primaries which may be included in an overlay tree as well as the maximum number of secondaries associated with a given primary. Note that the primaries immediately follow the root (which consists of MAIN and the library routines) in memory and each secondary immediately follows its associated primary (the primary with which it shares its primary identifier). A secondary is never loaded unless its associated primary is in memory. When a new primary is loaded during execution, any previously loaded primary (and secondary, if present) is destroyed. When a new secondary is loaded, any old secondary is destroyed, but the common primary remains. The root always remains throughout all overlay loading.

8.3 Overlay Directives

Every link begins with an OVERLAY directive. Each OVERLAY directive is inserted in the program deck immediately preceding the decks which make up the link to which it applies. OVERLAY directives encountered by the compiler are copied immediately to the fileset on which object decks are being written, from which they are to be read by the loader. An OVERLAY directive begins in column seven and has the following form:

```
OVERLAY (fn,l1,l2,Cnnnnnn)
```

where

fn fileset name onto which the generated overlay is to be written

l1 Primary identifier (in octal)

l2 Secondary identifier (in octal)

Cnnnnnn (optional): C is the character c and nnnnnn is 6 octal digits. If absent, this overlay is loaded normally. if present and blank common is assigned at the next lower level, this overlay is loaded nnnnnn words from the start of blank common. This provides a method for changing the size of blank common when the overlays are generated.

Both the primary and secondary identifiers must be in octal. However, the identifiers given in the CALL OVERLAY statement (see below) may be decimal.

The first overlay card must have a fileset name. Subsequent cards may omit fn; i.e., the form of the card is OVERLAY(l1,l2,Cnnnnnn) and the overlay is written on the same fileset.

The program card for the main program in the root must specify all needed fileset names, such as INPUT, OUTPUT, TAPE1, etc., for all links.

The pseudo-main program which defines the entry to each overlay differs from the real main program (in the root) in that it normally contains a RETURN statement to send control back to the routine which called the overlay, and the pseudo-main program contains no file declarations on the program card.

The structure selected for the overlaid program is imposed by ordering the overlay generation input as follows:

1. The input which defines the root appears first.

2. The input for each secondary overlay appears following its associated primary in the tree but preceding any subsequent primary.

There are many possible trees into which a program may be divided and it is up to the program designer to choose an appropriate one.

In order to create the overlay structure illustrated above and write it on the fileset named FILE, the input to the compiler could be a set of source decks and loader directives like the following:

```
OVERLAY(FILE,0,0)
(Decks for MAIN)
OVERLAY(FILE,1,0)
(Decks for INITIAL)
OVERLAY(FILE,2,0)
(Decks for PROCESS)
OVERLAY(FILE,2,1)
(Decks for GET)
OVERLAY(FILE,2,2)
(Decks for PUT)
OVERLAY(FILE,3,0)
(Decks for FINAL)
End of file
```

8.4 Overlay Calls

When an overlay is to be executed, it is called by the Fortran statement:

```
CALL OVERLAY (fn, l1, l2, p)
```

where

fn Hollerith constant or a variable containing the fileset name

l1 Primary identifier of the overlay

l2 Secondary identifier of the overlay

p Recall parameter. If p is 6HRECALL, the overlay is not to be reloaded if it is in memory; otherwise, specify zero.

All four parameters are required.

OVERLAY is a library subroutine which loads and executes the overlay requested.

The CALL OVERLAY statement not only loads the indicated overlay but also passes control to the first executable statement of the pseudo-main program in that overlay. No arguments can be transmitted except through shared common blocks. When the called pseudo-main program executes a RETURN, program control is given to the first executable statement after the CALL OVERLAY statement just as if the overlay pseudo-main program were the subroutine OVERLAY.

Examples:

```
DATA FILE/6HXYPLOT/
CALL OVERLAY (FILE,4,0,0)

NAME = 5LXTREE
CALL OVERLAY (NAME,1,2,6HRECALL)

CALL OVERLAY (5HXFILE,7,3,0)
```

Data transmission between storage and external units requires the FØRMAT statement (for coded mode only) and the I/O statement proper (Chapter 10). The I/O statement specifies the logical input/output unit, the process (READ, WRITE, etc.) and a list of any data to be moved. The unit of information transmitted to/from an external unit is called a record.* The FØRMAT statement specifies the manner in which the data is found or placed in records. In binary I/O statements no FØRMAT statement is used.

9.1 INPUT/OUTPUT LIST

The list portion of an input/output statement specifies the data items and the order, from left to right, of transmission. The input/output list can contain any number of elements in the following form:

$$a_1, a_2, a_3, \ldots$$

The list items a_i may be array names, simple or subscripted variables, or variables with implied DØ loops. Constants, functions, and expressions are not allowed. Items are separated by commas, and their order must correspond to any format specification associated with the list. Coded records are always read or written until the list is satisfied, or, on reading, an end-of-file is reached. The interaction between a FORMAT and a list is further explained in Section 9.7.

Subscripts of variables in an I/O list may be only in the following forms:

c*I±d

I±d

c*I

I

c

c and d are unsigned integer constants, and I is a simple integer variable, previously defined, or defined within an implied DØ loop.

---

* In the discussion presented in this Chapter the word "record" is used to refer to a unit record. The usual input unit record is an 80 column punched card, and the usual output unit record is a printed line of up to 133 characters. Unit records on other storage media are described in Chapter 13.

Examples of lists, used here with READ statements, follow. Note the number following the word READ is not part of the list, but refers to the FØRMAT statement to be described in Section 9.2.

Examples:

```
READ 100, A,B,C,D
READ 200, A,B,C(I),D(3,4),E(I,J,7),H
READ 101, J,A(J),I,B(I,J)
READ 102, DELTA(5*J+2,5*I-3,5*K),C,D(I+7)
READ 202, DELTA
READ 300, A,B,C,(D(I),I=1,10),E(5,7),F(J),(G(I),H(I),I=2,6,2)
READ 400, I,J,K,(((A(II,JJ,KK),II=1,I),JJ=1,J),KK=1,K)
READ 500, ((A(I,J),I=1,10,2),B(J,1),J=1,5),E,F,G(L+5,M-7)
```

9.1.1 Array Transmission

Part or all of an array can be represented for transmission as a single I/O list item by using an implied DO notation of the general form:

$$(((A(I,J,K),L_1=m_1,m_2,m_3),L_2=n_1,n_2,n_3),L_3=p_1,p_2,p_3)$$

which is proportionately simpler if the array has fewer subscripts, and where

m_i,n_i,p_i are unsigned integer constants or simple integer variables. If m_3,n_3, or p_3 is omitted, it is assumed equal to 1.

I,J,K are subscripts of A.

L_1,L_2,L_3 are index variables I,J,K in some order.

During execution, each subscript (index variable) is set to the initial index value: $L_1 = m_1$, $L_2 = n_1$, $L_3 = p_1$. The first (innermost) index variable defined in the list is incremented first, following the rules for nested DO loop execution. When the first index variable has reached its limit, m_2, it is reset to m_1; the next index variable to the right is incremented, and the process is repeated until all the index variables have been incremented to their maximum value. If m_1 is greater than m_2 initially, L_1 is given the value m_1 only.

An array name which appears without subscripts in an I/O list causes transmission of the entire array by columns (as in Section 2.6.1). This form is sometimes referred to as "short list" notation. Thus if B is dimensioned as

```
DIMENSIØN B(10,15)
```

the statement

```
READ 13,B
```

is equivalent to

```
READ 13,((B(I,J),I=1,10),J=1,15)
```

but the first form is faster.

An implied DØ loop can be used to transmit a simple variable more than one time. For example, the list item (B,A(K), K=1,5) causes the transmission of variable B five times. A list of the form K,(A(I),I=1,K) is permitted and, when reading, the input value of K is used in the implied DØ loop.

Examples:

1. Simple implied DØ loop list items

```
     READ 400,(A(I),I=1,10)
400  FØRMAT (E20.10)
```

With this format, the READ statement is equivalent to the following DØ loop.*

```
     DØ 5 I=1,10
5    READ 400, A(I)
```

2.
```
     READ 100, ((A(JV,JX),JV=2,20,2),JX=1,30)
     READ 200, (BETA(3*JØN+7),JØN=JØNA,JØNB,JØNC)
     READ 300 (((ITMLST(I,J+1,K-2),I=1,25),J=2,N),K=IVAR,IVMAX,4)
```

3.
```
     READ 600, (A(I),B(I),I=1,10)
600  FØRMAT (F10.2,E6.1)
```

With this format, the previous READ statement is equivalent to the DØ loop:*

```
     DØ 17 I=1,10
17   READ 600,A(I),B(I)
```

4.
```
     PRINT 700,(I,I=1,10)
```

This statement will print the index variable I, I=1,2,3,...10.

---

* Equivalence of the forms here depends on the fact that the FØRMAT statement specifies the same number of items as the list without the implied DØ. If the FØRMAT statement were different, the same results would not be produced since a formatted READ statement always begins a new unit record. See Section 9.7.

5. Nested implied DØ list items.

```
READ 100,((((A(I,J,K)B(I,L),C(J,N),I=1,10),J=1,5),K=1,8),
1L=1,15),N=2,7)
```

Data is transmitted in the following sequence (summarized):

```
A(1,1,1),B(1,1),C(1,2),A(2,1,1),B(2,1),C(1,2). . .
. . .A(10,1,1),B(10,1),C(1,2),A(1,2,1),B(1,1),C(2,2). . .
. . .A(10,2,1),B(10,1),C(2,2). . .A(10,5,1),B(10,1),C(5,2). . .
. . .A(10,5,8),B(10,1),C(5,2). . .A(10,5,8),B(10,15),C(5,2). . .
. . .A(10,5,8),B(10,15),C(5,7)
```

6. The following list item will transmit the array E(3,3) by columns:

```
READ 100,((E(I,J),I=1,3),J=1,3)
```

The following list item will transmit the array E(3,3) by rows:

```
READ 100,((E(I,J),J=1,3), I=1,3)
```

7. Short list notation.

```
DIMENSIØN MATRIX(3,4,7)
READ 100, MATRIX
```

The above items are equivalent to the following statements:

```
DIMENSIØN MATRIX(3,4,7)
READ 100,(((MATRIX(I,J,K),I=1,3),J=1,4),K=1,7)
```

This list is equivalent* to the nest of DO loops:

```
DØ 5 K=1,7
DØ 5 J=1,4
DØ 5 I=1,3
5  READ 100, MATRIX(I,J,K)
```

provided the FORMAT specifies one item per record, e.g.,

```
100  FØRMAT (I6)
```

9.2 FØRMAT DECLARATION

Coded input/output statements require a NAMELIST (see section 9.9) or a FORMAT declaration which contains conversion and editing information relating to internal/external structure of the corresponding I/O list items. A FORMAT declaration has the following form:

FØRMAT ($spec_1$, . . . ,k($spec_m$, . . .),$spec_n$, . . .)

$Spec_i$ format specification

k optional repetition factor which must be an unsigned integer constant.

---

* See previous footnote to this section.

The FØRMAT declaration is non-executable and may appear anywhere in the program. FØRMAT declarations must have a statement label in columns 1-5.

The data items in an I/O list are converted from one representation to another according to FØRMAT conversion specifications. An input conversion specification converts a coded data item into the internal word structure specified. Output conversions perform the reverse task. To be meaningfully converted, the data must agree with the format specifications in order and type. No type checking between list and FØRMAT is performed during execution. Formats may also contain editing specifications. In the examples below, the lower case letter "b" will often be used to indicate a blank.

9.2.1 Conversion Specification Summary:

Ew.d	Single precision real with exponent
Fw.d	Single precision real without exponent
Dw.d	Double precision real with exponent
Gw.d	Single precision real with or without exponent
Iw	Decimal integer conversion
⌀w	Octal integer conversion
Aw	Alphanumeric conversion
Rw	Alphanumeric conversion
Lw	Logical conversion

w, n and d are unsigned integer constants; w specifies the field width in number of character positions in the external record, and d specifies the number of digits to the right of the decimal, i.e., the fractional portion, within that field. The total field widths specified for one record must be ≤ 80 characters for card input/output or ≤ 132 characters for printer or ≤ 137 for tapes (see Chapter 11, LNGBCD).

Each complex data item in a list is converted on input/output according to a pair of consecutive Ew.d or Fw.d specifications.

Example:

```
      CØMPLEX A,B
      PRINT 10,A
   10 FØRMAT (1X,F7.2,F9.2)
      READ 11,B
   11 FØRMAT (E10.3,E10.3)
```

9.2.2 Editing Specification Summary:

±nP	Scaling factor (decimal)
wX	Space w columns to the right
wH	Insert/Receive w characters
Tn	Tab to column n
Z	Print leading zeros
/	Begin new record
...	Insert/Receive character (no *) string between * delimiters
≠...≠	Insert/Receive character (no ≠) string between ≠ delimiters

9.3 CONVERSION SPECIFICATIONS

In general, the following can be stated about numeric conversion specifications:

On input, blanks in numeric fields are interpreted as zeros; but a field of all blanks is converted to a minus zero.* The use of the plus sign is optional. A minus zero value may be tested for by use of the function MZERO (see 11.2.25).

On output, if the coded representation of a number is not large enough to fill the field allowed, it is right-adjusted in the field with leading blanks inserted. If too large, the field is filled with #s. When a zero is put out in a real format, no zeros appear after the decimal point. This provides a convenient distinction between an actual zero and a very small number.

9.3.1 Ew.d Output (REAL)

Real numbers in storage are converted to the coded character form for output with the E conversion, Ew.d. The field occupies w positions in the output record; with the real number right justified in the form:

b .a. . .a±eee $100 \leq eee \leq 323$

or

b .a. . .aE±ee $0 \leq ee \leq 99$

b indicates blank character or - sign, a's are the most significant digits of the integer and fractional part and eee are the digits in the exponent. If d is zero or blank, digits to the right of the decimal do not appear. Field width w must be sufficient to contain the significant digits, signs, decimal point, the letter E, and the exponent. Generally, $w \geq d+6$. Positive numbers need not reserve a space for the sign of the number.

If the field is not wide enough to contain the output value, #'s are inserted in the field. If the field is longer than the output value, the quantity is right justified with blank fill to the left. If the value is infinite or indefinite (see Appendix A), the field contains R or I respectively.

Examples:

```
      PRINT 10,A                                   A contains -67.32
10    FORMAT(1X,E10.3)                                 or    +67.32
```

Printed Result: b-.673E+02 or bb.673E+02

```
      PRINT 10,A
10    FORMAT(1X,E13.3)
```

Printed Result: bbbb-.673E+02 or bbbbb.673E+02

```
      PRINT 10,A
10    FORMAT(1X,E8.3)
```

Printed Result: ######## or .673E+02

Provision not made for sign

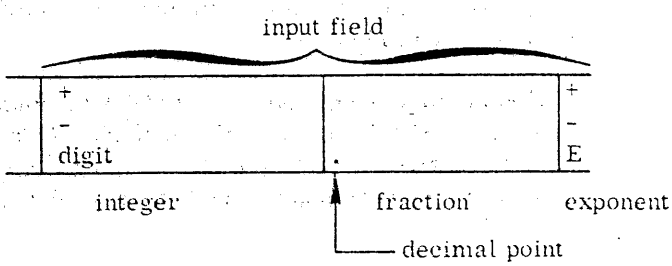
---

* Minus zero does not exist in all Fortran systems. Programs which depend on it may not be transferrable to such systems.

9.3.2
Ew.d Input
(REAL)

The E specification converts the number in the input field to a real number and stores it in the proper location.

Subfield structure of the input field:



The total number of characters in the input field is specified by w: this field is scanned from left to right: blanks are interpreted as zeros.

The integer subfield begins with a sign (+ or -) or a digit and may contain a string of digits. The integer field is terminated by a decimal point, D, E, +, -, or the end of the input field.

The fraction subfield which begins with a decimal point may contain a string of digits. The field is terminated by D, E, +, -, or the end of the input field.

The exponent subfield may begin with D, E, + or -. When it begins with D or E, the + is optional between D or E and the string of digits of the subfield. The value of the string of digits in the exponent subfield must be less than 323.

Trailing blanks are interpreted as zeros.

Permissible subfield combinations:

+1.6327E-04	integer fraction exponent
-32.7216	integer fraction
+328+5	integer, exponent
.629E-1	fraction exponent
+136	integer only
136	integer only
.07628431	fraction only
E-06 (interpreted as zero)	exponent only

In the Ew.d specification, d acts as a negative power-of-ten scaling factor when an external decimal point is not present. The internal representation of the input quantity is:

$$(\text{integer subfield}) \times 10^{-d} \times 10^{(\text{exponent subfield})}$$

For example, if the specification is E7.8, the input quantity 3267+05 is converted and stored as: $3267 \times 10^{-8} \times 10^{5} = 3.267$.

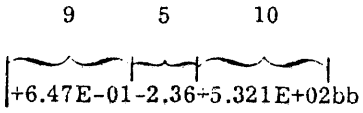
A decimal point in the input field overrides d. The input quantity 3.67294+5 read by an E9.d specification is always stored as 3.6729×10^{5}. When d does not appear, it is assumed to be zero.

The field length specified by w in Ew.d should always be the same as the length of the field containing the input number. When it is not, incorrect numbers may be read, converted, and stored as shown below. The field w includes the significant digits, signs, decimal point, E or D, and exponent.

Example of <u>incorrect</u> data input:

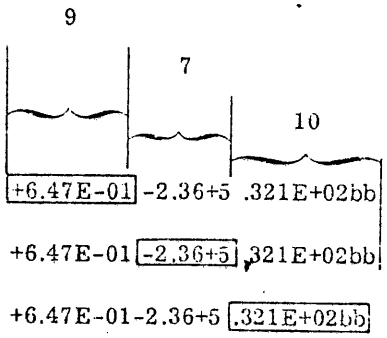
```
      READ 20,A,B,C
20    FØRMAT (E9.3,E7.2,E10.3)
```

Input quantities on the card are in three contiguous fields columns 1 through 24:



The second specification (E7.2) exceeds the width of the second field by two characters.

Reading proceeds as follows:



First, +6.47-01 is read, converted, and placed in location A. Next, -2.36+5 is read, converted, and placed in location B. The number actually desired was -2.36, but the specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input representation under the definitions and restrictions.

Finally, .321E+0200 is read, converted, and placed in location C. Here again, the input number is legitimate and is converted and stored, even though it is not the number desired.

The above example illustrates a situation where numbers are incorrectly read, converted, and stored, and yet there is no immediate indication that an error has occurred.

Examples of Ew.d conversion on input:

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	.14326	All subfields present
-12.437629E+1	E13.6	-124.37629	All subfields present
8936E+004	E9.10	.008936	No fraction subfield; input number converted as $8936. \times 10^{-10+4}$
327.625	E7.3	327.625	No exponent subfield
-.0003627+5	E11.7	-36.27	Integer subfield contains - only
-.0003627E5	E11.7	-36.27	Integer subfield contains - only
blanks	Ew.d	-0.	All subfields empty
1E1	E3.0	10.	No fraction subfield, input number converted as $1.x10^1$
E+06	E10.6	0.	No integer or fraction subfield; zero stored regardless of exponent field contents
1.bEb1	E6.3	10.	Blanks are interpreted as zeros
1.E1bb	E6.3	10^{100}	Trailing blanks in exponent are interpreted as zeros.

9.3.3
Fw.d Output (REAL)

The field occupies w positions in the output record; the corresponding list item must be a real (i.e., floating point) quantity, which appears as a decimal number, right justified:

ba. . .a.a. . .a

b indicates a blank or - sign. The a's represent the most significant digits of the number. The number of decimal places to the right of the decimal is specified by d. If d is zero or omitted, digits to the right of the decimal point do not appear.* If the number is positive, the + sign is suppressed. If the field is too short to accommodate the number, * fills the field. If the field is longer than required to accommodate the number, the number is right justified with blank fill to the left. If the output value is infinite or indefinite (see the end of section A.3), the field contains R or I respectively.

* When card fields are punched under F-specification, there is no way to suppress the decimal point except by programmed statements which punch the integral and decimal parts of the real number as two integers in adjacent fields.

Examples:

```
      PRINT 10,A                                  A contains +32.694
10    FORMAT(1X,F7.3)
        Printed Result:  b32.694

      PRINT 11,A
11    FORMAT (1X,F10.3)
        Printed Result:  bbbb32.694
      PRINT 12,A                                  A contains -32.694
12    FORMAT(1X,F6.3)
        Printed Result:  ######                   no provision for - sign

      PRINT 13,A,A                                A contains .32694
13    FORMAT(1X,F4.3,F6.3)
        Printed Result: .327b0.327
```

9.3.4
Fw.d Input
(REAL)

This specification is a modification of Ew.d. The input field consists of an integer and a fraction subfield. An omitted subfield is assumed to be zero. The restrictions described under Ew.d input apply, except that the scale factor will affect the F format (see Section 9.4). A decimal point punched in the field on the card will override the decimal point in the FORMAT specification.

Examples:

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field
37925	F5.7	.0037925	No fraction subfield; input number converted as 37925×10^{-7}
.62543	F6.5	.62543	No integer subfield
.62543	F6.2	.62543	Decimal point overrides d of specification
+144.15E-03	F11.2	.14415	Exponents are legitimate in F input and may have P-scaling
5bbbb	F5.2	500.00	No fraction subfield; input number converted as 50000×10^{-2}

9.3.5
Gw.d Output
(REAL)

The real data will be represented by F conversion unless the magnitude of the data exceeds the range that permits effective use of F conversion. In this case, the E conversion will represent the external output. Therefore, the effect of the scale factor is not implemented unless the magnitude of the data requires E conversion.

When F conversion is used under Gw.d output specification, 4 blanks are inserted within the field, right justified. Therefore, for effective use of F conversion, d must be $\leq$ w-6.

Examples:

```
PRINT 101, XYZ                          XYZ contains 77.132
   101 FORMAT(1X,G10.3)
     Printed Result:  77.132bbbb

PRINT 101, XYZ                          XYZ contains 1214635.1
   101 FORMAT (1X,G10.3)
     Printed Result:   bb.121E+07
```

9.3.6
Gw.d Input
(RFAL)

Gw.d specification is similar to the Fw.d INPUT specification.

9.3.7
Dw.d Output
(DOUBLE)

The field occupies w positions of the output record, the list item is a double precision quantity which appears as a decimal number, right justified:

b .a. . .a±eee $100 \leq eee \leq 323$

or

b .a. . .aD±ee $0 \leq ee \leq 99$

b indicates blank or minus zero. The specification for D conversion corresponds to the Ew.d output specification.

9.3.8
Dw.d Input
(DOUBLE)

D conversion corresponds to E conversion except that the list variables must be double precision names. D is equivalent to E as the beginning of an exponent subfield on the input record.

Example:

```
      DØUBLE Z,Y,X
      READ1,Z,Y,X
1     FØRMAT (D18.11,D15,D17.4)
```

Input Card:

```
-6.31675298443E-03+2.718926453147629347752886 9D-09
 18              15             17
```

9.3.9
Iw Output
(INTEGER)

I specification is used to convert decimal integer values. The output quantity occupies w output record positions, right justified:

ba. . .a

b is a blank or - sign. The a's are the most significant decimal digits of the integer. If the integer is positive, the + sign is suppressed. The maximum integer magnitude that can be put out is 281474976710655 ($2^{48} - 1$). If the integer is greater, the symbol R is placed right justified in the field.

If the field w is larger than required, the output quantity is right justified with blank fill to the left. If the field is too short, ≠s occupy the field. If the internal value is -0 and a field descriptor of I1 is given, - will be produced.

Example:

```
      PRINT 10,I,J,K                    I contains -3762
   10 FØRMAT (1X,I8,I10,I5)             J contains +4762937
                                        K contains +13
```

Printed Result: bbb-3762bbb4762937bbb13

8 10 5

9.3.10
Iw Input
(INTEGER)

The field is w characters in length, the list item is an integer variable, and the input data item is a decimal integer constant. The input field w consists of an integer subfield, which can contain only the characters +, -, 0 through 9, or blank. When a sign appears, it must precede the first digit in the field. Blanks (caution, this includes trailing blanks) are interpreted as zeros. The value is stored right justified in the specified variable. The maximum input value is 576460752303423487 ($2^{59}-1$).

Example:

```
      READ 10,I,J,K,L,M,N
   10 FØRMAT (I3,I7,I2,I3,I2,I4)
```

Input Card:

139bb-15bb18bb7b3b1b4

3 7 2 3 2 4

In storage:

I contains	139
J	-1500
K	18
L	7
M	3
N	104

9.3.11
Øw Output
(OCTAL)

Ø specification is used to convert octal integer values. The output quantity occupies w output record positions right justified:

aa. . .a

where a is an octal digit. If w is 20 or less, the rightmost w digits occur. If w is greater than 20, the number of right justified in the field with blanks to the left of the output quantity. A negative number is output in its seven's complement internal form.

9.3.12
Øw Input
(OCTAL)

Octal integer values are converted under Ø specification. The field is w characters in length, and the list item must be an integer variable.

The input field w consists of an integer subfield only (maximum of 20 octal digits) containing only the characters +, -, 0 through 7 or blank.

Only one sign may precede the first digit in the field. All blanks (including trailing blanks) are interpreted as zeros. Fields which contain only blanks are interpreted as -0.

Example:

```
      TYPE INTEGER P,Q,R
      READ 10,P,Q,R
10    FØRMAT (Ø10,Ø12,Ø2)
```

Input Card:

```
3737373737666b6644b444-0
   10          12    2
```

In storage:

```
P   0000000003737373737
Q   0000000666066440444
R   7777777777777777777
```

A negative octal number is represented internally in seven's complement form (20 digits) obtained by subtracting each digit of the octal number from seven. For example, if -703 is an input quantity, its internal representation is 7777777777777777074.

That is, 7777777777777777777
-00000000000000000703
7777777777777777074

9.3.13 Aw Output (ALPHANUMERIC)

A conversion is used to output alphanumeric characters. If w is 10 or more, the quantity appears right justified in the output field, blank fill to left. If w is less than 10, the output quantity represents the leftmost w characters of the internal word (see below). Binary zero characters are converted on output to 55B (blank).

9.3.14 Aw Input (ALPHANUMERIC)

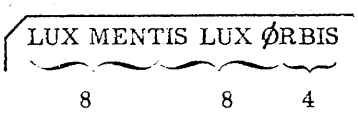
This specification accepts FORTRAN characters including blanks. The internal representation is display code (see Appendix A); the field width is w characters.

If w exceeds 10,* the input quantity is the rightmost 10 characters in the field. If w is 10 or less, the input quantity is stored as a left justified BCD word; the remaining spaces are blank filled.

Example:

```
      READ 10,Q,P,Ø
10    FØRMAT (A8,A8,A4)
```

Input Card:



In storage:

```
Q     LUXbMENTbb
P     ISbLUXbØbb
Ø     RBISbbbbbb
```

9.3.15 Rw Output (ALPHANUMERIC)

This specification is similar to the Aw Output with the following exception. If w is less than 10,* the output quantity represents the rightmost w characters of the internal word. If a binary zero is put out as a character, it is converted to the character code for blank.

---

* The number of characters which may be stored in a data item varies from one Fortran system to another. Ten is an unusually large limit. See Table B of Appendix A for examples of internal storage of alphanumeric data.

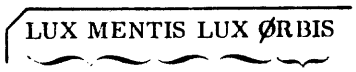
9.3.16
Rw Input
(ALPHANUMERIC)

This specification is the same as the Aw Input with the following exception. If w is less than 10,** the input quantity is stored as a right justified BCD word; the remaining spaces are filled with binary zeros.

Example:

```
      READ 10,Q,P,Ø
10    FØRMAT (R8,R8,R4)
```

Input Card:



In storage:

```
Q     00LUXbMENT  *
P     00ISbLUXbØ  *
Ø     000000RBIS  *
```

9.3.17
Lw Output
(LOGICAL)

L specification is used to output logical values. The output field is w characters long, and the list item must be a logical element.

A value of true or false in storage causes w-1 blanks followed by a T or F to be output.

Example:

```
      LØGICAL I, J, K, L      I contains -0    J contains 0
      PRINT 5, I, J, K, L     K contains -0    L contains -0
5     FØRMAT (4L3)
```

Result: bbTbbFbbTbbT

9.3.18
Lw Input
(LOGICAL)

This specification accepts logical quantities as list items. The field is considered true if the first non-blank character in the field is T or false if it is F. An all-blank field is considered false.

---

* Here the 0 represents the true zero (all zero bits), not the character zero.

** See footnote to section 9.3.15.

9.4 nP SCALE FACTOR

The D, E, F, and G conversion may be preceded by a scale factor which is: External number = Internal number $\times 10^{\text{scale factor}}$.

A scale factor is of the form:

nP

where n, the scale factor, is an unsigned or negative integer constant in the range $-8 \leq n \leq 8$.

When an input/output statement is initiated, a scale factor of zero is assumed. Once a scale factor has been given, it applies to all subsequently interpreted F,E,G, and D specifications, until another scale factor is encountered, and then that scale factor is used. For example, 0P nullifies the effect of a previous scale factor.

The scale factor is not automatically reset if the FØRMAT is rescanned.

Example:

```
FØRMAT(3PE12.6,F10.3,0PD18.7, -1P, F5.2)
```

The E12.6 and F10.3 specifications are scaled by 10^3, the D18.7 specification is not scaled, and the F5.2 specifition is scaled by 10^{-1} .

The scaling specification nP is not associated with a list element.

The format (3P,3I9,F10.2) is the same as the format (3I9,3PF10.2).

9.4.1 Fw.d Scaling

Input

The number in the input field is divided by 10^n and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is $314.1592 \times 10^{-2} = 3.141592$.

Output

The number in the output field is the internal number multiplied by 10^n. In the output representation, the decimal point is fixed; the number moves to the left or right, depending on whether the scale factor is plus or minus. For example, the internal number 3.1415926538 may be represented on output under scaled F specifications as follows:

Specification	Output Representation
F13.6	3.141593
1PF13.6	31.415927
3PF13.6	3141.592654
-1PF13.6	.314159

9.4.2
Ew.d or Dw.d Scaling

Input

The scale factor has the same effect as in the F specification unless the field has an exponent. In that event, there is no effect.

Output

The scale factor has the effect of shifting the output number left n places while reducing the exponent by n. Using 3.1415926538, some output representations corresponding to scaled E specifications are:

Specification	Output Representation
E20.2	0.31 E+01
1PE20.2	3.14 E-00
2PE20.2	31.41 E-01
3PE20.2	314.15 E-02
4PE20.2	3141.59 E-03
5PE20.2	31415.92 E-04
-1PE20.2	0.03 E+02

9.4.3
Gw.d Scaling

Input

Gw.d scaling on input is the same as Fw.d scaling on input.

Output

The effect of the scale factor is suspended unless the magnitude of the data to be converted is outside the range that permits the effective use of F conversion.

9.5 EDITING SPECIFICATIONS

Editing specifications, unlike the previous format conversion specifications, have no list elements associated with them. These specifications allow for insertion or skipping of characters in the input/output record and for starting a new record.

9.5.1 wX (SKIP)

This specification may be used to include w blanks in an output record or to skip w characters on an input record to permit spacing of input/output quantities. 0X is not permitted; bX is interpreted as 1X. In RUN Fortran, the comma following X is optional but it is required in standard Fortran.
Examples:

```
     INTEGER A                    A contains 7
     PRINT 10,A,B,C               B contains 13.6
                                  C contains 1462.37
 10  FØRMAT(3X,I2,3X,F6.2,6X,E12.5)
```

Printed Result: bbb7bbbb13.60bbbbbbb0.14624E+04*

```
     READ 11, R,S,T
 11  FØRMAT (F5.2,3X,F5.2,6X,F5.2)
```

Input Card:

14.62bb$13.78bCØSTb15.97

In storage:

R 14.62
S 13.78
T 15.97

9.5.2 wH Output (HOLLERITH)

With this specification 6-bit characters (given in the format itself), including blanks, may be put out in the form of comments, titles, and headings. w, an unsigned integer, specifies the number of characters to the right of H that are transmitted to the output record; w may specify a maximum of 133 characters. H denotes a Hollerith field; the comma following the field is optional.

Instead of prefixing the Hollerith field with wH, it may be defined by being contained between asterisks or not equal signs. (Therefore, *s are prohibited in a field delimited by asterisks and ≠s are prohibited in a field delimited by not equals.) No count is necessary. In either case, it is not necessary to set off a Hollerith field from other format specifiers (if any) by a comma.

---

* The same result is obtained with the FØRMAT(1X,I4,F9.2,E18.5).

Examples:

Source program:

```
     PRINT 20
 20  FØRMAT(28HbBLANKSbCØUNTbINbANbHbFIELD.)
```

or

```
 20 FØRMAT(*bBLANKSbCØUNTbINbANbHbFIELD.*)
```

produces the printed result:

```
 BLANKSbCØUNTbINbANbHbFIELD.
```

Source program:

```
     PRINT 30,A
 30 FØRMAT(6HbLMAX=,F5.2)
```

A contains 1.5. In the FØRMAT the comma after the = sign is optional.

produces the printed result:

```
 LMAX=b1.50
```

9.5.3
wH Input
(HOLLERITH)

The H specification may be used to read Hollerith characters into an existing H field within the FØRMAT specification.

Example:

Source program:

```
    READ 10
10  FØRMAT (27Hbbbbbbbbbbbbbbbbbbbbbbbbbbb)
```

Input Card:

bTHIS IS A VARIABLE HEADING

27 cols

After READ, the FØRMAT statement labeled 10 contains the alphanumeric information read from the input card; a subsequent reference to statement 10 in an output statement acts as follows:

```
PRINT 10
```

produces the printed result:

```
 THIS IS A VARIABLE HEADING
```

9.5.4
New Record

When an input/output statement is initiated, a new record is always started.* In addition, a new record is started when the end of the format is encountered if another list element is specified.

A slash (/) character in the specification list also signals the beginning of a new record. Consecutive slashes may appear in a format and they need not be separated from the other specifications by commas. During output, the slash may be used to skip lines, cards, or tape records. During input, it specifies that the next record is to be read. Format/list interaction when a slash is encountered is further explained in Section 9.7. K(/) results in K-1 lines being skipped, except at the end of the format in which case K lines are skipped.

Examples:

1) PRINT 10
10 FØRMAT (6X, 7HHEADING/ / /3X, 5HINPUT, 2X, 6HØUTPUT)

Printout:

HEADING		line 1
	(blank)	line 2
	(blank)	line 3
INPUTbbØUTPUT		line 4

Each line corresponds to a record. The second and third records are null and produce the line spacing illustrated.

2) PRINT 11, A, B, C, D
11 FØRMAT (1X,2E11.3/1X,2F7.3)

In storage:

A -11.6
B .325
C 46.327
D -14.261

Printout:

b-0.116E+02bb0.325E+00
b46.327-14.261

---

* Note that the first character of each record is interpreted by the printer as a carriage control character and is not printed. Unless provision is made for this character in the FØRMAT statement, e.g., using a wX or wH specification, information placed in this position is lost and unexpected spacing of the output may result.

```
3)          PRINT 11,A,B,C,D
      11    FØRMAT(1X,2E11.3/1HO,2F7.3/)
```

Printout:

```
 b-0.116E+02bb0.325E+00          line 1
                                 line 2
                    -(blank) -
 b46.327-14.261                  line 3
                                 line 4
                    -(blank) -
```

Line 2 on the printout above is blank because the zero in the first position of second output record (which actually prints as line 3) causes a double space before printing of the record.*

```
4)        PRINT 15,(A(I),I=1,9)
      15   FØRMAT (8HbRESULTS,2(/) (4X,3F8.2))
```

Printout:

```
     RESULTS                              line 1
                                          line 2
                        -(blank)-
           3.62   -4.03   -9.78          line 3
          -6.33    7.12    3.49          line 4
           6.21   -6.74   -1.18          line 5
```

9.5.5 Tabulation

Tn

The T (tab) specification is available for column selection control. When Tn is used, the format pointer is skipped to column n of the external record, and the next format specification is then processed. n may be any unsigned integer $\leq$ 132. If n=0, column 1 is assumed.

Tn Input (tab)

On input records, Tn enables data to be skipped or to be read in any desired order. If n > 80 when reading data from cards, the column pointer is moved to column n but a succeeding specification would read only blanks.

* The first character of an output record is interpreted by the printer as a carriage control and is not printed. See Section 10.1.

Example:

Input list: A,J,K

Format specification: (E7.2,T20,I2,T15,I4)

```
                 1     7  10   15   20
Input card:     /bb34.21bbb37b1b572b4b
                 |
```

In storage: A contains 34.21
J contains 40
K contains 572

Tn Output (tab)

On output, T enables information to be placed anywhere on the record in any order desired. The output line image is blanked prior to the actual formulation of a line.

Examples:

1. Format specification:
(T80,*COMMENTS*,T60,*HEADING4*T40*HEADING3*T20,*HEADING2*,T2,
HEADING1)

Printed output (numbers indicate print positions)*

```
1          19         39         59         79
|          |          |          |          |
HEADING1   HEADING2   HEADING3   HEADING4   COMMENTS
```

2. Output list: K,L,M

Format specification: (T20,I4,T36,I2,T28,I4)

In storage: K 372
L 0
M 4499

```
Output record:                        19       27       35
                                      |        |        |
                bbbbbbbbbbbbbbbbbbb372bbbb4499bbbbb0
```

9.5.6 Zero Suppression

Z Output

The appearance of a Z in an output specification acts as a switch causing leading zeros to be printed for all succeeding integer output quantities. The appearance of -Z turns the switch off, and normal output resumes. This switch is not automatically reset when a format is rescanned.

The compiler may give a non significant warning diagnostic when the Z specification is used.

---

* The first character position of a printed line is always used for carriage control.

Example:

In storage:

```
R    -632.43
I    32
J    -3762
X    37.95
K    4
```

Output list: R,I,J,X,K

Format specification: (1Hb,F7.2,Z,I4,I6,F8.2,-Z,I3)

Output: -632.430032-03762 37.95 4

9.6 REPEATED FORMAT SPECIFICATIONS

Format specifications may be repeated by using an unsigned integer constant, k (called a repetition factor), as follows: k spec or k(specs), where spec is any conversion
and specs is any set of format specifications. If k is omitted, it is assumed to be 1. For example, to print two quantities K,L:

```
      PRINT 10,K,L
10    FØRMAT(I2,I2)
```

Specifications for K,L are identical; the FØRMAT statement may also be:

```
10    FØRMAT(2I2)
```

When a group of format specifications repeats itself as in: FØRMAT(E15.3,F6.1,I4,I4,E15.3,F6.1,I4,I4), the use of k produces: FØRMAT(2(E15.3,F6.1,2I4)). Two levels of parentheses are allowed in addition to the outer parentheses delimiting the format specification.

Examples:

```
FØRMAT(3(4HbRØW,)*bYØURbBØAT*)
FØRMAT(2(1X,I5,3(F8.5,2X)))
FØRMAT(1H0,5I10/(10F10.5))
```

9.6.1 Unlimited Groups

If the end of the format specification is encountered and the input/output list is not exhausted, a new record is started. The format scan reverts to that parenthetical group terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the FØRMAT statement.

For example, in the output format:

```
 22  FØRMAT(10HORØWbNO.b=I2/(3X,6(I10,2Hb=F10.4)))
```

the group denoted by the arrows is the group repeated and is used to specify the format of output records 2,3,... if there are more than 13 output list elements.

Example:

```
     DIMENSIØN X(4,4)
        :
        :
     PRINT 100,(I,I=1,4),(J,(X(J,K),K=1,4),J=1,4)
100  FØRMAT(15X,*ARRAYbX*/1X,*CØLUMNb*,4I6/(1X,*RØW*,I2,2X,4F6.1))
```

produces the following printout:

```
                ARRAY X
CØLUMN      1      2      3      4
RØW 1     1.1    6.2    7.6    4.9
RØW 2     2.3    8.9    6.7    2.6
RØW 3     4.6    1.5    4.3    9.5
RØW 4     7.8    2.3    5.7    1.4
```

Note that if a scale factor has been used, the last scale factor in effect will be used if the format is repeated, thus possibly changing the conversion for elements which appeared before the first scale factor in the format specification.

Example:

```
FØRMAT(F10.5/(5E15.8,-2PF9.6))
```

If the list is not exhausted at the end of the format, the format will repeat starting with the 5E15.8 specification but with a scale factor of -2.

9.7 FORMAT AND LIST INTERACTION

When processing of a coded input/output statement is begun, a scan of the referenced FØRMAT statement is initiated. Each action then depends upon information jointly provided by the next element of the input/output list and the next specification obtained from the format.

When a READ statement is executed, the next record is read immediately. Additional records are read only as the format specifications demand. Any unprocessed characters of the current record will be skipped at the time of termination or when a slash is encountered.

When a WRITE or PRINT statement is executed, writing of the next record begins. Additional records are written as the format specifications demand.

Whenever a new record is required (as when a slash is encountered) processing of the new record begins with the first character position of the record, proceeding from left to right.

Except for the effect of repetition factors, the format is interpreted from left to right. Except for implied DO loops, the input/output list is also interpreted from left to right. To each conversion specification interpreted in the format, there corresponds one element in the input/output list, except that a complex element requires the interpretation of two F, E, or G specifications. (The conversion specifications, E, G, D, F, I, Ø, L, A, R, are described in section 9.3.)

Conversions are made as specified (if legal), regardless of the type designation of the list element.

No list item is associated with an editing specification (see Section 9.5). Whenever an editing specification is encountered in the format, it is executed. When a conversion specification is encountered, the list is examined first. If there is a list element, the indicated conversion is made. If no list element is found, the input/output statement is terminated.

Whenever the format scan reaches the end of the format, the input/output statement is terminated if the list is found to be exhausted. If not, a new record is started and the format scan proceeds as described in Unlimited Groups, Section 9.6.1.

9.8 VARIABLE FORMAT

Format specifications may be specified at the time of program execution. The specifications, including left and right parentheses but not the statement label nor the word 'FORMAT', may be read with A conversion, generated by an ENCODE statement (Section 10.6), or defined in a DATA statement and stored in an array. The name of the array containing the specifications is used in place of the FORMAT statement label in the associated input/output operation. The array name that appears, with or without a subscript, specifies the location of the first word of the FØRMAT information.

A variable format may be modified by replacing individual elements in the array containing it with the appropriate Hollerith constant.

Examples:

1. Assume the following format specifications:

```
(1X,E12.2,F8.2,I7,2E20.3,F9.3)
```

This information can be punched in an input card and read by the statements of the program such as:

```
      DIMENSIØN IVAR(3)
      READ 1, (IVAR(I),I=1,3)
    1 FØRMAT (3A10)
```

The elements of the input card are placed in storage as follows:

```
IVAR(1):   (1X,E12.2,
IVAR(2):   F8.2,I7,2E
IVAR(3):   20.3,F9.3)
```

A subsequent output statement in the same program can refer to these format specifications as:

```
PRINT IVAR, A,B,I,C,D,E
```

This produces exactly the same result as the statements:

```
      PRINT 10, A,B,I,C,D,E
   10 FØRMAT (1X,E12.2,F8.2,I7,2E20.3,F9.3)
```

2.
```
      DIMENSIØN LAIS1(3),LAIS2(3),A(6),LSN(3),TEMP(3)
      DATA LAIS1/25H(1HO,2F6.3,I7,2E12.2,3I1)/,LAIS2/
     1  23H(1X,I6,6X,3F4.1,2E12.2)/
```

Output statement:

```
      PRINT LAIS1,(A(I),I=1,2),K,B,C,(LSN(J),J=1,3)
```

which is the same as:

```
      PRINT 1,(A(I),I=1,2),K,B,C,(LSN(J),J=1,3)
    1  FØRMAT(1HO,2F6.3,I7,2E12.2,3I1)
```

Output statement:

```
      PRINT LAIS2,LA,(A(M),M=3,4),A(6),(TEMP(I),I=2,3)
```

which is the same as:

```
      PRINT 2,LA,(A(M),M=3,4),A(6),(TEMP(L),L=2,3)
    2  FØRMAT (1X,I6,6X,3F4.1,2E12.2)
```

3.
```
      DIMENSIØN LAIS(3), VALUE(8)
      DATA LAIS/27H(I3,14HbMEANbVALUEbIS,F6.3)/
```

Output statement:

```
      WRITE (10,LAIS)NUM,VALUE(6)
```

which is the same as:

```
      WRITE (10,10)NUM,VALUE(6)
   10 FØRMAT (I3,14HbMEANbVALUEbIS,F6.3)
```

9.9 NAMELIST STATEMENT

Variables placed in a NAMELIST statement may be converted on input or output without the need for format specifications. Instead the input record contains the names of the variables to be read and the values the variables are to be set to. The output record contains the names of the variables (as seen in the program) and the values of the variables converted to BCD.

The form of the NAMELIST statement is:

NAMELIST $/y_1/a_1/y_2/a_2/.../y_i/a_i/.../y_n/a_n$

y_i is an identifier of 1-7 characters.

a_i is a list of the form $b_1,b_2,...b_i,...,b_m$ where b_i is a variable or an array name.

Each y is a NAMELIST name; it must be different from all other names in the program. After it is defined, the name may appear only in READ or WRITE statements as described in Chapter 10. A NAMELIST name may be defined only once in a subprogram.

In any given NAMELIST statement, the list, a, of variable names or array names between the NAMELIST identifier, y, and the next NAMELIST identifier (or the end of the statement if no NAMELIST identifier follows) is associated with the identifier, y.

Examples:

```
PRØGRAM MAIN
NAMELIST/NAME1/N1,N2,R1,R2/NAME2/N3,R3,N4,N1

SUBRØUTINE XTRACT(A,B,C)
NAMELIST/CALL1/L1,L2,L3/CALL2/L3,P4,L5
```

A variable name or array name may be an element of more than one such list. In a subprogram, b_i may be a formal parameter identifying a variable or an array, but if it is an array, it may not have variable dimensions.

Chapter 10 describes the format of the input and output records and their interaction with the corresponding READ and WRITE statements.

The following definitions apply to all I/O statements:

i logical I/O unit number:

an integer constant of one or two digits (the first must not be zero) or

an integer variable with value from 0 to 99.

n FØRMAT declaration identifier:

statement number or

dimensioned variable identifier (with or without subscripts) which references the starting storage location of format information

L the input/output list (Section 9.1)

x namelist identifier (Section 9.9)

Logical I/O unit numbers do not have any predetermined mode (coded or binary) associated with them. The type of I/O statement determines the mode to be used. For each unit number i referenced by the program, there must be a file name TAPEi declared in the PRØGRAM statement.*

The filesets INPUT, OUTPUT, and PUNCH are needed for the READ n, PRINT n, and PUNCH n I/O statements, respectively. Other I/O statements may manipulate these filesets (except that fileset INPUT may not be written). This is done by using a logical unit number whose file name is equivalenced to the desired file.*†

There are two modes of reading or writing information on a fileset: binary and coded. The way in which the information is organized depends upon the mode as well as the I/O device on which the fileset is recorded.* Mixed mode filesets, i.e., filesets written partly in binary mode and partly in coded mode, should not be used.

Coded Mode

The information in Fortran coded mode files is organized into unit records. For the card reader or punch, a unit record is an 80-character card image. For the printer 133 characters (1 print line) constitutes a unit record. A unit record on magnetic tape contains up to 136 characters written in even parity.**

---

* Chapter 13 contains a further discussion of filesets, file names, physical and logical records.

† Section 7.4 shows how filenames are equivalenced. The footnote to Section 10.4 shows an example of equivalencing one of these files.

** The parity of a magnetic tape is the means by which the reading and writing of the information on tape is checked for validity.

Binary mode

A binary record on punched cards consists of all cards between the beginning of the deck and the first EØR (a card with 7-8-9 punched in column 1) or between consecutive EØRs. A binary record on magnetic tape is written in odd parity.*

When no other I/O device is assigned to a fileset, the fileset is recorded on disk or ECS.

10.1 OUTPUT STATEMENTS

PRINT n,L

Information in the list (L) is transferred from the storage locations to the standard output fileset (OUTPUT) as line images, 133 characters or less per line in accordance with the FØRMAT declaration, n. The maximum physical record length is 133 characters, but the first character of each unit record (line) is not printed, as it is used for carriage control when printing. Each PRINT statement or each new record starts a new print line.

The CDC system line printer uses a vertical spacing of six lines per inch, so the standard eleven inch page has room for sixty-six lines. To allow reasonable margins printing normally runs from line six through line sixty-one for a total of fifty-six lines per page. The horizontal spacing is ten characters per inch. Printers at remote stations do not necessarily follow the same conventions. The IBM printers use a vertical spacing of eight lines per inch. The following table will correspond to the IBM line printer positioning if all line numbers have 1 added. Example: Control Character 9 causes a skip to line 62.

Control Character	Effect (all actions are taken prior to printing the line)
Blank	Single space
0	Double space
-	Triple space
+	Suppress space
1	Skip to line 7 (of next page in most cases)
2	Skip to next of lines 10, 37 (next half page)
3	Skip to next of lines 9, 27, 45 (next third page)
4	Skip to next of lines 8, 22, 36, 50 (next quarter page)
6	Skip to line 1 just after concave paper fold
7	Skip to line 1 just after convex paper fold
8	Skip to line 1
9	Skip to line 62
X	Single space and enter automatic skip suppress mode (passing line 62 will not cause a skip to line 7); this mode continues until the mode is left
Y	Single space and leave automatic skip suppress mode.
Z	Single space but suppress automatic skip for this line

The "convex" fold in computer output is at the bottom of the output when folded normally and viewed by the prospective reader. It is the fold which is convex with respect to the viewer. "Concave" refers to a fold in the opposite direction

An unrecognized control character will be treated as a blank.

*See third footnote on previous page and the WRITE(i)L statement in following page.

PUNCH n,L

Information is transferred from the storage locations given in the list (L) to the standard Hollerith punch fileset (PUNCH). Information is transferred as 80 character or less unit records in accordance with the FØRMAT declaration, n.

WRITE (i,n)L

This form transfers information from storage locations given by the list (L) to the specified output unit (i) according to the FØRMAT declaration (n). The number of words in the list and the FØRMAT declaration determine the number of records that are written as discussed in Chapter 9.

If the file is to be printed, the first character of each record is not printed but is used as a carriage control character. If the programmer fails to allow for a carriage control character, the first character of the output data is lost on the printed listing and erratic line skipping may result.

WRITE(i)L

This form transfers information from the storage locations give by the list (L) to a specified output unit (i) in binary mode. If L is omitted, the WRITE(i) statement acts as a do-nothing statement. See READ(i)L in Section 10.2.

Examples:

```
     DIMENSIØN FSTNME(2), FIRMNM(4)
     DIMENSIØN M(10,5), B(4000), AMAX(10), C(20,20), LSTNME(2)
     LØGICAL A(260)
     REAL ITMNØ
     INTEGER ACCT, TELNØ, SHPDTE
     WRITE(10) A,B
     DØ 5 I=1,10
5    WRITE(6) AMAX(I),(M(I,J),J=1,5)
     PRINT 51,(A(I),I=1,20)
51   FØRMAT(X23HTRUTHbMATRIXbVALUESbARE/(3X,4L3))
     PUNCH 52, ACCT, LSTNME,FSTNME,TELNØ,SHPDTE,ITMNØ
52   FØRMAT(I8,3X,4A10,2X,I10,1X,I5,F8.2)
     WRITE(2,53)B,C,D
53   FØRMAT(4E21.9)
     WRITE(2,52) ICØDE,FIRMNM,LØC
     WRITE(2,54)
54   FØRMAT(*bTHISbSTATEMENTbHASbNØbDATAbLIST.*)
```

WRITE (i,x)

Write coded information to unit i as follows:

1. One record consisting of a $ in column 2 immediately followed by the NAMELIST identifier, x.

2. As many records as are needed to contain the current values of all variables in the NAMELIST associated with x. Simple variables are written as v=c.

   Elements of dimensioned variables are written in the order in which they are stored internally. (Section 2.6.1).

   The data fields are made large enough to include all significant digits. Logical values appear as .T. and .F. No data appears in column 1 of any record.

3. One record consisting of a $ in column 2 immediately followed by the letters END.

The records written by such a WRITE statement may be read by a READ(i,x) statement where x is the same NAMELIST identifier.

10.2 READ STATEMENTS

READ n,L

One or more unit records are read* from the standard input file (i.e., fileset INPUT). Information is converted from left to right in accordance with format specification (n), and it is stored in the locations named by the list (L). Each new READ statement starts a new record.

Example:

```
      READ 10,A,B,C
10    FØRMAT (3F10.4)
```

---

* A program should not expect to be terminated for encountering an end-of-file (EOF), since provision is allowed in the language to test for an EOF, the IF(EOF,i) statement. Instead the job is terminated only when on a subsequent READ, the EOF indicator has not been turned off by the test statement. List items unsatisfied when the EOF is encountered will be set to minus zero in memory. For historical reasons, INPUT is inconsistent with all other filesets in that any EOR will be treated as if it were an EOF by a coded READ.

READ (i,n)L

This form transfers information from a specified unit (i) to storage locations named by the list (L), according to the format specification (n). The information is read in coded mode.

The number of words converted and records read is determined by the list and format specifications, which must conform to the record structure on the specified unit.* Each READ (i,n) statement starts a new record. See footnote to READ n,L.

READ (i)L

This form transfers one binary record of information from a specified unit (i) to the storage locations indicated by the list (L).

Records to be read by READ(i) should be written in binary mode. The number of words in the list of READ(i)L must not exceed the number of words in the corresponding WRITE statement.** If the list is longer than the record, the excess words in the list are left unchanged in memory. If the list is shorter than the record, the untransmitted words are skipped. Each READ(i) statement starts a new record.

Examples:

```
     DIMENSIØN C(264)
     DIMENSIØN BMAX(10), M2(10,5), A(100,50)
     DØUBLE PRECISIØN DB(4)
     DIMENSIØN Z(8)
     READ (10)C
     DØ 7 I=1,10
7    READ(6) BMAX(I), (M2(I,J),J=1,5)
     READ(5)                (skip one logical record on unit 5)
     READ(6) ((A(I,J),I=1,100),J=1,50)
     READ(10,50)X,Y,Z
50   FØRMAT (3F10.6)
     READ(10,51) DB
51   FØRMAT (4D20.12)
     READ 51, DB
     READ(2,52) (Z(J),J=1,8)
52   FØRMAT (F10.4)
```

---

* Attempting to use more than 136 characters of a unit record will normally terminate the job, but see Chapter 11: LNGBCD.

** See Chapter 11: the LENGTH function.

READ (i,x)

The current file of unit i is scanned until either an end-of-file or a record with a $ in column 2 immediately followed by the NAMELIST name, x, with no embedded blanks but followed by a blank is located. When the name, x, is encountered, succeeding data items are read until a $ is encountered. The data items are placed in the variables and arrays associated with the NAMELIST x. If no data block with the correct name is found, a fatal error exit occurs. If no data are to be read by this statement, a void block with the correct name should be used.

Data items are separated by commas. They may be in any of three forms:

$$v = c$$

$$a = d_1,\ldots,d_j$$

$$a(n) = d_1,\ldots,d_m$$

v is a variable name

c is a constant

a is an array name

n is an integer constant subscript

The d_i are simple constants or repeated constants of the form k*c, where k*c implies that the constant, c, is to be repeated k times.

Example: The statements:

```
PROGRAM SHWØRD(INPUT,ØUTPUT,TAPE7=INPUT)
DIMENSIØN Y(3,5)
LØGICAL L
CØMPLEX Z
NAMELIST/HURRY/I1,I2,I3,K,M,Y,Z,L
READ (7,HURRY)
```

and the card input records, starting in Col. 2:

```
$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,
12.0E1,13,4*14,Z=(1.,2.),
K=16, M=17$
```

produce the following values upon execution:

```
I1=1              Y(1,2)=14.0
I2=2              Y(2,2)=14.0
I3=3              Y(3,2)=14.0
Y(3,5)=26.0       Y(1,3)=14.0
Y(1,1)=11.0       K=16
Y(2,1)=120.0      M=17
Y(3,1)=13.0       Z=(1.,2.)
                  L=.TRUE.
```

The number of constants, including repetition, given for an unsubscripted array name must equal the number of elements in that array. For a subscripted array name, the number of constants need not equal, but may not exceed, the number of array elements which follow the specified element, plus one.

v=c	variable, v, is set to c
$a=d_1,\ldots,d_j$	the values, $d_1,\ldots,d_j$, are stored in consecutive elements of array, a, in the order in which the array is stored internally.
$a(n)=d_1,\ldots,d_m$	elements are filled consecutively starting at a(n)

The specified constant of the NAMELIST statement may be integer, real, double precision, complex of the form (c_1,c_2) or logical of the form .T., .TRUE.,.F., .FALSE. A logical or complex variable may be set only to a logical or complex constant, respectively. Any other variable may be set to an integer, real or double precision constant. Such a constant is converted to the type of its associated variable.

Constants and repeated constant fields may not include embedded blanks. Blanks, however, may appear elsewhere in data records.

A maximum of 120 characters per input record is permitted. More than one record may be used for input data. All except the last record must end with a constant followed by a comma, and no serial numbers may appear; the first column of each record is ignored.

The set of data items may consist of any subset of the variable names associated with x. These names need not appear in the same order in which they appear in the defining NAMELIST statement. Those variables not specified are left unchanged in memory.

10.3 FILE HANDLING STATEMENTS

REWIND i

Logical I/O unit i is rewound, i.e., positioned so that the next record to be processed is the first record on this unit. If the unit is already rewound, REWIND does nothing.

ENDFILE i

An end-of-file is written on unit i.

BACKSPACE i

Logical I/O unit i is backspaced one record, i.e., one unit record if unit i is formatted, one binary record if unit i is unformatted, one logical record if unit i is buffered. If unit i is rewound or has not been moved from its beginning by the Fortran program, the BACKSPACE does nothing. If end-of-file has just been passed, a BACKSPACE leaves the unit positioned immediately before the end-of-file. BACKSPACE should not be used on a fileset on which NAMELIST I/O is used.

10.4 File Status Testing Statements

IF (ENDFILE i)n_1,n_2

IF (EØF,i)n_1,n_2

IF (EØF(i))n_1,n_2,n_1

These statements check the previous read operation on unit i to determine if an end-of-file has been encountered.* If so, control is transferred to statement n_1; if not, control is transferred to statement n_2. If an end-of-file has been encountered, the end-of-file indicator for that unit is on and the execution of this statement turns it off. If the end-of-file indicator is not turned off for a unit, an attempt to read more information from the unit will terminate the job.

If (UNIT,i)n_1,n_2,n_3,n_4

- n_1 I/O in progress on unit i
- n_2 previous I/O complete on unit i, with no error
- n_3 EOF sensed on last input operation
- n_4 parity or lost data error on last input operation

This statement is used for testing the status of unit i when a BUFFER statement (Section 10.5) is used for input/output on unit i. When it is necessary to wait for an I/O operation to complete, execute

CALL XRCL

between IF(UNIT,i) tests to reduce central processor charges while waiting.

IF(IØCHECK,i)n_1,n_2

This statement is used to check for a parity error in the last record read on logical I/O unit i. If the error occurred control is transferred to statement n_1, otherwise control is transferred to statement n_2. If a parity error occurs and is not checked by this statement, a subsequent attempt to read from this unit will terminate execution with error code 53.

---

* An EØF on the standard input device, fileset INPUT, may be tested by equivalencing the filename INPUT to another filename on the PRØGRAM card (Section 7.4) Thus, PRØGRAM MAIN(INPUT,ØUTPUT,TAPE50=INPUT) and IF(EØF,50)20,30 would test for the end of input. Due to an historical quirk, an EØR from a simple 7-8-9 card on fileset INPUT (only) will test as an end-of-file. On all other files, EØR's are ignored in coded reads and only EØF's will be sensed by this test. If it is necessary to test for EØR on coded reads, see Chapter 11: IEØR.

10.5 BUFFER STATEMENTS

The BUFFER statements allow the programmer to control the use of I/O units and to buffer the information being sent to/from these units. These statements combined with the ENCØDE/DECØDE statements permit the FORTRAN programmer to construct complicated input/output schemes.

Because the operating system normally attempts to overlap I/O and execution, the Buffer statements frequently do not result in much improvement in I/O efficiency. Also, the use of Buffer statements is fraught with hazards due to their handling of the structure of logical records. For these reasons, use of the buffer statements in normal circumstances is not recommended.

The primary differences between buffer I/O and read/write I/O statements are given below:

1. The mode of transmission (coded or binary) is tacitly implied by the form of the read/write control statement. In a buffer control statement, mode must be specified by a mode indicator.

2. The READ/WRITE control statements are associated with a list and, in coded transmission, with a FØRMAT statement. The buffer control statements are not associated with a list; data transmission is to or from a single area in storage.

3. A buffer control statement initiates data transmission, and then returns control to the program, permitting the program to perform other tasks while data transmission is in progress. Before buffered data is used, the status of the buffer operation should be checked. (Difference: a READ or WRITE statement completes the operation before returning control to the program.)

The forms of the BUFFER statements are:

```
BUFFER IN(i,p)(fw,ℓw)
BUFFER ØUT(i,p)(fw,ℓw)
```

where

- i is a logical unit number.
- p is a mode key. May be specified by an integer constant or simple integer variable (not subscripted). 0 for coded mode; 1 for binary mode.
- fw is a variable identifier, the first word of a block of data to be transmitted.
- ℓw is a variable identifier, the last word of a block of data to be transmitted.

A BUFFER OUT statement with an lw address which is one less than the fw address will write a zero length logical record. In all other cases the address of lw must be greater than or equal to that of fw or an execution error message is given.

Examples:

```
DIMENSIØN A(100)
N=6
BUFFER ØUT(N,1)(A(1),a(100))

CØMMØN/BUFF/DATA(10),CAL(50)
MØDE=0
BUFFER IN(9,MØDE)(DATA(1),CAL(50))
```

BUFFER IN(i,p)(A,B)

Information is transmitted from unit i in mode p to storage locations A through B. Thus, in the second example above, coded information is read from unit 9 into labeled common area BUFF beginning at DATA(1), the first word of the block, and extending through CAL(50), the last word of the block.

The BUFFER IN statement issues a system request for input from fileset i in mode p into the given block. Execution continues in parallel with the physical input operation. The completion of the operation can be determined only by the IF(UNIT,i) statement (p. 10-8). Once the read operation is completed and the data transferred into the block, the amount of data actually read can be determined by the library function LENGTH(i), see Chapter 11, which returns the length (in C.M. words) of the entire record.* If the record read was longer than the given buffer, the excess words are counted, but skipped. If the record is shorter than the buffer, unused buffer words are not changed.

BUFFER ØUT (i,p)(A,B)

Information is transmitted from storage locations A through B and one logical record is written on unit i in mode p containing all the words from A to B inclusive. Thus in the first example above, binary information is transmitted to unit N from the block area defined by A(1) and A(100), i.e., all of array A is transmitted.

The BUFFER OUT statement issues a system request for output onto fileset i in mode p from the given block. Execution continues in parallel with the physical output operation. The completion of the operation can be determined only by the IF(UNIT,i) statement (see elsewhere in this section). Once the output operation is completed, the file can be referenced by other I/O statements. One logical record is written by the system for each BUFFER OUT request.

---

* A logical record even on a coded file. A coded logical record is not usually a unit record. See Chapter 13.

The following restrictions apply to the use of BUFFER IN/ BUFFER ØUT:

a. The statement IF(UNIT,i) is used to determine the completion of a BUFFER statement and to check for a file mark when using BUFFER IN. It must be used before any further I/O request of unit i is made (including a reference to the function LENGTH).

b. Other Fortran I/O statements referring to unit i, such as REWIND i, must not be used until the buffer operation is completed as indicated by the IF(UNIT,i) statement. Mixing READ/WRITE and BUFFER IN/BUFFER ØUT statements for the same I/O unit is not recommended.

c. BUFFER I/O statements process complete logical records. See Chapter 13 for a discussion of logical records.

10.6 INTERNAL DATA TRANSMISSION (ENCODE/DECODE)

Information may be transferred under a format specification from one area of central memory to another using the ENCODE/ DECODE* statements; no external devices are involved. These statements are quite slow, the central processor time required being about the same as an actual formatted I/O operation with the same format. They should be used only where the flexibility of the format control is essential or where they are executed relatively few times.

Uses of ENCODE/DECODE statements:

All information stored in the central memory of the computer is in the form of strings of binary digits (0's and 1's). However, two different binary representations may be used, depending upon whether the format specification used to read the data was a numeric conversion specification or an alphanumeric conversion specification (see Appendix A). For instance, consider the decimal integer 8 on a data card. If it is read into memory under an I format, it is stored in a 60-bit word in the form: 0-----01000_2. For ease of reference, it is said to be stored as 10 octal or 10B. If, on the other hand, the same card were read under an alphanumeric conversion specification, e.g., A1, it would appear in memory as a string of 60 binary digits of the form: 100011101101---101 (i.e., 435555--55B). This representation is called internal BCD and uses the 6-bit binary equivalents of the octal digits 01B to 32B to represent the 26 letters of the alphabet, 33B to 44B for the digits 0 to 9 and 45B to 77B for various special characters (see Appendix A).

*ENCODE and DECODE are non-standard and may not function within other Fortran systems.

The ENCODE/DECODE statements are available to permit the user to

1. rearrange and manipulate information stored in display code (e.g., to break up into several words a string of characters stored (packed) into one word, or to alter or fill in some part of a format specification stored in an array).

and

2. convert information from coded to binary (DECODE) or vice versa (ENCODE).

NOTE: All integer values are stored as binary information regardless of whether they are input under O or I conversion specification, which dictate only what conversion is to occur before the values are stored. Thus, if 11 is read under O conversion into I, it is stored as an 11B, whereas if 11 is read under I conversion into II, it is stored as 13B. When these two values are used in arithmetic expressions in a program, octal (i.e., binary) arithmetic is performed and proper results are generated; that is, I = I + II will store 24B (=20) into I, which is the correct value.

These statements have the following form:

ENCODE (c,f,v)ℓ
DECODE (c,f,v)ℓ

where

c is an unsigned integer constant or a simple integer variable (not subscripted) specifying the number of characters in the record. Under RUN c may be up to 150 BCD characters (c > 150 is an error which is caught at compilation time if c is an integer constant and at execution time if c is an integer variable).

f is the statement number or variable identifier identifying the format specification,

v is a variable or array name (with or without subscripts) specifying the starting location of the BCD information,

ℓ is the input/output list.

Information transfer into or out of the area specified by v begins with the leftmost character position of the location given by v and continues until c characters (10 per word) have been transferred [i.e., read (DECODE)

or stored (ENCODE)]. For ENCODE if c is not a multiple of 10, the record ends in the middle of a word and the remainder of the word is blank filled. For DECODE, if the record ends with a partial word, the balance of the word is ignored.

Since each succeeding record begins a new computer word, the number of words allocated for each record is (c + 9)/10 truncated if necessary to the nearest integer.

ENCODE

The list of variables, ℓ, is transmitted according to the format specification identified by f and stored in successive locations starting at v with c internal BCD characters per record. If c is not a multiple of 10, the record ends in the midst of a word and the remainder of the word is blank filled. If the I/O list ℓ and the format specification translate more than c characters per record, an execution error message is generated.

Examples:

1. ENCODE may be used to calculate a field definition in a format specification at execution time. Assume that the user wishes to have a format specification of the form (2A10,Im) but wants to specify a value for m at some point during the execution of the program, subject to the restriction $2 \leq m \leq 9$. The following statements allow m to vary.

```
      .
      .
      .
      IF(M.LT.10 .AND. M .GT. 1)1,2
    1 ENCODE(9,100,SPECMAT)M
  100 FORMAT (7H(2A10,I,I1,1H))
      .
      .
      .
      PRINT SPECMAT,A,B,J
      .
      .
      .
    2 PRINT 10
   10 FORMAT (*bMbOUTbOFbRANGE)
```

M is tested to insure it is in the proper range. If not, control transfers to the statement labeled 2, which in this case prints a message. If M is in the proper range, ENCODE packs the coded equivalent of the value of M into the character string forming the format specification in SPECMAT. For instance if M were 5, SPECMAT would contain (2A10,I5)b (in display code:51350134335611405255).

```
      .
      .
      A(1) = 10HABCDEFGHIJ
      A(2) = 10HKLMNOPQRST
      B(1) = 10HPQRSTUVWXY
      B(2) = 10HZ123456789
      .
```

2. (C = multiple of 10)

```
      DIMENSION ALPHA(4)
      .
      .
      .
      ENCODE (20,1,ALPHA)A,B
    1   FORMAT(A10,A5/A10,A10)
      .
      .
      .
```

Result: (1st record)

```
      ALPHA(1) = ABCDEFGHIJ
      ALPHA(2) = KLMNObbbbb
```

(2nd record)

```
      ALPHA(3) = PQRSTUVWXY
      ALPHA(4) = Z123456789
```

3. (c ≠ multiple of 10)

```
      DIMENSION ALPHA(10)
      .
      .
      .
      ENCODE(16,1,ALPHA)A,B
    1   FORMAT(A10,A6)
```

```
      1st record
      ABCDEFGHIJKLMNOPbbbb
      └─────────┘└─────────┘
      ALPHA(1)   ALPHA(2)

                          2nd record
                    PQRSTUVWXYZ12345bbbb
                    └─────────┘└─────────┘
                     ALPHA(3)   ALPHA(4)
```

DECODE

The information in c consecutive internal BCD characters (starting at address v) is transmitted according to the format specification indicated by f and stored in the list variables ℓ. If a record ends with a partial word, the balance of the word is ignored. However, if the specification f is greater than c (record length) per record, an execution error message is given. Attempting to DECODE an illegal format code or a character illegal under a given conversion specification gives an execution error message.

If a binary zero (six zero bits) is encountered anywhere in a record, it is treated as a blank. If two binary zeros occur consecutively (12 zero bits) in the processing of a record, the remaining characters to be transmitted in the record, if any, are converted to blanks; the next record begins at the word following the word containing the two binary zeros.

Examples:

1.

```
      .
      .
     DIMENSION GAMMA(4),A6(2),B6(2)
     GAMMA(1)=10HHEADERb121   }   1st record
     GAMMA(2)=10HHEADbb0131   }
     GAMMA(3)=10HHEADERb122   }   2nd record
     GAMMA(4)=10HHEADbb0231   }
     DECODE(18,1,GAMMA)A6,B6
   1 FORMAT(A10,A8)
```

```
Result:  A6(1) = HEADERb121
         A6(2) = HEADbb01bb
         B6(1) = HEADERb122
         B6(2) = HEADbb02bb
```

2. DECODE will be used in this example to break the DATE into three integer words containing the MONTH, DAY and YEAR.

```
      .
      .
      .
     INTEGER DATE, MONTH, DAY, YEAR
     DATE = 10Hb01/08/70b
     DECODE(10,205,DATE) MONTH, DAY, YEAR
 205 FORMAT(3(1X,I2))
```

After execution of the DECODE statement, the contents of the integer words will be:

MONTH = 1 (or 0 ----- 01B)
DAY = 8 (or 0 ----- 10B)
YEAR = 70 (0 ----- 106B)

The internal BCD contents of DATE = 553334503343504233;

The above statements decode the number values representing the date by executing the conversion under the format 1X,I2, three times; i.e., taking the next two BCD characters (01 = 3334) and converting them to an integer, skipping the next character (/ = 50B), converting the next two BCD characters to the equivalent integer number (08 = 3343), and finally skipping the next character (/ = 50B), and converting the last two BCD characters 4233 (= 70) to the equivalent integer.

3. The following illustrates one method of packing the partial contents of two words into one word. Information is stored in LOC(1) and LOC(6) as:

```
LOC(1) = SSSSSXXXXX
       (10 BCD characters/word)
LOC(6) = XXXXXDDDDD
```

to form a word of the form: SSSSSDDDDD in storage location NAME:

```
      :
      :
      DECODE(10,1,LOC(6)) TEMP
    1 FORMAT(5X,A5)
      ENCODE(10,2,NAME)  LOC(1), TEMP
    2 FORMAT(2A5)
```

The DECODE statement places the last 5 BCD characters of LOC(6) into the first 5 characters of TEMP. The ENCODE statement then packs the first 6 characters of LOC(6) and of TEMP into NAME.

With the R specification, the statement may be shortened to:

```
      :
      :
      ENCODE(10,1,NAME)LOC(1),LOC(6)
    1 FORMAT (A5,R5)
```

PREDEFINED FUNCTIONS AND SUBROUTINES 11

In addition to the subprograms which the user may define for himself, many functions and subroutines are automatically available to the Fortran programmer, either through the automatic generation of the required code by the compiler, or through the standard subprogram library (fileset SUBRLIB) which is supplied as a part of the operating system.

Additional specialized libraries such as the Graphical Display System library (fileset GDSLIB) and the mathematical subprogram library (fileset MATHLIB) exist, and may also be searched for required subprograms at the user's option (see CLDR in CALIDOSCOPE Control Statements).

The intrinsic (in line code) function names are reserved (i.e., cannot be used to denote a program name, subscripted variable or another subprogram) unless the name occurs in a DIMENSION or EXTERNAL statement, is redefined by an arithmetic statement function, or is used without an argument list. Of course, it is then unavailable as an intrinsic function within that subprogram. If a type statement which changes the type associated with an intrinsic function name appears in a program unit, the intrinsic function is superseded.

The table in the following section describes the supported functions and subroutines available. It does not describe those library subprograms which are called implicitly, e.g., for the execution of a PRINT statement, or only by other library subprograms. In the column showing the form of the references, the variables used as arguments and to receive function values are coded to indicate type as follows (appended numbers indicate argument positions):

B,B1,B2,...	are bit strings (type is irrelevant)
C,C1,C2,...	are complex numbers
D,D1,D2,...	are double precision (D.P.) numbers
H,H1,H2,...	are Hollerith (character string) data
I,I1,I2,...	are integers
L,L1,L2,...	are logical values
R,R1,R2,...	are real numbers
S1,S2,...	are external subroutines

If an array is expected as an argument, an A is inserted following the type code letter. The dimension of the array may be indicated in parentheses, e.g., IA1(5) means first argument is an integer array of dimension five.

If the argument is an external function, an F is inserted following the type code letter.

If there is an option as to argument type, acceptable types appear separated by slashes (e.g., H1/I1 for Hollerith or integer as first argument).

Variable length argument lists are indicated by the minimum argument list followed by an ellipsis.

Deck names are listed only when they differ from the reference (entry point) names. An "(I)" under deck name indicates an intrinsic function.

References may refer to library writeups (e.g., K2 CAL REGISTR), sections of this manual (e.g., 8.2.3), error numbers discussed in chapter 15 (e.g., 62), or to the American National Standards Institute Fortran Standard (ANSI). Functions marked ANSI are available in every Fortran IV system which meets the standard.

The arguments of the trigonometric functions are expressed in radians, as are the results of the inverse trigonometric functions. Multi-valued complex functions calculate only a single value.

Examples

1. R=AMIN0(I1,I2,...) Minimum of Arguments (I) ANSI

   indicates that the function AMIN0 has a variable number of arguments (at least two) which are integers and returns a real result. The function is defined to have a value which is equal to the mininum of the arguments with which it is called. The "(I)" in the Deck Name column indicates that AMIN0 is compiled in line so there is no separate deck for it. The "ANSI" in the Reference column indicates that AMIN0 is a function found in all standard Fortran implementations.

2. CALL OVERLAY(H1/I1,I2,I3,H4) Load and Execute Program Overlay 8.4

   indicates that OVERLAY is a subroutine with four arguments, the first being either Hollerith or integer, the second and third being integers and the fourth being Hollerith. OVERLAY is the library routine used to execute overlays. A detailed description is found in section 8.4 of this manual.

11.1 Table of Fortran Functions and Subroutines

Form of Use	Definition	Deck Name	Reference
CALL ABORT(H1)	Terminate Job Step with Error Status		11.2.22
R = ABS(R1)	Absolute Value: |R1|	(I)	ANSI
R = ACOS(R1)	Arcosine	ASINCOS	
R = AIMAG(C1)	Imaginary Part of Complex	(I)	ANSI
R = AINT(R1)	Integer Part of Real	(I)	ANSI, 11.2.27
R = ALOG(R1)	Natural Logarithm	ALNLOG	ANSI
R = ALOG10(R1)	Common Logarithm	ALNLOG	ANSI
R = AMAX0(I1,I2,...)	Maximum of Arguments	(I)	ANSI
R = AMAX1(R1,R2,...)	Maximum of Arguments	(I)	ANSI
R = AMIN0(I1,I2,...)	Minimum of Arguments	(I)	ANSI
R = AMIN1(R1,R2,...)	Mininum of Arguments	(I)	ANSI
R = AMOD(R1,R2)	R1 modulo R2	(I)	ANSI, 11.2.13
L = AND(B1,B2,...)	Boolean Product	(I)	11.2.2
R = ASIN(R1)	Arcsine	ASINCOS	
R = ATAN(R1)	Arctangent (R1)		ANSI
R = ATAN2(R1,R2)	Arctangent (R1/R2)		ANSI
CALL BLOK(H1/I1)	Force Blocked Binary on I/O Unit	NOBLOK	11.2.1

Form of Use	Definition	Deck Name	Reference
R = CABS(C1)	Absolute value		ANSI
C = CCOS(C1)	Complex Cosine		ANSI
C = CEXP(C1)	Complex Exponential		ANSI
CALL CLDISK	Close Random Access Fileset	TSDISK	I9 CAL TSDISK
C = CLOG(C1)	Complex Natural Logarithm		ANSI
C = CMPLX(R1,R2)	Complex R1+iR2 from Parts	(I)	ANSI
L = COMPL(B1)	Complement (Logical Negation)	(I)	11.2.2
C = CONJG(C1)	Complex Conjugate	(I)	ANSI
R = COS(R1)	Cosine	SINCOS	ANSI
C = CSIN(C1)	Complex Sine		ANSI
C = CSQRT(C1)	Complex Square Root		ANSI
D = DABS(D1)	D.P. Absolute Value: |D1|		ANSI, 11.2.26
D = DATAN(D1)	D.P. Arctangent (D1)		ANSI
D = DATAN2(D1,D2)	D.P. Arctangent (D1/D2)	DATAN	ANSI
D = DBLE(R1)	Conversion: Real to Double Precision		ANSI, 11.2.4
D = DCOS(D1)	D.P. Cosine	DSINCOS	ANSI
D = DEXP(D1)	D.P. Exponential		ANSI
R = DIM(R1,R2)	Positive Difference: R1-min(R1,R2)	(I)	ANSI

Form of Use	Definition	Deck Name	Reference
CALL DISPLA(H1,I2/R2)	Display Name and Value on Job Log		11.2.5
D = DLOG(D1)	D.P. Natural Logarithm	DLNLOG	ANSI
D = DLOG10(D1)	D.P. Common Logarithm	DLNLOG	ANSI
D = DMAX1(D1,D2,...)	D.P. Maximum of Arguments	(I)	ANSI
D = DMIN1(D1,D2,...)	D.P. Minimum of Arguments	(I)	ANSI
D = DMOD(D1,D2)	D.P. D1 Modulo D2		ANSI, 11.2.13
D = D$IGN(D1,D2)	D.P. |D1| with sign of D2		ANSI
D = DSIN(D1)	D.P. Sine	DSINCOS	ANSI
D = D$QRT(D1)	D.P. Square Root		ANSI
CALL DUMP(B1,B2,I3,...)	Dump Memory and Terminate Execution		11.2.6
CALL DUMPREG	Print CPU Register Contents		Q4 CAL REGDUMP
CALL DVCHK(I1)	For compatibility only, use LEGVAR		11.2.15
IF(EOF(H1/I1))	EOF≠0 means End of File on I/O Unit		11.2.7
CALL EXIT	Terminate Execution Normally		
R = EXP(R1)	Exponential: e to the R1 Power		ANSI
CALL FDEBUG(H1,...)	Program Tracing Control		N1 CAL DEBUG
R = FLOAT(I1)	Conversion: Integer to Real	(I)	ANSI
CALL GETCJE	Fetch Current Job Environment to /CJE/		Z1 CAL GETCJE

Form of Use	Definition	Deck Name	Reference
CALL GETREG(BA1(8))	Fetch System Communication Registers	REGISTR	K2 CAL REGISTR
I = IABS(I1)	Absolute Value: |I1|	(I)	ANSI
I = IDIM(I1,I2)	Positive Difference: I1-min(I1,I2)	(I)	ANSI
I = IDINT(D1)	Integer Part of D.P. Number		ANSI,11.2.26, 11.2.27
I = IEOI(H1/I1)	I>0 means EOI on I/O Unit		11.2.8
I = IEOR(H1/I1)	End of Record Level on I/O Unit		I0 CAL IEOR
I = IFIX(R1)	Integer Part of Real Number	(I)	ANSI, 11.2.27
IF(INDVCEX(H1/I1))	Check for Input Device Capacity Exceeded		11.2.9
I = INT(R1)	Integer Part of Real Number	(I)	ANSI, 11.2.27
I = ISIGN(I1,I2)	|I1| with Sign of I2	(I)	ANSI
L = KOMMON(H1,H2,I3,I4)	Control Fileset Common/Local Status		Q4 CAL KOMMON
I = LEFT(B1,I2)	Nominal Left Shift B1 by I2 Bits		M2 CAL LEFT
I = LEGVAR(R1)	Legitimacy of Real Variable		11.2.10
I = LENGTH(H1/I1)	Length of Last Record Read on I/O Unit		11.2.11
CALL LNGBCD(I1,BA2(I1))	Extend Formatted I/O Unit Record Size		I4 CAL LNGBCD
I = LOCF(B1)	Location of Argument: Machine Address		
I = LRDISK(I1)	Length of Record on Random Access File	TSDISK	I9 CAL TSDISK

Form of Use	Definition	Deck Name	Reference
I = MAX0(I1,I2,...)	Maximum of Arguments	(I)	ANSI
I = MAX1(R1,R2,...)	Maximum of Arguments	(I)	ANSI
CALL MCLOCK(H1)	Fetch Current Time of Day	MTIME	11.2.23, Z2 CAL MTIME
CALL MDATE(H1)	Fetch Current Date	MTIME	11.2.24 Z2 CAL MTIME
CALL MEMORY(H1,I2,I3)	Control ECS and Central Memory Fields		11.2.12, Q4 CAL MEMORY
I = MILSEC(B1)	Milliseconds of CP Time Used in Job	SECOND	Z1 CAL SECOND
I = MIN0(I1,I2,...)	Minimum of Arguments	(I)	ANSI
I = MIN1(R1,R2,...)	Mininum of Arguments	(I)	ANSI
I = MOD(I1,I2)	I1 modulo I2	(I)	ANSI, 11.2.13
CALL MTDISK	Empty Random Access Fileset	TSDISK	I9 CAL TSDISK
L = MZERO(B1)	Test for Minus Zero		11.2.25
CALL NARG(I1)	Return Current Argument Count		Z9 CAL NARG
CALL NMDISK(H1)	Set Random Access Fileset Name	TSDISK	I9 CAL TSDISK
CALL NOBLOK(H1/I1)	Suppress Blocked Binary on I/O Unit		11.2.14
CALL OPDISK(I1,IA2(I1))	Open Random Access Fileset	TSDISK	I9 CAL TSDISK
L = OR(B1,B2,...)	Boolean Sum of Arguments	(I)	11.2.2

Form of Use	Definition	Deck Name	Reference
CALL OVERFL(I1)	For compatibility only, use LEGVAR		11.2.15
CALL OVERLAY(H1/I1,I2,I3,H4)	Load and Execute Program Overlay		8.4
CALL PDUMP(B1,B2,I3,...)	Dump Memory and Proceed		11.2.6
CALL PUTREG(BA1(8))	Store System Communication Registers	REGISTR	K2 CAL REGISTR
R = RANF(R1)	Pseudo Random Number Generator		11.2.16
CALL RDDISK(I1,BA2(I3),I3)	Read Random Access Fileset	TSDISK	I9 CAL TSDISK
R = REAL(C1)	Real Part of Complex	(I)	ANSI
CALL RECS(BA1,I2,I3)	Read Extended Core Storage	ECSIO	K2 CAL ECSIO
CALL REMARK(H1)	Insert Remark in Job Log		11.2.17
CALL RERECS(BA1,I2,I3)	Reread Extended Core Storage	ECSIO	K2 CAL ECSIO
CALL RETURN(H1/I1)	Return Assigned I/O Device	REWINM	11.2.18
CALL SECOND(R1)	Seconds of CP Time Used in Job		Z1 CAL SECOND
CALL SETFXB(H1/I1,I2,I3,I4)	Set Fixed Block Logical Record Size	SETPRU	I4 CAL SETFXB
CALL SETPRU(H1/I1,I2)	Set Physical Record Size Limit		I4 CAL SETPRU
CALL SETRDCT(I1/H1,I2)	Set Read Error Recovery for Magnetic Tape		11.2.21
R = SIGN(R1,R2)	|R1| with Sign of R2	(I)	ANSI
R = SIN(R1)	Sine	SINCOS	ANSI
CALL SLITE(I1)	Simulate Sense Light		11.2.3

Form of Use	Definition	Deck Name	Reference
CALL SLITET(I1,I2)	Reset and Test Simulated Sense Light		11.2.3
R = SNGL(D1)	Truncate D.P. Value to Real		ANSI, 11.2.26
CALL SORTR(BA1,I2,...)	Radix Exchange Method Sort	TSORTR	M1 CAL SORTR
R = SQRT(R1)	Square Root		ANSI
CALL SSWTCH(I1,I2)	Test Simulated Sense Switch		11.2.3
CALL $START	For compatibility only, use REMARK		11.2.15
CALL SYSTEMP(B1,B2,B3,B4,B5,B6,I7,H8)	Diagnostic Print with Traceback		11.2.19
R = TAN(R1)	Tangent		
R = TANH(R1)	Hyperbolic Tangent		ANSI
CALL TIME	For compatibility only, use REMARK		11.2.15
CALL TRAILB(H1/I1,I2)	Control Trailing Blanks in Coded Output		J4 CAL TRAILB
CALL WECS(BA1,I2,I3)	Write Extended Core Storage	ECSIO	K2 CAL ECSIO
CALL WRDISK(I1,BA2(I3),I3)	Write Random Access Fileset	TSDISK	I9 CAL TSDISK
CALL XRCL	Request Recall Status		11.2.20

11.2 Descriptive Notes for Functions and Subroutines

The following notes supply only summary information for those library routines which have separate writeups.

11.2.1 BLOK

The BLOK subroutine allows the user to force the use of blocked binary on a fileset for which it is not normally used. This applies only to filesets processed with binary READ and WRITE statements. The statement:

```
CALL BLOK(unit)
```

must be executed before any I/O operations are performed on the fileset. The argument unit is the logical I/O unit number for the fileset or the L-form Hollerith literal fileset name given on the PROGRAM statement.

For more detail on blocked binary see Section 13.1.1. See also NOBLOK in Section 11.2.14.

11.2.2 Boolean Functions: AND, OR, COMPL

The boolean functions (AND, OR, COMPL) operate in a manner similar to the masking operations (see Section 3.4), and are redundant to some extent. The table in Section 3.4 which defines the masking operator is related to the Boolean functions as follows:

```
COMPL(B1)    is equivalent to   .NOT.  B1
AND(B1,B2)   is equivalent to   B1 .AND.  B2
OR(B1,B2)    is equivalent to   B1 .OR.  B2
```

The Boolean functions are used when more than two arguments are to be combined by AND or OR or when it is anticipated that the program may later be converted for a different machine.

11.2.3 Console Simulation Routines: SLITE, SLITET, SSWTCH

On the console of the computer for which Fortran was originally implemented (IBM 704) were four indicator lights which could be tested and turned on and off by programmed instructions. There were also six switches which could be set manually and tested by programs. These were known as sense lights and sense switches respectively. To provide compatibility for programs which used these features, three routines are provided which are used as follows:

Set Light CALL SLITE(I) turns on simulated light I for $1 \leq I \leq 6$, and turns off all the simulated lights if I = 0. The simulated lights are all off at the beginning of execution.

Sense Light Test CALL SLITET(I,J) sets J = 1 if simulated light I is on and turns light I off; sets J = 2 if simulated light I is off and leaves it off.

Sense Switch Test CALL SSWTCH(I,J) sets J = 1 if simulated switch I is on, otherwise SSWTCH sets J = 2. The simulated switch is unchanged. See the SWITCH statement in CALIDOSCOPE Control Statements.

11.2.4 DBLE

The use of the DBLE function does not extend the precision of its argument. It merely supplies a correctly formatted low order part of value zero to go along with the original argument which is the high order part of the result.

11.2.5 DISPLA

The first argument is a name which is printed on the job log followed by the value of the second argument. If the second argument is a normalized real number, it is printed in a format similar to "E" formatted output with fifteen significant figures, otherwise it is printed as a sixteen digit integer (with leading zeroes).

11.2.6 DUMP, PDUMP

DUMP and PDUMP differ only in that PDUMP returns to the calling program after completion of the dump requests, whereas DUMP terminates execution of the job step. Arguments are supplied in groups of three (with up to sixteen groups). Each group of three arguments specifies a block of central memory to be printed out. These arguments have the following order:

(1) First word to be dumped (Not the address of the first word)
(2) Last word to be dumped (Not the address of the last word)
(3) Format code as follows:
    0 or 3: Octal dump
    1: Real numbers to be dumped in floating decimal
    2: Integers to be dumped in decimal

Thus

```
I = 0
J = 1000
CALL PDUMP(I,J,0)
```

is not a request to dump the first 100 words of the user's memory, but to dump the block of memory beginning with the storage allocated to variable I and running through the storage allocated to variable J. In order to dump between calculated addresses, the following trick may be used:

Assume M is an array which is declared in the program, that the value of I is the address of the first word to be dumped and the value of J is the address of the final word to be dumped. Then

```
K = 1 - LOCF(M)
CALL PDUMP(M(K+I),M(K+J),0)
```

will perform the required dump.

If it is desired to express a dump block limit for either DUMP or PDUMP by reference to a statement location rather than a variable, use the statement number followed by the letter "S" as the argument, e.g.,

```
CALL PDUMP(10S,105S,0)
```

11.2.7 EOF

The EOF function permits end of file checking in a manner compatible with the statement forms of ANSI Fortran. The statement:

```
IF(EOF(1)) 10, 20, 10
```

is equivalent to the non-standard statement:

```
IF(EOF,1) 10, 20
```

The form IF(EOF(I)) GO TO 10 may be used by declaring EOF to be of type logical. EOF may have as its argument the logical I/O unit number or the fileset name as an L-type literal (again, non-standard):

```
IF(EOF(5LINPUT)) 10, 20
```

which may avoid the need for files to be declared equivalent on the PROGRAM card.

11.2.8 IEOI

IEOI is a function to test for end of information encountered on a fileset (and to clear the flag in a manner analogous to the IF(EOF(I)) statement). The IEOI function has one argument which is the logical I/O unit number or the L-form Hollerith literal fileset name. IEOI has a positive nonzero value if an end-of-information has been encountered, and zero otherwise.

An end-of-file status will always be set along with the end-of-information status, so EOI need be tested for only if an EOF is found.

11.2.9 INDVCEX

The Input Device Capacity Exceeded flag is turned on when a physical record longer than the stated PRU length (see Chapter 13) is read. Executing the function INDVCEX turns off the flag and returns a positive nonzero result if the flag was on. If the flag was off INDVCEX returns a positive zero. If an attempt is made to continue reading without resetting the flag after this error occurs, execution is terminated with an error code 63.

INDVCEX has a single argument, the logical I/O unit number or L-form Hollerith literal fileset name.

11.2.10 LEGVAR

LEGVAR tests the legitimacy of the value of its argument and returns the following values:

Argument Value	Result
Indefinite	-1
Legitimate	0
Infinite	+1

For details on these forms, see Section A.3.

11.2.11 LENGTH

LENGTH is a function which is called with a single argument which is either a logical I/O unit number or an L-form Hollerith literal fileset name. LENGTH returns the number of words contained in the last record read from the indicated unit. If the read was done with a BUFFER IN statement, the record is a logical record. If the read was done with an unformatted READ statement, the record

is a binary record. LENGTH does not apply to formatted READ operations.

If the list on the last READ exceeded the amount of data contained in the record, LENGTH returns a negative number whose absolute value is the true length of the record. If this condition occurs and LENGTH is not called, execution is terminated with error code 54 when the next operation is attempted on the fileset.

11.2.12 MEMORY

The length of the current central memory field may be determined by the statement

```
CALL MEMORY(2HCM,0,LENGTH)
```

which will return the CM field length in the variable LENGTH.

Similarly, the statement

```
CALL MEMORY(3HECS,0,LENGTH)
```

will return the length of the ECS field in LENGTH.

Other calls to MEMORY to adjust the field lengths and to determine the maximum amounts of memory allowed are available. For details, see library writeup Q4 CAL MEMORY.

11.2.13 Modular Functions: AMOD, DMOD, MOD

The modular functions are defined as follows:

```
f = arg1 - [arg1/arg2]*arg2
```

where [x] is the integer of greatest magnitude whose sign is the same as that of x and whose magnitude does not exceed the magnitude of x, e.g., [3.1] is 3, [-2.5] is -2

11.2.14 NOBLOK

The NOBLOK subroutine allows the user to <u>inhibit</u> blocking for a disk fileset processed by binary WRITE or READ statements. (Blocking is <u>automatically</u> inhibited for all magnetic tape filesets, for disk filesets with a print or punch disposition, and for filesets named INPUT, OUTPUT, PUNCH or PUNCHB). The form of the call is

```
CALL NOBLOK(unit)
```

where unit is a logical I/O unit number or the L-form Hollerith literal fileset name. This call must appear before the first use of the fileset in the program.

For more detail on blocked binary see Section 13.1.1. See also BLOK in Section 11.2.1.

11.2.15 Obsolete Routines: DVCHK, OVERFL, START, TIME

These routines are obsolete. They are retained in the system only to avoid introducing incompatibilities for existing programs. DVCHK and OVERFL are a remnant of the conversion of the programs which had formerly run on the 7090. Their purpose is now accomplished, so far as the CDC hardware allows, by the LEGVAR function. START and TIME just placed messages in the job log which is now done by REMARK.

11.2.16 RANF(R1)

R1=0 The RANF function returns a sequence of pseudo-uniform random real values between zero and one on successive calls with a zero argument.

R1≠0 Selection and repetition of the generated sequence may be controlled by RANF references with nonzero argument as follows:

R1<0 RANF returns the last preceding result of RANF(0) or the predefined basis of the pseudo-random sequence if there have been no previous references by RANF(0).

R1>0 R1 is altered if necessary to set its low order bit to one and its magnitude to between zero and one (by inserting a characteristic of 1717B). The result is used as a new basis for the sequence being generated and is returned as the result of this reference to RANF as well as in R1 (therefore R1 must not be a constant in this case). The result is not necessarily normalized.

11.2.17 REMARK

A message may be transmitted to the job log from a Fortran program by the statement:

```
CALL REMARK(m)
```

where m is a Hollerith literal constant, variable or array containing the string of characters which comprise the message. The maximum length of the string is forty characters.

11.2.18 RETURN

RETURN performs the same action on a fileset as the UNLOAD control card. The call has the form:

```
CALL RETURN (I)
```

where I may be either an integer designating the logical unit number for the fileset, or the L-form Hollerith literal name of the fileset as it appears in the PROGRAM statement. For example:

```
CALL RETURN(3)
```

or

```
I=3
CALL RETURN(I)
```

or

```
CALL RETURN(5LTAPE3)
```

all unload the fileset TAPE3.

11.2.19 SYSTEMP

SYSTEMP may be called to issue diagnostics for conditions detected within a Fortran program. The seventh argument to SYSTEMP is the error number (51 for a fatal error, 52 for a non-fatal error), and the eighth is an error message in the form of a Hollerith literal which begins with a carriage control character. SYSTEMP prints out the diagnostic supplied by the user followed by a traceback and tallies the error for the final summary. The first six arguments of a SYSTEMP call should be the first six arguments to the routine calling SYSTEMP. If this routine has fewer than six arguments, they should appear in the same positions in the SYSTEMP call and dummy arguments added to complete the six.

11.2.20 XRCL

When XRCL is called, a periodic recall request is issued to the operating system. This means that the job relinquishes the central processor until a fixed period of time elapses or until a peripheral processor posts some progress on a task for the job. For example, on an I/O operation this could mean that a physical record has been processed and the buffer pointers updated. The job will proceed immediately if there is no peripheral processor active for the job. Repeated periodic recall requests without any intervening peripheral processor activity may cause an error termination of the job step.

11.2.21 SETRDCT

Subroutine SETRDCT is used to set the error recovery procedure which will be used for each physical record of a magnetic tape fileset in case of errors while reading. The default is currently nine attempts (eight retries). The first argument is the logical I/O unit number or the L-form literal fileset name. The second argument is a code specifying the procedure to be used. Zero indicates the system default is to be used. If the code is one, the system will not attempt to reread the record and the wrong mode read detection is disabled. A count of one is useful in reading tapes generated with off line devices which do not generate correct parity indications. The IF(IOCHECK,i)n1,n2 statement should then be used to check each read operation (see Section 10.3).

11.2.22 ABORT

The statement

```
CALL ABORT (message)
```

where message is a Hollerith literal constant, variable, or array containing a coded message of up to sixty characters, will immediately abort the current job step and enter the given message in the job log. The effect of aborting a job step is described in CALIDOSCOPE Control Statements in the section "Control Statement Processing Sequence".

11.2.23 MCLOCK

The statement

```
CALL MCLOCK(ITIME)
```

sets the variable ITIME to the current time of day as a display coded string in the following format:

```
10Hbhh:mm:ssb
```

where b is a blank, hh is a number from 0 to 23 designating hours, mm and ss are numbers from 0 to 59 designating minutes and seconds respectively. The twenty-four hour clock system is used.

11.2.24 MDATE

The statement

```
CALL MDATE(IDATE)
```

sets the variable IDATE to the current date as a display coded string in the following format:

```
10Hbddbmmmbyy
```

where b is a blank, dd is a number from 1 to 31, mmm is a three letter month designation and yy is the last two digits of the current year.

11.2.25 MZERO

The function MZERO has the logical value .TRUE. if its argument is a minus zero and has the logical value .FALSE. if its argument has any other value.

11.2.26 DABS, IDINT, SNGL

In violation of the ANSI Fortran standard, the DABS, IDINT, and SNGL functions are external rather than intrinsic functions. This raises the possibility of some name conflicts which might not otherwise occur. Also see next section with respect to IDINT.

11.2.27 IFIX, INT, AINT, IDINT

The definition of the IFIX function in the ANSI Fortran standard is ambiguous. As implemented, it is the same as the function INT. These function values may be described as:

> Sign of argument times largest integer less than or equal to absolute value of argument.

SUBPROGRAM STATEMENTS

Subprogram Declarations	Executable	Section
PRØGRAM name (f_1,...,f_n)	N	7.3
SUBRØUTINE name (p_1,...,p_n)	N	7.4
FUNCTIØN name (p_1,...,p_n)	N	7.6
type FUNCTIØN name (p_1,...,p_n)	N	7.6
BLØCK DATA name	N	7.14
Subprogram Linkage Statements		
ENTRY name	N	7.12
EXTERNAL $name_1$, $name_2$, ...	N	7.13
CALL name	Y	7.5
CALL name (p_1,...,p_n)	Y	7.5
RETURN	Y	6.7

DATA DECLARATION AND STORAGE ALLOCATION

Type Declaration		
CØMPLEX list	N	5.1
DØUBLE PRECISIØN list	N	5.1
DØUBLE list	N	5.1
REAL list	N	5.1
INTEGER list	N	5.1
LOGICAL list	N	5.1

N = No Y = Yes

Storage Allocation	Executable	Section
DIMENSIØN $V_1,V_2,\ldots,V_n$	N	5.2
CØMMØN/i_1/$list_1$/i_2/$list_2$ ...	N	5.3
EQUIVALENCE $(A_1,B_1,\ldots).(A_2,B_2,\ldots)\ldots$	N	5.4
Storage Initialization		
DATA I_1/LIST/,I_2/LIST/,...	N	5.5
STATEMENT FUNCTION		
name$(p_1,p_2,\ldots,p_n)$=Expression	N	7.8
REPLACEMENT STATEMENTS		
A=Arithmetic Expression	Y	4.1
L=Logical/Relational Expression	Y	4.3
M=Masking Expression	Y	4.4
EXECUTION SEQUENCE CONTROL		
Intraprogram Transfers		
GØ TØ n (n is a statement label)	Y	6.1.1
GØ TØ m (m is an integer variable)	Y	6.1.2
GØ TØ m, $(n_1,\ldots,n_m)$	Y	6.1.2
GØ TØ $(n_1,\ldots,n_m)$,i	Y	6.1.4
Conditional Statements		
IF (A)n_1,n_2,n_3	Y	6.2.1
IF (L)n_1,n_2	Y	6.2.3
IF (L)s	Y	6.2.2
IF (ENDFILE i)n_1,n_2	Y	10.4
IF (EØF,i)n_1,n_2	Y	10.4
IF (UNIT,i)n_1,n_2,n_3,n_4	Y	10.4
IF (IØCHECK,i)n_1,n_2	Y	10.4

Loop Control		Executable	Section
	DØ n i=m_1,m_2,m_3	Y	6.3
Miscellaneous			
	ASSIGN s to m	Y	6.1.3
n	CONTINUE	Y	6.4
	PAUSE	Y	6.5
	PAUSE n	Y	6.5
	STØP	Y	6.6
	STØP n	Y	6.6
I/O FORMAT			
	FØRMAT ($spec_1$,$spec_2$,...)	N	9.2
	NAMELIST/y_1/a_1/y_2/a_2/ ...	N	9.9
I/O OPERATION STATEMENTS			
	READ n,L	Y	10.2
	PRINT n,L	Y	10.1
	PUNCH n,L	Y	10.1
	READ (i,x)	Y	10.2
	READ (i,n)L	Y	10.2
	WRITE (i,x)	Y	10.1
	WRITE (i,n)L	Y	10.1
	READ(i)L	Y	10.2
	WRITE (i)L	Y	10.1
	ENCØDE (c,n,v)L	Y	10.6
	DECØDE (c,n,v)L	Y	10.6
	BUFFER IN (i,p)(fw, lw)	Y	10.5
	BUFFER ØUT (i,p) (fw, lw)	Y	10.5

I/O File Handling	Executable	Section
END FILE i	Y	10.3
REWIND i	Y	10.3
BACKSPACE i	Y	10.3
PROGRAM AND SUBPROGRAM TERMINATION		
END	N	6.8

13.0 Filesets

All information handled by the Operating System is stored in named collections called filesets (sets of files) which are structured in a manner dependent upon both logical and physical considerations. That is, the representation of the data within the computing system should be structured to reflect the natural relationships among the data and to make it convenient and economical to use for its intended purpose. In other words the data should be represented in a logical form. However, since a computer consists of physical components with varying physical characteristics, information must be recorded on each medium in a form that can be "understood" by the component being used to process it.

The criteria for structuring information to be transferred between humans and a computer are exactly analoguous to those for transferring it between humans. A person wishing to convey his ideas clearly to another person usually writes them on paper, indicating the basic level of a thought with a statement ended by a period, designating the next higher level, a group of related statements, by a paragraph and groups of related paragraphs by a chapter, etc. Further guides are often provided in the form of heading and numbering. This representation of logical structure, however, since it is recorded on the physical medium of paper, is restricted in format by the number of letters that fit on a line and the number of lines on a page. It is this sort of constraint which must be dealt with in organizing data for particular media.

13.0.1 Fileset Access

The operating system recognizes two arrangements of filesets: sequential- and random-access. A sequential-access fileset is one in which information is read or written in sequential order, as on a magnetic tape. It has a beginning and an end, and is positioned at a given point at any time, ready to read or write the next record. When a sequential-access fileset is written, any information following the record written is erased, Since the last write determines the end of the useful information on the fileset, it is not permitted to read after writing on the fileset without having moved backward over the section just written. All of the Fortran language I/O statements deal with sequential-access filesets.

A random-access fileset is one in which any record may be read or rewritten at any time, without processing intervening data to accomplish the repositioning. New records may also be added at any time. This type of accessibility results

from the fact that an index* of where records are found is associated with the fileset. See I9 CAL TSDISK as an example of a library program for processing random-access filesets.

A sequential fileset may reside on magnetic tape or in disk storage or on punched cards (from which it will be copied to the disk before being used). Random access filesets may currently reside only on disk (a rotating storage device, see Section 13.2.4) or in ECS. The great majority of users will be concerned only with sequential filesets.

13.0.2 Fileset Names

The names which may be used for filesets are the same as those which may be used for variables, arrays, etc., i.e., alphanumeric identifiers. Certain fileset names are reserved so that they always refer to particular filesets. For example, the fileset containing the job input following the control record is given the name INPUT; the name OUTPUT refers to the information normally printed by a line printer; PUNCH refers to the images of the cards normally punched in coded mode by the card punch; and PUNCHB to the images of cards normally punched in binary mode. Other filesets may be assigned special disposition by the user via REQUEST or COMMON (see CALIDOSCOPE Control Statements).

Within a Fortran program, input/output statements are automatically associated with particular filesets from which or onto which information is to be transferred. Since most I/O is performed via the card reader and line printer, the Fortran READ n statement has the fileset named INPUT associated with it; the PRINT statement, the fileset named OUTPUT. For convenience, the user may establish his own associations between filesets which he creates and wishes to use and the I/O statements of his program. This is accomplished by declaring file names on the PROGRAM statement (the first statement in a Fortran main program) which correspond to the filesets referred to by the particular I/O statements used. (See Chapter 10.) Although the user may attach other names to filesets, the I/O statements are constructed to render most convenient use of names of the form TAPEi, where $1 \leq i \leq 99$. If i=5, a statement of the form READ(5) refers to the fileset of the name TAPE5 which must appear in the PROGRAM statement. The value i is the logical I/O unit number. (See Section 7.4 for greater detail.) Note that names of the form TAPEi are a carry-over from earlier days when tape was the most commonly used I/O medium. Filesets so named need not be on magnetic tape and in most circumstances will probably reside on the disk (see Section 13.2.4). In general,

*The index is normally the last record of the fileset and is distinguished by a record level number of 15B.

the PROGRAM statement defines which filesets are to be referenced by the I/O statements within the program. If a fileset named does not already exist for the job, the system creates an empty fileset on the disk and gives it the requested fileset name. Fileset names may be equivalenced to one another in the PROGRAM statement so that two or more logical I/O unit numbers refer to the same fileset. In addition, the EXECUTE or LGO statements allow other fileset names to be substituted for the file names stated in the PROGRAM statement.

The following examples illustrate the topics just discussed.

```
1.    PROGRAM  RECORD(INPUT,OUTPUT,TAPE6)
      DIMENSION NAME(4), MONACT(100)
           :
      READ 10, NAME, PAY
 10   FORMAT(4A10,F10.2)
           :
      PRINT 11, NAME, TPAY
 11   FORMAT(5X,4A10,F10.2)
           :
      WRITE(6)(MONACT(I),I=1,50)
           :
      END
```

The program uses the filesets INPUT, OUTPUT, which are normally created automatically by the operating system, and TAPE6, which will be created by the system as an empty fileset on the disk if it has not been created by a jobstep preceding execution of this program.

```
2.    PROGRAM MAIN(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      DIMENSION A(100),B(50),C(50)
      READ 11,A
      PRINT 12,A
      READ (5,11)B
      WRITE (6,12)B
           :
      END
```

TAPE5 is equivalenced to INPUT and TAPE6 is equivalenced to OUTPUT. Thus, both READ statements read from the INPUT fileset and the WRITE and PRINT statements both put information to be printed on the OUTPUT fileset.

3. J9998,1,100,40000. J. DOE PAYMASTER
RUN.
REQUEST,UCSED,I.2137
LGO,,,UCSED.
7-8-9 Card

```
PROGRAM FILEH(OUTPUT,TAPE6,PUNCHB,TAPE7=PUNCHB)
      :
      :
READ(6) ERECORD
      :
      :
WRITE(7) TOTALS
      :
      :
END
```

Because the LGO card has substituted the fileset UCSED for the fileset TAPE6, the READ(6) statement reads the fileset UCSED which has been assigned to magnetic tape. Of course, TAPE6 does not have to be replaced by UCSED, but it was done to demonstrate the generality of fileset naming and substitution. A more usual form is to have "REQUEST,TAPE6,I.2137" instead of "REQUEST,UCSED,I.2137" and to have "LGO" instead of "LGO,,,UCSED."

Note that the order of the parameters (the fileset names) on the LGO card is important here: the first name is associated with the name of the fileset on which the program resides, the second name on the card is associated with the first fileset name given in the PROGRAM statement, etc. (Thus, the fileset name UCSED is associated with the second name, TAPE6.) Only 3 parameters are allowed in this example, since TAPE7 is already equivalenced in the PROGRAM statement to PUNCHB and therefore is not counted as a fileset name parameter. Only the fileset names explicitly stated on the LGO card override the program fileset names listed in the PROGRAM statement. Thus, OUTPUT and PUNCHB are unchanged in this example.

The WRITE(7) statement causes binary information to be placed on the PUNCHB fileset (a fileset predeclared by the system for the punching of binary cards).

13.0.3 Fileset Disposition

Except for those filesets automatically associated with special dispositions by the operating system* all the filesets

---

* The filesets which automatically receive dispositions are:
OUTPUT - Print (PR) disposition
PUNCH - Punch Hollerith (PU) disposition
PUNCHB - Punch Binary (PB) disposition
INPUT - Protected from writing in case it is necessary to restart the job. INPUT is released after execution is completed. The special character of INPUT is not, strictly speaking, a disposition.

associated with the execution of a job are normally temporary and are assigned to the disk. That is, in the absence of a specific request to the contrary, storage space for a fileset is allocated by the operating system as the fileset is created and this space is released and is available for reuse as soon as the job completes execution.

By use of the REQUEST Control Statement* a fileset may be assigned to magnetic tape, or post-execution processing such as printing or punching may be designated. By use of the COMMON Control Statement* (not related to the Fortran COMMON statement) a fileset may be retained on the disk for a limited period after the job creating it terminates. The COMMON Control Statement also allows the attachment of a previously created common fileset to the current job. Public filesets may also be attached to the job with this statement.

Only one disposition may be made for any fileset, i.e., it may not be both printed and punched, nor may it be made common and printed, etc.

13.1 The Logical Structure of Filesets

In the logical arrangement of information in a fileset to be handled by the Operating System, the following structural levels are defined: logical records of fourteen** levels, and files.

13.1.1. Logical Record

A logical record is a string of physical records (see Section 13.2) which is delimited by an end-of-record (EOR) whose exact form depends on the storage medium used. In a Fortran binary fileset on tape, each binary record normally constitutes a logical record. This is known as the unblocked binary form and is less compact than the form called blocked binary which is normally written on most disk filesets (all those without dispositions specified). It is possible to override the assumed blocking of a fileset by calling library routines BLOK or NOBLOK (see Chapter 11) if desired.

In the blocked form an additional word appears at the beginning and end of each binary record to declare its length and position within the fileset. These words are processed entirely automatically and need not be allowed for by the user except that he must not attempt to read a blocked fileset as unblocked, nor vice versa. Their presence allows the fileset to be written without including an EOR for each binary record. Thus, blocking is most important when the binary records are short.

Binary filesets may be converted between the blocked and unblocked forms with the BBLOK and BUBLOK utility programs.*

---

* See CALIDOSCOPE Control Statements.

** Two addftional levels exist but are reserved for special use, e.g., a zero length record of level 15 (17B) is an end-of-file (EOF).

Coded filesets are not normally divided into logical records except for the fileset INPUT where an EOR is treated as an end-of-file. If one wishes to test for EOR on a coded fileset, library routine IEOR may be used (see Chapter 11).

Logical record level numbers: The Operating System allows related logical records to be grouped into an organized hierarchy by means of level numbers, much like the ideas treated in a book are organized. (See Figure 13.1) A single logical record forms the unit of information and therefore has the lowest level number. A higher level number delimits a set of logical records consisting of the logical record at that level plus all preceding records (of a lower level) back to one of an equal or higher level. Up to 14 different levels may be defined (00 to 15B). In many cases, although not from standard Fortran, the user may take advantage of this structure imposed by the System in order to organize his own records. For example, the CALIDOSCOPE fileset positioning control statments SKPF and SKPB can position filesets using level numbers (see CALIDOSCOPE Control Statements).

Logical Records	Level Number	Paragraph	Section	Chapter
1	0	Employee A	Department	College
2	1	Employee B	W	Q
3	0	Employee C	Department	
4	0	Employee D	X	
5	2	Employee E		
6	0	Employee F	Department	College
7	0	Employee G	Y	R
8	2	Employee H		
9	0	Employee I	Department	College
10	2	Employee H	Z	S
⋮	⋮	⋮	⋮	⋮

Figure 13.1

A schematic description of the use of record level numbers to organize a fileset containing University employee records.

13.1.2 File

All logical records pertaining to a general subject may be formed into a major collection of information called a file. A file need not be divided into logical records. A file being written may be terminated by executing an

```
ENDFILE i
```

Fortran statement. When reading, an

```
IF(ENDFILE i)
```

or

```
IF(EOF(i))
```

or

```
IF(EOF,i)
```

may be used to test whether an end-of-file (EOR on fileset INPUT) was encountered during the previous access.

13.2 The Physical Structure of Filesets

The logical structure of information is so defined that filesets may be stored on assorted media without the logical structure being lost due to a medium's physical characteristics. Thus a fileset can be recorded on disk, magnetic tape or cards. A set of cards read by the card reader becomes a fileset on disk. A set of lines to be printed on a line printer is also stored as a fileset on disk until it has been printed. When a fileset is transferred from one medium to another (e.g., cards to disk, disk to tape or vice versa) each of which possesses different physical properties, it is the logical structure of the fileset which prevents the information contained in the fileset from losing its structure and thereby its meaning as a result of being transcribed to a different medium.

This section defines some terms and concepts relevant to the physical structure of filesets. Subsections explain the particular methods used to represent logical structure on each available storage medium.

Physical Record Unit: On a physical device, the actual units in which the information is recorded are called physical records; they define the amount of information which is moved as a unit to or from that medium in one transfer (read or write) operation. This is a purely physical element, hence the term physical record. The maximum size of physical records normally used on a particular device is called the physical record unit (PRU) for that device. On some devices the PRU is completely determined by the hardware (e.g., card readers and line printers; the PRU on other devices is fixed by the operating system (e.g., disk and ECS); finally, on some devices a PRU size is assumed by the operating system but may be overridden by the user (e.g., external magnetic tape, see I9 CAL SETPRU).

Blocking: Efficient use of devices such as magnetic tape and disk demands that logical information be grouped or blocked into larger PRUs than are used for cards. On magnetic tape, a 3/4" gap (at least) of blank tape, called the inter-record gap (or just record gap), is required by the tape drive between physical records to allow time for the physical device to stop and start up again. If the PRU size were equivalent to that of cards, only 14% of the recording surface of the tape would contain information, since 80 columns of information can be written on 1/7" of tape at 556 characters per inch.

Blocking is a general term used to described the action of combining two or more records into a standard element, called a block, and for dividing long records (normally binary) into several such blocks for the purpose of improving I/O efficiency. In general, blocking may be fixed or variable, the important factor being that there be an established scheme for deblocking (restoring to its original structure) the information once it has been blocked. In fixed blocking, the length of each record is predetermined and constant, with a constant number of elements in each block. In variable blocking, the length of the records varies and is signalled either by a unique terminator, or by a pointer word at the beginning (and/or end) of the record telling how long the record is. A block as used here may be thought of as being synonymous with a physical record; it may contain a partial record, exactly one record, or more than one record.

Summary of Input-Output Devices and Storage Media

The following input and output devices are presently connected to the CDC 6400 computers and are used in transferring information between storage media.

1. Card Reader

   The CDC 405 card reader is used to read Job decks submitted at the Input counter into the system. It is capable of reading 80-column punch cards at a maximum rate of 1200 cards per minute.

2. Card Punch

   The CDC 415 card punch is used to punch output from user's jobs when requested. It can punch 250 80-column cards per minute.

3. Line Printers

   There are both CDC 501 and IBM 1403 line printers in the system. The printers produce printed output from the system at a maximum rate of 1000 lines per minute

with 132 characters per line. See Table H of Appendix A for the character set used.

4. Magnetic Tape Units

There are five CDC 604 magnetic tape units in the system. They use 1/2" seven track magnetic tapes which serve as a permanent, portable storage medium in cases where cards would be too bulky and/or inconvenient. All reels of magnetic tape have two aluminum reflective spots, one at the beginning and one at the end, to signal the beginning and end, respectively, of the physical reel. They can be recorded and read at densities of 200, 556, or 800 7-bit frames (6-bits plus parity check bit) per inch and at a speed of 75 inches per second. Thus at 800 characters per inch, the transfer rate is 60,000 characters per second.

5. Disk

There is one CDC 6638 Disk Storage Device in the system. It is a random-access large-capacity memory commonly used for job input waiting to be processed, output filesets waiting to be printed, punched or sent to remote terminals, user scratch filesets, public filesets, user common filesets, system library programs, etc. Once the disk is positioned, information may be read or written at the effective rate of 420,000 characters per second. The capacity of the entire disk memory is 131,072,000 characters including space occupied by the Operating System itself; a half-track, which has a maximum capacity of 32,000 characters, is the smallest amount of disk which can be assigned to a fileset.

6. Remote Terminals

There are numerous remote terminals, such as Teletypes, which are located on campus and are owned or rented by individual departments or their members (see Guide to Computer Center Services).

7. Extended Core Storage

There are 500,000 words of auxiliary core storage in the system which may be accessed as an I/O device. Part of this storage is now allocated for system residence, part for optimization of fileset handling, and part for direct access (e.g., see library routine K2 CAL ECSIO).

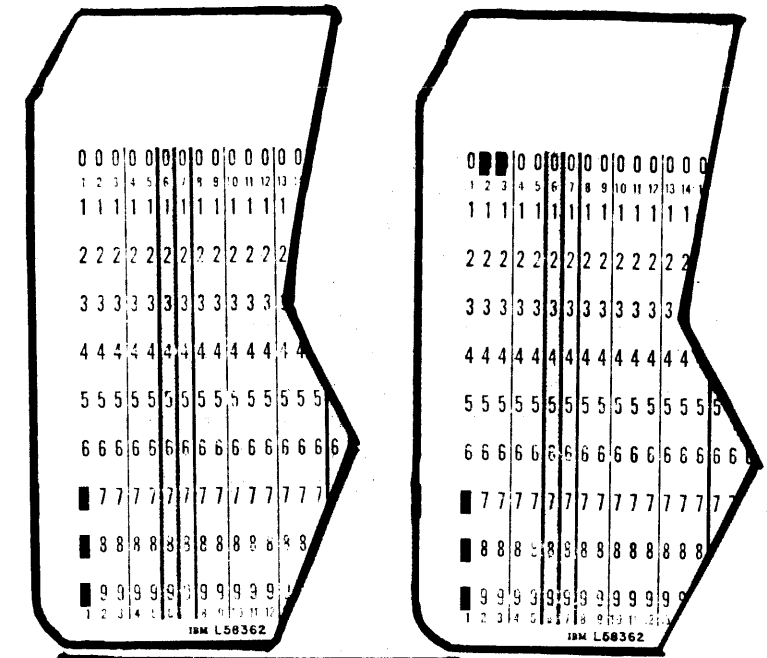
In addition to the devices connected directly to the computer, there are separate devices for graphical output. See the Graphical Display System Manual.

13.2.1 Punched Cards

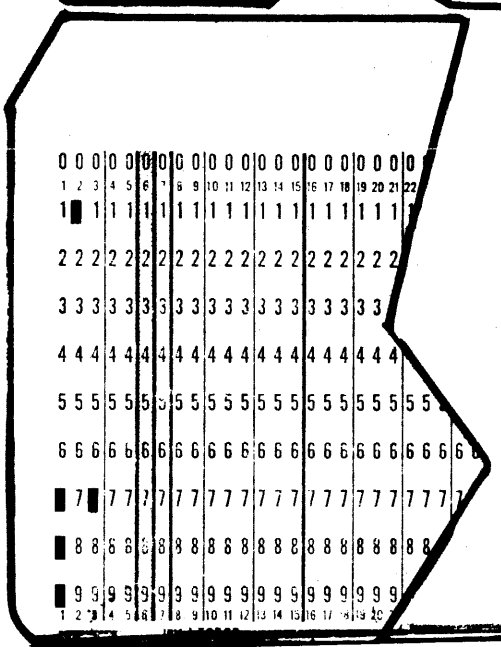
For files which are to be read by a card reader or have been produced by a card punch and therefore are recorded on punched cards, the 80 columns of a card comprise one physical record. Since cards are of fixed size, every physical record is a full PRU. The end of a logical record is indicated by a card with

Figure 2

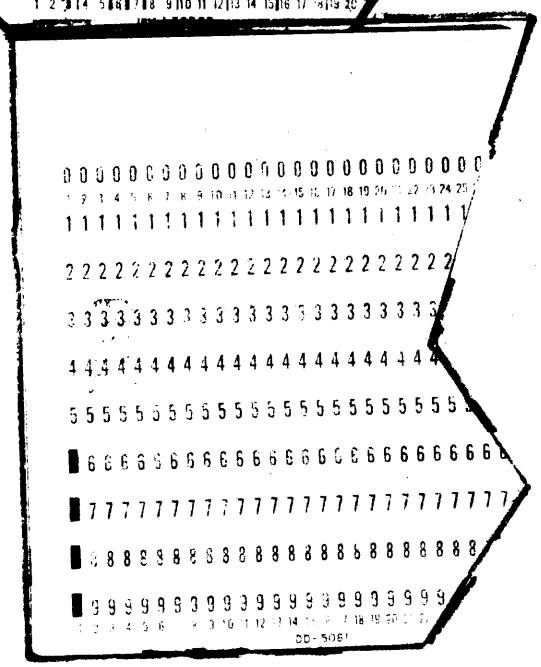
Fileset Structure Delimiters for Filesets on Punched Cards



7-8-9 or EOR card. These two forms are equivalent; the card stock and printing are immaterial.



7-8-9 level 17 or EOF card; card stock and printing are immaterial.



6-7-8-9 or EOI card. Pink cards with 4 square corners are used to end job decks; EOI cards punched by the computer are on normal punch stock.

the multiple punch 7-8-9 in column one; the level number associated with that record consists of two octal digits punched in columns 2 and 3. Since the level number is normally 00, these columns will usually be blank. An end-of-record level 17B occurs only as an end-of-file. The end of a job is indicated by a card with the multiple punch 6-7-8-9 in column one. See figure 13.2. Also see 14.2.6 for examples of other cards.

13.2.2 Printed Output

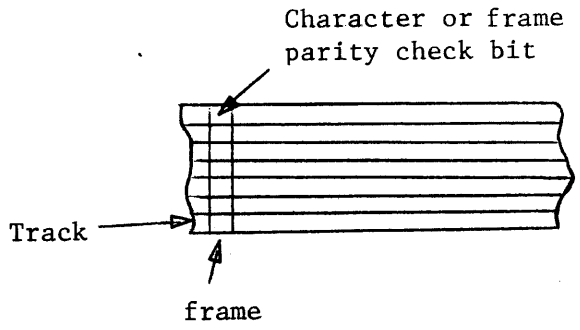
Printed output is produced from filesets produced during job execution and given a print disposition. Since printed output is the only form produced directly by the computer for human "input", its characteristics are quite different from media designed for reentry into the computer.

Each unit record is represented by a printed line of up to 132 characters but the printer has an additional feature which uses an additional character. The first character of each unit record is used to control the vertical movement of the paper past the printing position. A detailed list of the available carriage control characters appears under the PRINT statement (Section 10.1). Logical record structure is ignored in producing printed output.

13.2.3 Magnetic Tapes

The CDC 604 tape drives handle 7-track tapes only. 9-track tapes made on machines using that format must be copied to 7-track tapes before use on the 6400.
A fileset may consist of one or more tape reels.

Parity Checking: Parity checking is a scheme used to check for erroneous bits of information (caused on tape, for example, by dust on the recording surface or skew while reading or writing) on storage media. In the case of 7-track tape a character or frame parity check examines the six information carrying bits in a column across the width of the tape called a frame and inserts a parity bit as the seventh bit in the frame. A block, track, or longitudinal parity check examines the bits along a track of a tape and inserts a parity bit in the corresponding bit position of a terminating frame consisting solely of block parity check bits. If information is written in even parity, the instances of one bits in a frame or track of a block are counted, and a binary 1 or 0 is inserted as the parity bit such that the total of ones is even. If information is written in odd parity, the same count is performed, but the parity bit is inserted such that the total of ones in each frame is odd. The track parity checking always uses even parity.



Section of Magnetic Tape

The parity checking scheme provides a fairly reliable means for checking against erroneous information, but the even parity check introduces a complication; a character consisting of all zeroes has an even number of ones, and therefore, the parity bit is a zero. However, an all zero character would exist on tape as a gap; a series of zeroes would appear as blank tape, and the tape drive would be unable to differentiate between a zero and a stretch of tape containing no information. To avoid this confusion, another bit configuration is written to represent the character code for 0, leaving available for use only 63 of the possible 64 characters which can be represented by six bits ($2^6 = 64$). Odd parity does not produce this problem. Currently on tape, binary information is written in odd parity, and coded information in even parity.

External and internal tapes:

Magnetic tapes are of two types: Those written by and intended for use on a CDC 6000 series machine, (called internal tapes) and those written in a generalized format such that they may be used as well by other types of machines (called external tapes).

INTERNAL TAPE FORMATS (I or U tapes*)

1) Binary - A file written on magnetic tape in binary mode exists as a series of physical records not exceeding the standard length (PRU) of 512 CM words. The end of a logical record is signalled by a 48 bit

* See REQUEST in CALIDOSCOPE Control Statements.

marker in which the four low order bits indicate the level number in binary. If the information in the logical record does not fit exactly into an integral number of PRUs, the 48 bit marker is appended to the last physical record written, forming a short physical record. Otherwise the marker is written as a separate physical record which is said to have zero length. In either case, the marker is called an end-of-record (EOR). The end of a file is indicated by an end-of-record to terminate the last record, followed by a second end-of-record of level 17B. This constitutes a "zero length logical record of level 17B" and is called an end-of-file. Binary tapes are always recorded in odd parity.

2. Coded - The structure of coded files written for use on CDC 6000 series machines is basically the same as that of binary with three exceptions. The first exception is that the tape is recorded in even parity. Second, the standard PRU is 128 CM words rather than 512, since some of the space in the peripheral processor which would be allotted to the buffer is taken up to perform the required conversion of the coded information. The third exception stems from the fact that all coded files are composed of one or more unit records (lines), where a unit record is a variable length sequence of characters ($\leq$ 137 characters) terminated by the external BCD terminator 1632_8* converted internally to a zero byte** (0000_8) in Central Memory. The unit records are simply concatenated without regard to the boundaries between physical records. This is an example of variable blocking, mentioned above, i.e., the length of each unit record is completely determined by the position of the terminator. Thus, the division into logical records is not needed to provide for orderly handling of the information.

In both modes on an internal tape, the fact that the remainder of the tape contains no valid information and is to be ignored is signaled by an End-of-Information (EOI). This consists of a

---

* The use of this code as the unit record demarcation has the unfortunate side effect of restricting the use of the character code for colon ":" from data which is to be read or written on coded internal tapes.

** A zero byte used to terminate a unit record in the I/O buffer contains a minimum of twelve zero bits but may contain more since it is always extended, if necessary, to include the low order twelve bits of the CM word which contains it.

octal 17 written in even parity followed by a block parity check character and is generally referred to as a tape mark, since it may be used in other contexts to signify something other than EOI. The tape mark is followed by a special series of characters called an EOI trailer label, to indicate that there is no more information on the fileset.

EXTERNAL TAPE FORMATS (X tapes*)

All magnetic tape filesets which are not in the formats described above for internal tapes are external tapes. The PRU length may be specified by the user** if the defaults indicated below do not fit the user's needs.

Binary - The standard PRU is 512 words (5120 frames). The tape format differs from that of a binary internal tape in only three respects:

(1) There are no level numbers associated with the logical records on external tapes, level zero being assumed. Therefore, each logical record consists merely of a sequence of one or more physical records ending with a short (less than a PRU) record. All but the last physical record in each logical record are one PRU in length. The 48 bit marker is used only when a zero length physical record is required as in binary internal format.

(2) End-of-file is indicated by a tape mark (see above) which is also considered to be a logical record of length zero and level 17B.

(3) There is no EOI on an external tape.

Coded - Each file is considered to contain a single logical record. Points (2) and (3) under binary external tape above also apply to coded external tape.

When writing coded external tape each unit record is written as a separate physical record. Thus although the PRU length is 128 words (1280 frames) the records are normally limited by the Fortran formatted I/O routines to 137 characters.

---

* See REQUEST in CALIDOSCOPE Control Statements.

** See I9 CAL SETPRU.

This limitation can be bypassed (see I4 CAL LNGBCD) so long as the unit records remain shorter than the PRU length*. Due to a peculiarity of CDC hardware all physical records written must contain an even number of characters. If a unit record contains an odd number of characters, a blank will be added at the end. The Fortran formatted output routines normally discard trailing blanks from each unit record (which saves space on disk and coded internal tapes) but this may be controlled by the user (see J4 CAL TRAILB).

When reading coded external tape each physical record is normally read as a unit record so that a zero byte (see note under coded internal tape) is appended to the data read in. The zero byte is not added if the physical record is exactly one PRU in length; thus, the longest record which can be read preserving the record structure is two frames less than a PRU (maximum unit record length of 1278 characters with the default PRU).

For reading of tapes prepared on computer systems which use fixed blocking special controls are available through the library routine I4 CAL SETFXB.

Note: When writing coded records of the usual size (no more than 137 characters) the peripheral processor time required with external tape may be five or ten times greater than with internal tape.

13.2.4 Disk Storage

The CDC 6638 Disk File is composed of 72 magnetic disks which are arranged in two stacks mounted on revolving shafts; the 36 disks in each stack are in turn arranged in two groups of 18 disks each. The two top groups are referred to as UNIT 0; the two bottom as UNIT 1. See Figure 13.3. Data are recorded onto the disk surface by read/write heads fitted onto an access arm. The arm can be moved by a positioner to any track on the surfaces it accesses, and once on the proper track, the revolution of the disk brings a desired location to the read/write head. The surfaces of a disk are also divided radially into 100 sectors of 64 words which are the physical records.

The positioning parameters serve as coordinates for locating each record of information stored on a Disk File. Whereas information stored on magnetic tape is only available sequentially, i.e., after all the information recorded preceding it has been read, information stored on a disk file may (but need not) be referenced as a random-access fileset (e.g., via I9 CAL TSDISK).

---

* For unit records longer than 1278 characters see I9 CAL SETPRU.

The disk file has a fixed PRU of 64 CM words and is always recorded in odd parity. A logical record consists of data written continuously in one or more sectors. A logical record always begins at the beginning of a sector. The time required to position the disk (about 125 milliseconds) is large compared to the time required to read each physical record (about 1 millisecond). This renders the transfer of large records the most efficient way to use the disk.

Filesets residing on disk have the same structure as internal binary tapes, except for the fact that an end-of-information (EOI) mark is not actually written onto disk, but note is kept of its location in a central memory table maintained by the system outside the user's memory field.

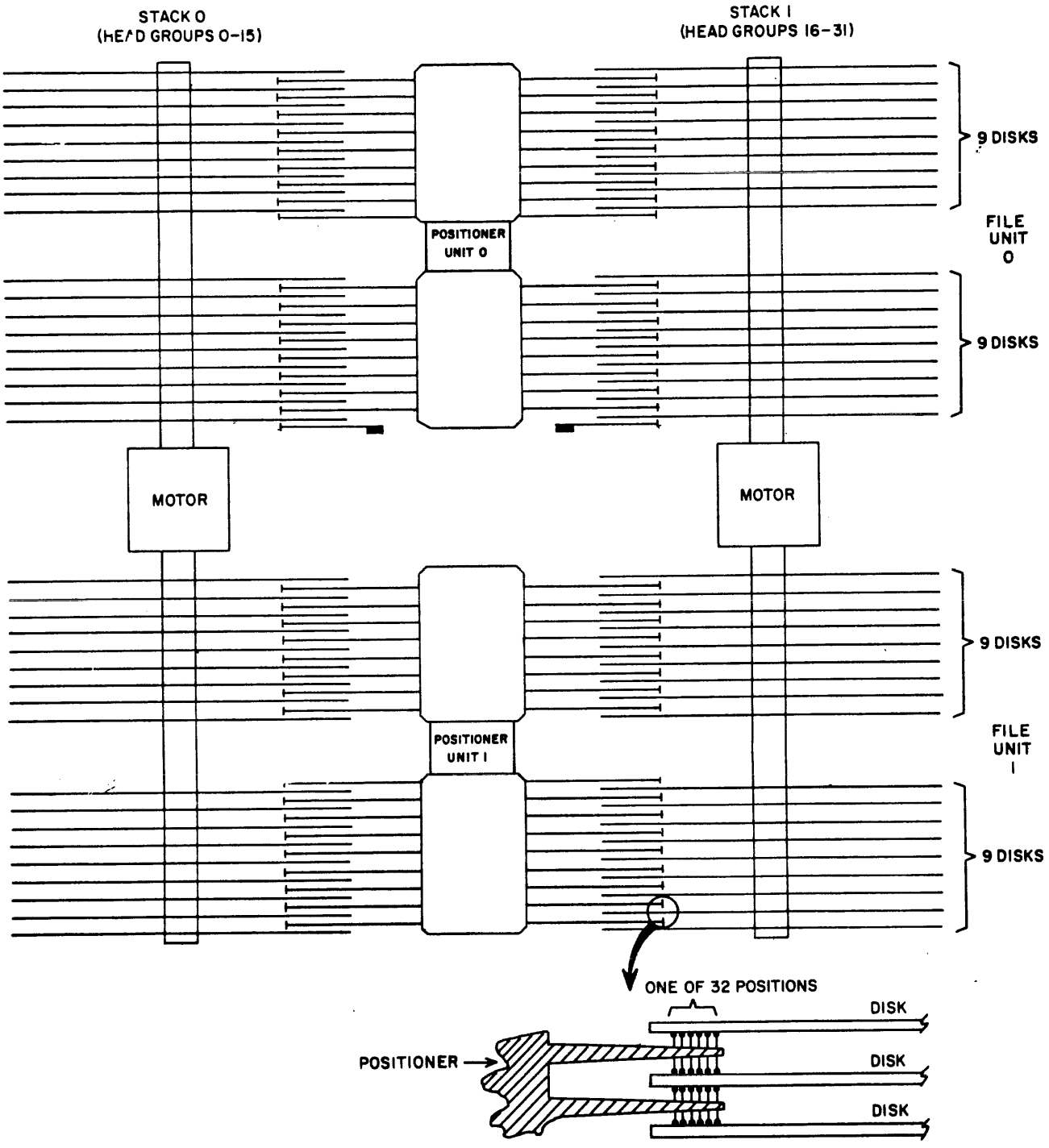


Figure 13.3. 6638 Disk File Disks and Positioners

HOW TO USE THE RUN FORTRAN COMPILER Chapter 14

This chapter shows a typical setup for running a program coded in Fortran and explains the output from a sample program.

After a program has been coded in Fortran, it (along with any data) is either punched on cards for entry into the computer or typed into the computer via a keyboard terminal. In order for the program to be interpreted by the computer, it must be organized in a particular sequence and accompanied by certain control instructions which indicate to the operating system, which controls the processing of programs, how the information is to be handled. The operating system must be told, for example, whether the program is written in Fortran, in some other language, or a combination of languages; it must be told where the beginning and end of the program deck are; it must be told where the data cards begin and end, and where any additional sources of input and destinations for output are (e.g., magnetic tape).

The unit of work which is submitted to the operating system by the programmer is called a job. It begins with a group of lines called control statements, the first of which is a special statement called the Job Identification Statement. The job is terminated by a special code which denotes the end of the information that is being submitted.

The operating system is itself a collection of programs which, given the information from the control statements, controls the sequence of events occurring within a job and communicates necessary information to the computer operator (for example, which magnetic tapes, if any, are required). It must also arrange the scheduling of jobs with respect to each other since it can process up to 6 jobs simultaneously in addition to those being printed, punched, held in queues, etc.

The remainder of the chapter refers to a program punched on cards. For information about running a program from a keyboard terminal, see Appendix A, Table I.

14.1 Flow of Control

A typical job deck is shown in Figure 14.1. The job illustrated here calls for a compilation (i.e., conversion to a form executable by the computer) of a Fortran program (labelled "source deck" in the figure) and execution of the resultant program (i.e., performing the set of calculations specified by the program) with the data cards shown.

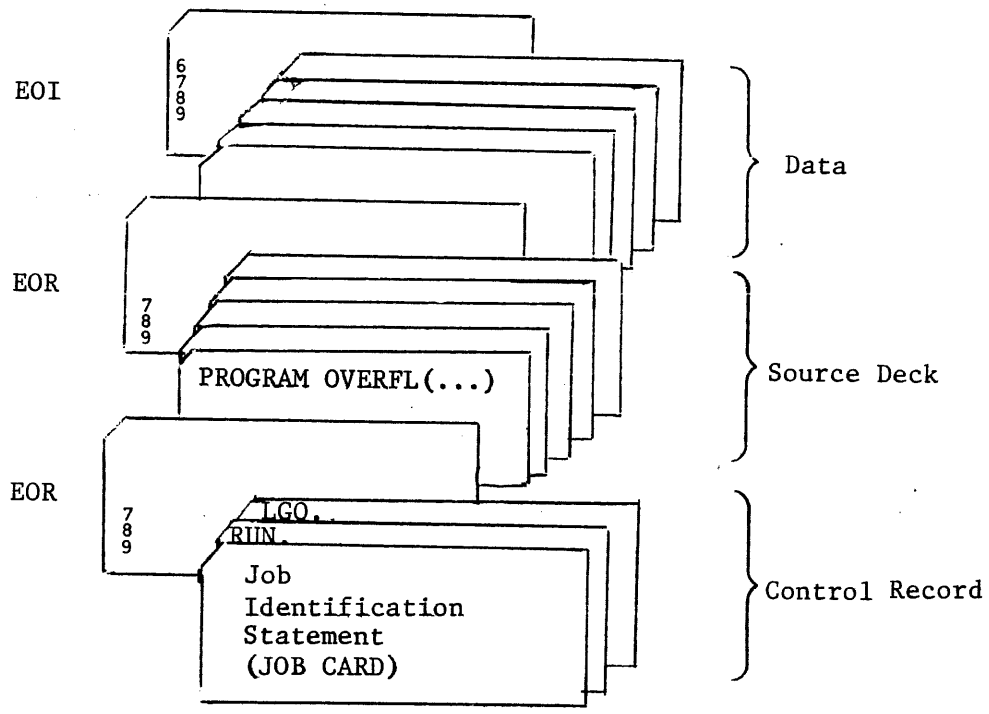


Figure 14.1 Typical Job Deck

The set of control statements heads the deck and is separated from the remainder of the deck by an end-of-record (EOR) card (the card with rows 7, 8, and 9 of column 1 punched out), which signals the end of a logical grouping of cards. The particular control statements and the corresponding arrangement of the remainder of the deck (everything after the first EOR) depend upon the purpose and structure of the job (see other examples in Section 14.3). In Figure 14.1, the body of the job deck consists of a Fortran source deck, terminated by an EOR card, followed by the data cards to be read as input by the executing program. Note that the EOR card

may be omitted for the last record of the job deck since it is implied by the special pink EOI (end-of-information) card that ends the deck. For detailed information about control statements, see Calidoscope Control Statements.

When the job is introduced into the computer, it is placed in an input queue along with other jobs waiting to be processed. A job scheduler decides the order in which jobs from the input queue will be executed. When the job is brought to execution, the operating system uses the control statements to direct the processing of the job.

First, all the cards following the first EOR card are given the fileset name INPUT. Then the system begins processing the control cards in order. In our case, the first card encountered is the 'RUN' card. This card causes the system to load the RUN Fortran compiler and start it executing.

The RUN compiler begins processing cards from the INPUT fileset. Its task is to convert the Fortran source statements into a form which can be used by the loader and/or to inform the user of any errors it may detect in the program(s). The compiler stops processing cards when it encounters the EOR card. It produces a listing on a fileset called OUTPUT to be printed. The listing contains information about the program(s) compiled, including any error diagnostics. If there were no fatal errors, the compiler also produces a version of the program in a form suitable for the loader (called relocatable object code) on a fileset named LGO. If the RUN compiler found an error which prevented it from compiling a usable program, it aborts, which sets a flag called the current error flag. It then returns control to the operating system.

The system tests the current error flag and, if it is on, skips control statements until an error control statement is found. If no error control statements are found the operating system terminates the job. However, if the flag is not set, it proceeds to process the next control statement, in our case, the LGO[1] card. The LGO card causes the system to begin execution of a program named CLDR (the CALIDOSCOPE Loader).

The loader begins processing the relocatable binary information from the fileset LGO, which is where the RUN compiler left the converted form of the Fortran program. The task of the loader is to arrange the program in memory so that it can be executed. After it has gotten the subprogram(s) from the LGO fileset into memory, it performs a library search for additional subprograms

---

[1] From Load and Go into execution.

which are explicitly or implicitly required in order to run the program. (The library contains subprograms to perform frequently used operations; these subprograms are already in relocatable object code form.) For instance, the program may explicitly use the COS function. If so, the loader finds the subprogram for computing cosines in the library and loads it into memory.

It also happens that a Fortran PRINT statement compiles code which calls a subprogram called OUTPTC[2] so that programs containing PRINT statements implicitly call OUTPTC. The loader finds OUTPTC in the library and puts it into memory too. If the user has also accidentally misspelled the name of a function in one of his Fortran statements (perhaps he wrote SINE instead of SIN), then (unless he provided his own subprogram named SINE) the loader will search the library for SINE. Failing to find it, the loader will place a warning message in the OUTPUT fileset. In any case, it produces a load map, describing where in memory it loaded the various subprograms.

Should the loader run into a sufficiently bad problem, it will abort. If no major problems were encountered, the loader will begin execution of the program it has loaded into memory. Execution will begin, in our case, with the first executable statement in the program OVERFL. At this point, the first data card is wating to be read from the INPUT fileset. What happens next is entirely up to the program which the user wrote. It may generate answers (correct or incorrect') on the OUTPUT fileset and then terminate normally, or it may 'blow up' and terminate abnormally for any of a number of reasons. Some of the most popular abnormal terminations are due to calling a function with illegal arguments, which terminates with a Fortran execution diagnostic (see 15.4.2), using the value of a variable which has never been set, which may generate a mode 4 error (see 15.4.4), or using a bad index when referencing an array, which results in a mode 1 error (see 15.4.4) if you are lucky and causes mysterious failures if you are unlucky.

When the job finishes (normally or abnormally) the OUTPUT fileset is placed in an output queue along with the output from various other jobs waiting for a printer to become available. The next section discusses the output from a sample job in some detail.

To summarize the handling of a typical job in the system, we distinguish three phases:

1. entering the input queue and waiting for a chance to execute

---

[2] From Output of Coded information.

2. executing

3. waiting in the output queue for an available printer and then being printed.

The execution phase of our sample job can be further subdivided into three steps:

a. compilation
b. loading
c. execution of the program.

It is important to distinguish these steps because errors may occur during any of them, and the significance of a particular error is highly dependent on which phase of the job the error applies to. It is also possible to save computer time by bypassing some steps, notably compilation, when circumstances permit (see 14.3).

14.2 Output of a sample job

This section describes the output of a sample job. The sample uses a RUN Fortran program which does not solve a 'real' problem. Rather, it demonstrates a variety of different features, including a number of errors. The program contains many Fortran comments explaining different aspects of the program. It also prints explanatory comments on the output.

The printer output has been reproduced on the next 10 pages about a factor of 2 smaller than it would appear on a line printer. Numbered comments have been added on the right to explain various features of the output.

The printed output is a combination of the following pieces: MSFILE, Job Log, source listing, compiler storage map, load map, and execution output. These aspects are discussed in sections following the output. Besides printed output, a job may have card output and/or magnetic tape output. These are also discussed briefly.

```
16.48.12 H: LP  04  BEGIN OUTPUT
SAMPLE PROGRAM        9997     200
SAMPLE PROGRAM        9997     200
SAMPLE PROGRAM        9997     200
SAMPLE PROGRAM        9997     200
SAMPLE PROGRAM        9997     200
```

```
SAMPLE PROGRAM        9997     200          AG  10/06/73
SAMPLE PROGRAM        9997     200          AG  10/06/73
SAMPLE PROGRAM        9997     200          AG  10/06/73
SAMPLE PROGRAM        9997     200          AG  10/06/73
SAMPLE PROGRAM        9997     200          AG  10/06/73
```

C U R R E N T S Y S T E M M E S S A G E S

04 AUG 73 ***** CLDR AS STANDARD LOADER FOR ALL *****
WE PLAN TO ESTABLISH THE CALIDOSCOPE LOADER, CLDR, AS THE STANDARD LOADER DURING THE WEEKEND OF AUGUST 18 AND 19. THIS MEANS THAT CLDR WILL NORMALLY BE USED TO PROCESS LOAD, LGO, EXECUTE, AND NOGO CONTROL STATEMENTS. POTENTIAL AREAS OF DIFFICULTY ARE DISCUSSED IN THE AUGUST NEWSLETTER. WE ANTICIPATE THAT THEY WILL IMPACT LESS THAN ONE PERCENT OF THE JOBS RUN. PLEASE REPORT ANY PROBLEMS NOT INCLUDED IN THIS LIST TO THOS SUMNER, 221 EVANS HALL, OR TO THE PROGRAMMING CONSULTANT, 217 EVANS. IF DIFFICULTY ARISES, THE CDC LOADER MAY BE USED BY INSERTING THE CONTROL STATEMENT LOADER,PPLOADR IMMEDIATELY AFTER THE JCB IDENTIFICATION STATEMENT.

16 SEP 73 INPUT PASSWORDS
AN INPUT PASSWORD FEATURE WAS INSTALLED TODAY FOR TESTING. 7-8-9 CARDS WHICH CONTAIN THE LETTERS ≠PW=≠ WILL BE IGNORED OR CAUSE THE JOB TO BE ABORTED WITH THE MESSAGE ≠BAD PASSWORD≠ OR ≠JOB DELETED. INCORRECT INPUT PASSWORD≠. THIS SHOULD NOT AFFECT ANYONE USING VALID 7-8-9 CARDS, BUT ANYONE HAVING UNUSUAL PROBLEMS WITH 7-8-9 CARDS SHOULD SEE THE CONSULTANT OR GREG SMALL (223 EVANS). ADDITIONALLY, A NEW NON-FATAL CARD READER MESSAGE, ≠ERROR ON 789 CARD≠, WILL OCCURR FOR 7-8-9 CARDS WHICH DO NOT STRICTLY CONFORM TO THE EOR CARD SPECIFICATION.

18 SEP 73 NOGO PROBLEM
ON MONDAY 17 SEPT, THERE WAS A PROBLEM WITH NOGO LOAD SEQUENCES WHICH RESULTED IN JOBS RUNNING TO THEIP TIME LIMITS. THIS PROBLEM HAS BEEN FIXED AND ANYONE WHOSE JOB EXPERIENCED IT SHOULD BRING THEIR JOB≠S OUTPUT TO THE CONSULTANTS OFFICE FOR A REFUND.

18 SEP 73 NEW JCB CARD DEFAULTS / ONLINE BALANCE
AS OF ABOUT 1100 ON 18 SEP 73, THE FOLLOWING NEW JOB CARD STANDARD FIELDS WERE PUT INTO THE JET. AS USUAL, THESE VALUES MAY BE CHANGED BY APPLICATION TO THE ACCOUNTING DEPARTMENT IN 239 EVANS HALL. ALSO AT THIS TIME THE NEW ONLINE BALANCE WAS TURNED ON. SEE THE POSTED DISCRIPTION AT THE INPUT DESK OR ON THE BCARD OUTSIDE 88 EVANS.

FIELD		JOBS 0000 - 7999: DEFAULT	JOBS 0000 - 7999: MAXIMUM	JOBS 8000 - 9999: DEFAULT	JOBS 8000 - 9999: MAXIMUM
TAPES	(D)	0	3	0	0
TIME	(B)	4	1000	2	20
CM FL	(B)	40000	120000	40000	120000
PAGES	(D)	25	1000	25	100
PUNCH	(D)	100	10000	100	1000
ECS FL	(B)	0	304000	0	0

22 SEP 73 PUBLIC FILESET CHANGES
ASS: TEXT360
REPLACE: ALPHAC, TSP

22 SEP 73 NEW TSP
A NEW TSP COMMON FILE VERSION WILL BE IN EFFECT AS OF SEPTEMBER 22. THE NEW VERSION REQUIRES ECS AND ALSO ALLOWS A LARGE DECREASE IN CM FIELD LENGTH. THE TYPICAL SMALL JOB WILL WORK WITH 70000B FIELD LENGTH AND 20B ECS. A FULLER EXPLANATION IS GIVEN IN A HANDOUT, AVAILABLE IN 673 EVANS.

04 OCT 73 ATTENTION GDS USERS
A NEW ERROR CHECKING FACILITY HAS BEEN ADDED TO GDSLIB WHICH EXAMINES THE SPECS VALUES ON USER CALLS TO DETERMINE IF THEY HAVE BEEN INITIALIZED. IF ANY REQUIRED VALUES ARE UNINITIALIZED, THE JOB WILL ABORT AS BEFORE BUT WITH A TRACEBACK INSTEAD OF THE OLD MODE 4 ERROR. UNFORTUNATELY, AT LEAST ONE ERROR EXISTS IN THE IMPLEMENTATION. IF YOU CALL ANY OF THE ROUTINES FABLIX, FABLIY, FAGLIX, OR FAGLIY, YOUR PROGRAM WILL CURRENTLY ABORT IF YOU HAVE NOT SET SPEC(1) THROUGH SPECS(10). THIS WILL BE CORRECTED ON MONDAY. YOU CAN AVOID THE PROBLEM IN THE INTERIM BY SETTING THESE SPECS VALUES TO ZERO. IF ANY OTHER PROBLEMS ARISE PLEASE CONTACT JOHN WELLS, X2-1410 OR BILL INGRAM X2-1724.

06 OCT 73 CONSULTING HOUR CHANGE
EFFECTIVE MON. OCT. 15 THE PROGRAMMER CONSULTANT HOURS WILL BE: 10-12 A.M. AND 1-5 P.M. MON.-FRI.

06 OCT 73 HAZELTINE DEMONSTRATION
THERE WILL BE A DEMONSTRATION OF THE FOLLLOWING NEW PRODUCTS FROM HAZELTINE ON TUES. AND WED. OCT. 16 AND 17 10 A.M. TO 5 P.M. IN ROOM 251 EVANS: 1000 AND 2000 TERMINALS, TWO TYPES OF PRINTERS (IMPACT AND THERMAL), AND A TAPE CASSETTE.

06 OCT 73 INTERESTED IN PERMANENT FILESETS
THE PERMFILE SYSTEM AND DOCUMENTATION ARE ESSENTIALLY READY FOR RELEASE. HOWEVER, WE CURRENTLY HAVE PRECIOUS LITTLE DISK STORAGE TO ALLOCATE TO PERMFILES. CONSEQUENTLY, WE ARE ASKING CUSTOMERS WHO ARE INTERESTED IN USING PERMFILES TO FILL OUT A FORM ESTIMATING THEIR SPACE REQUIREMENTS. WE HOPE TO FORMULATE A FAIR POLICY BASED ON THIS SURVEY AND WE MAY ACTUALLY ALLOCATE SPACE BASED ON THE RETURNED FORMS. FORMS ARE AVAILABLE IN 239 EVANS. PLEASE TRY TO COMPLETE A FORM BY 15 OCT IF INTERESTED.

PAGE 1 OF SAMPLE OUTPUT. OUTPUT IDENTIFICATION, JOB SEPARATORS, AND MSFILE.

```
XXXXXXXXXXXXXXXXXX                  XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXX
XXXXXXXXXXXXXXXXXX                  XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXX
XXXXXXXXXXXXXXXXXX                  XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXXXXXXXXXX                 XXXXXXXXXXX
1XXXXXX10XXXXXXX20XXXXXXX30XXXXXXX40XXXXXXX50XXXXXXX60XXXXXXX70XXXXXXX80XXXXXXX90XXXXXXX100XXXXXXX110XXXXXX120XXXXXX130XX
```

```
XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXXXXXXXXX                XXXXXXXXXXXX
```

(1)
10/06/73 - CALIDOSCOPE (SCM) VER.01.2-A 07/21/73 MACHINE A
(2) 16.48.00 H: CR 05 J9997.   SAMPLE PROGRAM
16.48.04 H: CR 05   81 CARDS INPUT
16.48.04 $:JOB J9997AG AT CTLPT 3.

16.48.04 *:RUN,,,,,,,,,CR.
16.48.04 $:CM=16384(40000B), EC=0, CP=0, PP=0.172, SP=0
16.48.05 W   1 WARNING IN OVERFL
16.48.05 *:LGO.
16.48.06   BEGIN OVERFL , CP TIME LOADING   .249
16.48.07 $:CM=4544(10700B), CP=0.536, PP=2.320, SP=0.043
16.48.07 F:ARITHMETIC ERROR MODE 2 AT ADDRESS 003250
16.48.07 I: INFINITE OPERAND USED
16.48.07 I: OPERAND MAY HAVE RESULTED FROM A DIVISION BY ZERO (3)
16.48.07 I: REFERENCE TO WORD 001026 WHOSE VALUE IS INFINITE

16.48.07 $:JOB COMPLETED. CP=0.553, PP=2.853, SP=0.048
16.48.07 $:PRINTED LINES = 227, PUNCHED CARDS = 0 (4)
16.48.08 $:EFFECTIVE TIME = 1.104 SEC, JOB COST = $0.105

16.48.08 I:FUNDS REMAINING = $41.366

JOBLOG

(1) DATE AND (2) TIME WHEN THE JOB ENTERED THE SYSTEM.

(3) MESSAGE GIVING ABSOLUTE ADDRESS + 1 AND OTHER INFORMATION ABOUT A FATAL ERROR. SEE 15.4.4 AND 15.5.

(4) MESSAGES GIVING THE APPROXIMATE AMOUNT OF OUTPUT GENERATED AND THE APPROXIMATE COST OF THE JOB.

RUN FORTRAN COMPILER VERSION 2.3 B.3 06 OCT 73 16:48:04 PAGE NO. 1

```
                PROGRAM OVERFL(INPUT,OUTPUT,TAPE5=INPUT)
              C..THIS SAMPLE PROGRAM DEMONSTRATES SEVERAL FEATURES OF RUN FORTRAN
(5)           C..LISTINGS, INCLUDING A VARIETY OF ERRORS.  BECAUSE OF THE HIGH DENSITY
              C..OF ERRORS, IT SHOULD NOT BE TRUSTED FOR EXAMPLES OF HOW TO DO THINGS.
000003                COMMON VARD(30)/BLOK1/VARNAM(50),VARNM2(100),VARNM3(200)
              C..READ A FROM A CARD AND PRINT IT FOR LATER VISUAL CHECKING.
000003                READ(5,1) A
000011          1     FORMAT (E10.0)
000011                PRINT 6, A, A
000021          6     FORMAT(*0A = *G10.4,*, BUT IF THE PRINT FIELD IS NOT LARGE ENOUGH*
                     1*, A PRINTS LIKE THIS *F10.4)
              C..GENERATE AN INFINITE VALUE B BY DIVIDING A BY ZERO.  THEN PRINT B.
000021                B=A/0.0
000023                PRINT 2,B,B
000033          2     FORMAT  (*0DIVIDING BY ZERO RESULTS IN AN INFINITE VALUE, WHICH PR
                     1INTS AS * G7.4,* OR OCTAL *021)
              C..THE FOLLOWING STATEMENT TESTS THE VALUE OF B AND BRANCHES TO
              C..     30 IF B IS INDEFINITE
              C..     20 IF B IS IN RANGE
              C..     10 IF B IS INFINITE
000033                IF (LEGVAR(B)) 30,20,10
              C..AN ELEMENT IN THE VARNAM  ARRAY IS GIVEN A VALUE. A SUBSEQUENT
              C..REFERENCE TO THE SAME ELEMENT MISSPELLS THE VARIABLE NAME, CAUSING
              C..THE COMPILER TO MISTAKE IT FOR AN EXTERNAL FUNCTION REFERENCE.
              C..ALSO, THERE IS NO WAY FOR THE NEXT STATEMENT TO BE EXECUTED AND THE
              C..COMPILER FLAGS THIS ANOMALY WITH A NON-FATAL DIAGNOSTIC.
(6) 000037            VARNAM(5)=5.0
***NP ******************************
000040                C=VARNM(5)
000043          10    CONTINUE
000043                CALL WORK(ARGP,16.)
000045                C=ARGP
              C..BECAUSE THE SUBROUTINE WORK ALWAYS TAKES THE SQUARE ROOT OF ITS
              C..SECOND ARGUMENT, THIS CALL CAUSES A NON-FATAL EXECUTION TIME ERROR
              C..DETECTED BY THE SQUARE ROOT FUNCTION.
000047                CALL WORK(ARGP,-1.5)
000051                PRINT 11,ARGP,ARGP
000061          11    FORMAT (*0SQRT OF NEGATIVE NUMBER IS AN INDEFINITE VALUE, WHICH PR
                     1INTS AS * G7.4,* OR OCTAL *,021)
              C..THE NEXT STATEMENT CAUSES WORK TO MULTIPLY BY AN INFINITE NUMBER,
              C..A FATAL ERROR WHICH TERMINATES THE JOB.
000061                CALL WORK(ARGP,B)
000063                GO TO 30
              C..THIS CODE HANDLES THE CASE OF B LEGAL (WHICH CANT HAPPEN IN
              C..THIS EXAMPLE).
000064          20    CONTINUE
              C..THIS CODE HANDLES THE CASE OF B INDEFINITE (WHICH CANT HAPPEN IN
              C..THIS EXAMPLE).
000064          30    PRINT 31
000070          31    FORMAT (*0 HELP - THERE HAS BEEN A HIDEOUS DISASTER*)
000070                END
```

PAGE 3 OF SAMPLE OUTPUT

SOURCE LISTING OF THE MAIN PROGRAM OVERFL PRODUCED BY THE RUN COMPILER

(5) THIS COLUMN GIVES THE RELATIVE LOCATION OF THE CODE COMPILED FOR EACH STATEMENT.

(6) A COMPILER DIAGNOSTIC INDICATING AN ERROR IN THE PREVIOUS STATEMENT. SEE (10) BELOW.

CROSS REFERENCE MAP-OVERFL

PROGRAM LENGTH INCLUDING I/O BUFFERS
002237

STATEMENT FUNCTION REFERENCES (7) (9)

(8) LOCATION GEN TAG SYM TAG REFERENCES

STATEMENT NUMBER REFERENCES

LOCATION	GEN TAG	SYM TAG	REFERENCES					
000102	C00010	1	000003					
000121	C00027	2	000023					
000105	C00013	6	000011					
000043	L00022	10	000036					
000140	C00046	11	000051					
000064	L00035	20	000035	000036	000063			
000064	L00035	30	000035	000036	000063			
000153	C00061	31	000064					

BLOCK NAMES AND LENGTHS
- 000036 BLOK1 - 000536

VARIABLE REFERENCES

LOCATION	GEN TAG	SYM TAG	REFERENCES					
000167	V00005	A	000006	000014	000016	000021		
000172	V00010	ARGP	000043	000045	000046	000054	000056	000061
000170	V00006	B	000022	000026	000030	000033	000061	
000171	V00007	C	000042	000046				
000000C01	A00001	VARD	NONE					
000000C02	A00002	VARNAM	NONE					
000062C02	A00003	VARNM2	NONE					
000226C02	A00004	VARNM3	NONE					

START OF CONSTANTS
000072

START OF TEMPORARIES
000161

START OF INDIRECTS
000167

UNUSED COMPILER SPACE
002600

1 WARNING IN OVERFL

NP********NO PATH TO THIS STATEMENT (10)
000037

PAGE 4 OF SAMPLE OUTPUT

COMPLETE COMPILER STORAGE MAP FOR MAIN PROGRAM OVERFL.

(7) STATEMENT LABELS USED IN THE PROGRAM

AND

(8) THE RELATIVE LOCATION OF THE CODE COMPILED FOR THE STATEMENT

AND

(9) A LIST OF THE LOCATIONS WHERE THE STATEMENT IS REFERENCED.

(10) AN EXPLANATION OF THE DIAGNOSTIC OCCURRING AT LOCATION 37 IN THE SOURCE LISTING.

```
                SUBROUTINE WORK(S,X)
              C..
              C..THIS SUBROUTINE IS SET UP TO MAKE LOTS OF MISTAKES FOR THE MAIN
              C..PROGRAM.
              C..
              C..ON THE FIRST CALL, X=16.0 IS LEGAL AND EXECUTION IS NORMAL
              C..ON THE SECOND CALL, X=-1.5 AND THE SQUARE ROOT FUNCTION DETECTS
              C..    A NON-FATAL EXECUTION ERROR
              C..ON THE FINAL CALL, X IS INFINITE AND THE MULTIPLICATION CAUSES A
              C..    FATAL APITMETIC ERROR
              C..
              C..WORK ALSO DECLARES SOME VARIABLES THAT IT DOES NOT USE.  THE
              C..VARIABLES IN COMMON SERVE ONLY TO DEMONSTRATE CERTAIN FEATURES OF
              C..BLANK AND LABELED COMMON.  THE VARIABLE DEMO IS DEFINED SO THAT AN
              C..UNINITIALIZED VARIABLE WILL OCCUR IN THE DUMP.
              C..
000005              COMMON VARC(100),VARD(30)/BLOK1/VNAME1(50),VNAME2(100),VNAME3(200)  (11)
                   1/BLOCK2/XXX(100)
000005              INTEGER DEMO
000005              X1=X
000005              X2=2.*X1
000007              S=SQRT(X)
000013              RETURN
000013              END
```

Page 5 of Sample Output

Source listing for the subroutine WORK produced by the RUN compiler

(11) Note that WORK defines 3 COMMON blocks, blank COMMON and labeled blocks BLOK1 and BLOCK2, See (12) below. Also note that if WORK and OVERFL referenced VARD(1), they would not get the same memory location because blank COMMON is allocated differently in the two routines.

```
CROSS REFERENCE MAP-WORK

SUBPROGRAM LENGTH
000026

STATEMENT FUNCTION REFERENCES

      LOCATION  GEN TAG   SYM TAG   REFERENCES

STATEMENT NUMBER REFERENCES

      LOCATION  GEN TAG   SYM TAG   REFERENCES

BLOCK NAMES AND LENGTHS
      -  000202    BLOK1  -  000536    BLOCK2 -  000144
(12)
VARIABLE REFERENCES

      LOCATION  GEN TAG   SYM TAG   REFERENCES
      000023    V00011    DEMO      NONE
      000000C01 A00001    VARC      NONE
      000144C01 A00002    VARD      NONE
      000000C02 A00003    VNAME1    NONE
      000062C02 A00004    VNAME2    NONE
(13)  000226C02 A00005    VNAME3    NONE
      000000C03 A00006    XXX       NONE
      000024    V00012    X1        000006
      000025    V00013    X2        000007

START OF CONSTANTS
000015

START OF TEMPORARIES
000017

START OF INDIRECTS
000023

UNUSED COMPILER SPACE
003100
```


PAGE 6 OF SAMPLE OUTPUT

COMPLETE COMPILER STORAGE MAP FOR SUBROUTINE WORK.

(12) BLANK COMMON IS INDICATED HERE BY A BLANK NAME FOLLOWED BY A LENGTH OF 202B. BLOK1 IS THE SECOND COMMON BLOCK DECLARED AND BLOCK2 IS THE THIRD.

(13) THIS LINE SAYS THAT VNAME3 IS LOCATED AT ADDRESS 226B RELATIVE TO THE BEGINNING OF THE SECOND COMMON BLOCK DECLARED, I.E., BLOK1.

```
BEGIN CLDR2.2F   6 OCT 73 16:48:06 , CPU TIME USED IS     .283 SECONDS


 BLOCK       ORIGIN (15)  LENGTH      DATE       TIME      PROCESSOR  FILE
/BLOK1  /    000100       000536
 OVERFL      000636 (14)  002237     06 OCT 73 16:48:04    RUN2.3B3   LGO
/BLOCK2 /    003075       000144
 WORK        003241       000026     06 OCT 73 16:48:04    RUN2.3B3   LGO
 INPUTC      003267       001025     20 APR 73 19:59:46    CMP1.1A4   SUBRLIB
 IOMSG$      004314       000226     12 NOV 70 12:32:16    CMP1.1A4   SUBRLIB
 IO.SUP      004542       001326     20 APR 73 19:59:46    CMP1.1A4   SUBRLIB
 LEGVAR      006070       000005     12 NOV 70 12:32:16    CMP1.1A4   SUBRLIB
 OUTPTC      006075       001203     20 APR 73 19:59:46    CMP1.1A4   SUBRLIB
 SQRT        007300       000043     02 DEC 71 01:04:33    CMP1.1A4   SUBRLIB
/SYS.MEM/    007343       000005
 SYSTEM      007350       001024     20 APR 73 19:59:46    CMP1.1A4   SUBRLIB
 SYS.MAP     010374       000021     DIAGNOSTIC STORAGE MAP
//           010415       000202 (16)


UNSATISFIED REFERENCES :

ENTRY     DECK     LOCATION : FROM DECK AT RELATIVE (ABSOLUTE) ADDRESSES


VARNM     -             007461: OVERFL    000041 (000677)    (17)

REQUIRES 027357B WORDS TO LOAD, 010617B WORDS TO EXECUTE
```

PAGE 7 OF SAMPLE OUTPUT

LOAD MAP

(14) 636 IS THE LOAD ADDRESS FOR OVERFL AND 3241 IS THE LOAD ADDRESS FOR WORK.

(15) 100 IS THE LOAD ADDRESS FOR BLOK1 AND 3075 IS THE LOAD ADDRESS FOR BLOCK2

(16) THE LENGTH OF BLANK COMMON IS 202 AS DECLARED IN WORK, NOT 36 AS DECLARED IN OVERFL.

(17) A PROGRAM VARNM IS REFERENCED AT AT ABSOLUTE ADDRESS 677 WHICH IS RELATIVE ADDRESS 41B IN OVERFL, I.E., IN THE STATEMENT

```
C=VARNM(5)
```

WHICH BEGINS AT RELATIVE ADDRESS 40B IN OVERFL.

```
A =  .1235E+06, BUT IF THE PRINT FIELD IS NOT LARGE ENOUGH, A PRINTS LIKE THIS **********

DIVIDING BY ZERO RESULTS IN AN INFINITE VALUE, WHICH PRINTS AS       R OR OCTAL  3777000000000000000

NEGATIVE ARGUMENT
ERROR NUMBER      39 DETECTED BY SQRT    AT ADDRESS 007323
CALLED FROM WORK     AT 003251 = 000010 IN WORK
CALLED FROM OVERFL   AT 000706 = 000050 IN OVERFL

SQRT OF NEGATIVE NUMBER IS AN INDEFINITE VALUE, WHICH PRINTS AS       I OR OCTAL  1777000000000000000
```

MEMORY DUMP AUTOMATICALLY PRODUCED BECAUSE OF THE FATAL ARITHMETIC ERROR

```
DMP - EXCHANGE PACKAGE DUMP - CALLED AT 16:48:07 ON 10/06/73 BY SYSTEM - INFINITE OPERAND USED

 P = 000000   RA = 143500   FL = 010700   EM = 07   ECS RA = 01121000   ECS FL = 00000000   MA = 013743

X0 = 77777 77777 77770 00000   A0 = 004643  C(A0) = 00000 00000 00000 00055  B0 = 000000  PP CALL = OPE 23 000000 002053
X1 = 00000 00000 10260 00000   A1 = 006175  C(A1) = 00000 00000 00000 02053  B1 = 001030  C(B1) = 17770 00000 00000 00000
X2 = 37770 00000 00000 00000   A2 = 001026  C(A2) = 37770 00000 00000 00000  B2 = 001026  C(B2) = 37770 00000 00000 00000
X3 = 17214 00000 00000 00000   A3 = 003256  C(A3) = 17214 00000 00000 00000  B3 = 004542  C(B3) = 33232 12224 55170 65516
X4 = 37770 00000 00000 00000   A4 = 004554  C(A4) = 33333 30000 00000 00000  B4 = 000000  C(B4) = 00020 03250 00000 00000
X5 = 00000 00000 00000 00007   A5 = 004555  C(A5) = 00000 00000 00000 00007  B5 = 000001  C(B5) = 00000 00000 00000 00000
X6 = 37770 00000 00000 00000   A6 = 003265  C(A6) = 37770 00000 00000 00000  B6 = 000001  C(B6) = 00000 00000 00000 00000
X7 = 37770 00000 00000 00000   A7 = 003260  C(A7) = 00000 00000 10260 01030  B7 = 007256  C(B7) = 00000 00000 00000 00017

$0 = 00000 00000 00000 00000   $1 = 00000 00000 00000 00000   $2 = 00000 00000 00000 00000   $3 = 00000 00000 00000 00000
$4 = 00000 00000 00000 00000   $5 = 00000 00000 00000 00000   $6 = 00000 00000 00000 00000   $7 = 00000 00000 00000 00000

SYSTEM COMMUNICATION AREA

000000    00020 03250 00000 00000    00000 00000 00000 00000    11162 02524 00000 01031    17252 42025 24000 02053
000004    24012 00540 00000 01031    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000
                                  WORDS 000010 TO 000063 ALL CONTAIN  00000 00000 00000 00000
000064    14071 70000 00000 00000    00000 00000 00000 10617    00000 00004 00000 00100    00000 00000 00000 00000
000070    17260 52206 14570 00000    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000
000074    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000

PP PARAMETER AREA

002050    60007 77777 02004 02050    60007 77777 02004 02051    60007 77777 02004 02052    17252 42025 24000 00015
002054    02000 00000 40140 02074    00000 00000 00000 02160    00000 00000 00000 02074    41140 62000 01000 03075
002060    00000 00000 00000 00003    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000
002064    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000    00000 00000 00000 00000
002070    00000 00000 00000 00000    41000 00000 00000 00000    00000 00000 00000 00000    00000 00601 00000 00200

P-40 THROUGH P+40

                                  WORDS 003210 TO 003237 ALL CONTAIN  60007 77777 02004 *****
003240    60007 77777 02004 03240    27172 21300 00000 00002    04000 00721 00000 00000    43052 76710 15770 76120
003244    15110 20122 36717 46000    51700 03260 56220 10622    51300 03256 10466 46000    51600 03265 40734 66120
003250    51700 03266 46000 46000    01000 07301 07010 03241    51500 03260 63150 21522    63250 56610 04000 03242
003254    51100 03257 10711 46000    01000 07474 07000 03241    17214 00000 00000 00000    00000 00000 00000 00000
003260    00000 00000 10260 01030    60007 77777 02004 03261    60007 77777 02004 03262    60007 77777 02004 03263
003264    60007 77777 02004 03264    37770 00000 00000 00000    60561 77777 77777 77777    11162 02524 03000 00000
003270    04000 00644 00000 00000    04000 00647 00000 00000    04100 03304 06100 03274    76610 51600 03356 46000
003274    01000 03360 07000 03267    51200 03356 05020 03271    76600 54620 51600 04772    50660 00001 51200 04774
003300    03220 03271 43001 15720    54720 50220 00002 46000    51100 03352 52110 00005    73710 53720 04000 03271
003304    51100 03271 10611 46000    51600 03270 46000 46000    01000 05217 07000 03267    66100 51500 03353 46000
003310    01000 05073 07030 03267    01000 06060 46000 46000    76620 76120 51600 03352    61620 00005 61200 04542
```

```
16.49.33 H: LP  04    1081  LINES OUTPUT
SAMPLE PROGRAM        9997    200
SAMPLE PROGRAM        9997    200
SAMPLE PROGRAM        9997    200
SAMPLE PROGRAM        9997    200
SAMPLE PROGRAM        9997    200
```

```
SAMPLE PROGRAM        9997    200        AG  10/06/73
SAMPLE PROGRAM        9997    200        AG  10/06/73
SAMPLE PROGRAM        9997    200        AG  10/06/73
SAMPLE PROGRAM        9997    200        AG  10/06/73
SAMPLE PROGRAM        9997    200        AG  10/06/73
```

PAGE 10 OF SAMPLE OUTPUT
FINAL SEPARATOR PAGE

14.2.1 The MSFILE and Job Log

The first page of each output printed at the Computer Center has page separators printed across the folds of the paper to make it easier for the output from different jobs to be identified. The body of this page contains MSFILE - notices giving current information relevant to Computer Center operations, e.g., bugs found (or fixed), features added (or to be deleted), upcoming events, etc.

The Job Log (sometimes called the "Dayfile") contains an entry for each significant occurrence during the running of the job. It constitutes the first page of the job output proper. All control statements executed are reproduced here, along with other messages from the system, from the operators, etc., most of them self-explanatory. Probably the most important point for the user to note is that many (but not all) error messages appear here. (See section 15.4.3.)

The first line of the Job Log appears as follows:

mm/dd/yy - CALIDOSCOPE (SCM) vvv.xx mm/dd/yy MACHINE A

where the first mm/dd/yy is the date on which the job was run, vvv is REC if the System has recovered from a failure, and VER otherwise, xx is the release number of the version, the second mm/dd/yy is the date of the most recent system modification.

Lines after the first have the following general format (some exceptions are noted in section 15.4.3):

hh.mm.ss ac messages

where hh.mm.ss gives the time of day in hours, minutes, and seconds since midnight.

The next field, ac, contains a pair of characters which indicate the type and origin of the messages, respectively. a may be any one of the following:

Character	Message Type
*	Control statement
O	Operator
I	Informational
W	Warning
F	Fatal Error
b	User comment
$	Accounting
S	Statistic

E	Error in Operating System
/	Hardware error
|	Continuation of previous message
H	Message from HYDRA (the program which control reading and printing of jobs)
G	Generated Control statement

c is either ":" or blank depending upon whether the message originated from a peripheral processor program which is part of the Operating System or from a Central Program, respectively.

The message itself follows. If the message is longer than 120 characters, it will be continued on the next line after a repetition of the time.

A line by line interpretation of the Job Log from the sample output follows:

```
16.48.00 H: CR  35 J9997.    SAMPLE PROGRAM
16.48.04 H: CR  05     81  CARDS INPUT
16.48.04 $:JOB J9997AG AT CTLPT 3.
```

The Job deck was read in at 16.48.00 on the card reader with logical number 5 and 81 cards were in the deck. The job name J9997AG was given and the job entered execution at control point 3.

```
16.48.34 *:RUN,,,,,,,,,CR.
16.48.36 $:CM=16384(40000B),EC=0,  CP=0,  PP=0.172,  SP=0
16.48.37 W    1 WARNING IN OVERFL.
```

The RUN card is reproduced (the CR option was specified in order to provide a sample of the cross-reference listing - a simple 'RUN' is adequate for most purposes --see section 15.1). The accounting message indicates that at the time the job entered execution, it requested a Central Memory field length of 16384 (decimal) words, the equivalent of 40000 (octal) words, but did not call for any Extended Core Storage. The number of seconds of Central Processor and Peripheral Processor time required up to that point was zero and .172 respectively. System Processor time used was 0. During compilation, one warning message was generated by the program OVERFL.

```
16.48.05 *:LGO.
16.48.06   BEGIN OVERFL, CP TIME LOADING  .249
16.48.07 $:CM=4544(10700B),  CP=0.536,  PP=2.320,  SP=0.043
```

The LGO control statement is reproduced followed by a message indicating the initial entry name of the program and the central processor time used in the loading process. This is followed by a

second accounting message indicating the CM field length was automatically reduced (after loading and prior to execution) to 4544 (decimal) words, the equivalent of 10700 (octal) words. To this point, Central Processor, Peripheral Processor and System Processor time consumed were .536, 2.320, and .043 seconds respectively. ECS is not listed because it did not change.

```
16.48.07 F:ARITHMETIC ERROR MODE 2 AT ADDRESS 003250
16.48.07 I:  INFINITE OPERAND USED
16.48.07 I:  OPERAND MAY HAVE RESULTED FROM A DIVISION BY ZERO
16.48.07.I:  REFERENCE TO WORD 01026 WHOSE VALUE IS INFINITE
```

Execution of the main program OVERFL began at 16.48.06 and within a second an execution error was found which made it undesirable for execution to continue. The next four messages indicate the nature of the fatal error and various suggestions as to what may have caused it. (See Sections 15.4.3, 15.4.4, and 15.5.)

```
16.48.07 $:JOB COMPLETED.   CP=0.553,  PP=2.853,  SP=0.048
16.48.07 $:PRINTED LINES =  227, PUNCHED CARDS = 0
16.48.07 $:EFFECTIVE TIME 1.104 SEC,  JOB COST = $0.105
16.48.08 I:FUNDS REMAINING = $41.366
```

The last four messages on the Job Log are accounting messages. The first indicates the total amounts of Central Processor (CP) time, Peripheral Processor (PP) time, and System Processor (SP) time required for executing the entire job (reading, compilation, loading, execution, printing). All time values are in decimal. The second message estimates the print and punch output generated, excluding Job Log and MSFILE messages. The last two messages give the time used for calculating the job cost, followed by the cost itself and, finally, the funds remaining in the account (this is not printed for all accounts). The formula for calculating the cost of a job may be found in the Guide to Computer Center Services.

14.2.2 Source Listing

The compiler normally reproduces each source statement on the OUTPUT fileset.[1] Each executable statement is prefixed with a relative location (relative address) which indicates where the object code generated for that statement begins relative to the subprogram origin. The locations are given in octal. The subprogram origin has a relative address of zero but is assigned an absolute address (relocated) during loading.

Statements, such as FORMAT, COMMON, DIMENSION, etc., which are non-executable, i.e., do not generate object code, are listed with the relative location of the next executable statement in the subprogram. A CONTINUE statement which is not used as the terminator of a DO loop will be treated similarly.

All cards with a C punched in column 1 are printed as comments and otherwise are ignored by the compiler. The compiler may, however, alter the ordr of some comments with respect to source statements.

All local variables and constants used by a suprogram (excluding formal parameters) are assigned to relative locations beginning at the point where the subprogram object code ended. Variables assigned to a common block are given locations relative to the beginning of the block. Common blocks are given numbers in order of appearance.

Compiler diagnostics may also appear in the source listing. The sample contains an 'NP' diagnostic following the line at relative address 000037. See 15.4.1 for a discussion of compiler diagnostics.

14.2.3 Compiler Storage Map

After each subprogram has been successfully compiled (no fatal errors have been diagnosed) and listed, the subprogram length is printed. In the sample output, it is 2237 (octal) CM words, including space for the I/O buffers (usually 1001B words each), for the main program OVERFL and 26B for the subroutine WORK.

If the references option on the RUN control statement has been specified, an extended subprogram storage map which lists references is then printed. Under the heading STATEMENT FUNCTION REFERENCES (there are none in either subprogram) are given the relative addresses assigned to programmer-defined arithmetic statement functions together with the generated compiler tag assigned and the relative addresses of all references to each function listed. Under STATEMENT NUMBER REFERENCES are listed the

---

[1] The listing produced during compilation may be suppressed by inserting a statement with the word NOLIST starting in column 7 before the PROGRAM, SUBROUTINE, BLOCK DATA, FUNCTION or IDENT (for Compass subprograms) statement. The listing remains turned off for all succeeding subprograms until listing is restored by a statement containing the word LIST beginning in column 7. NOLIST is overridden whenever fatal errors (not warnings) occur.

statement label numbers used in the subprogram. The programmer-assigned label or name is SYM TAG. Under GEN TAG is the name which the compiler has generated to use to refer to the symbol. Statement labels serving as branch points carry the prefix L. FORMAT statements and constants carry the prefix C. In the main program, OVERFL, for example, the statement which has been labelled 10 is compiled beginning at relative location 43B. The compiler has assigned that location a label, namely L00022, and a reference was made to it from relative location 36B. The location given for the statement label number identifying a FORMAT statement is the location in which the display code for the format itself, the alphanumerics between and including the outer parentheses, is stored; it is not the relative location listed with the statement in the source listing. Thus FORMAT statement number 1 is stored at location 102B. It is given the label C00010 by the compiler and a reference is made to it by an I/O statement at relative location 3, a READ statement.

Under the next heading BLOCK NAMES AND LENGTHS are listed the blank and labelled common blocks and their lengths. Thus in the sample output, the compiler storage map for subroutine WORK shows three common blocks: blank common with a length of 202B; BLOK1 with 536B words, and BLOK2 with 144B words. As can be seen from the load map for the sample program, blank and labelled common blocks are stored in different areas of the program's central memory field.

Under VARIABLE REFERENCES are listed the relative addresses of most programmer-defined variable names actually referenced and where they are referenced. The compiler-generated label (GEN TAG) is prefixed with a V in the case of a simple variable, or with an A, for an array name. Locations given for variables in COMMON are suffixed by Cnn, where nn denotes the order of the particular common block as listed under BLOCK NAMES AND LENGTHS and the address portion indicates the relative position with respect to the start of the COMMON block. Thus in subroutine WORK, the array XXX is located at the beginning (000000) of the COMMON block 03 (BLOK2).

The programmer should bear in mind that because of the way the compiler operates not all references will be listed. An actual physical reference is necessary before the reference is placed in the reference map. If the required variable address is already in a register, the compiler will use the address in the register and not make an actual variable reference by name. Since subprogram formal parameters are always located in pointer registers their references are never listed. A reference to a statement number will not be listed if an actual jump is not necessary, such as when the code simply falls through to the next statement and the compilation of a jump instruction is therefore not necessary. On

the other hand, the code compiled for a logical IF statement usually involves a jump to the next statement so a reference may appear for it even though there is no reference in the source poogram.

Following the list of VARIABLE REFERENCES is printed the starting location for the constants used in the program for locations used as temporary storage during calculations, etc., and for the area in which the variables are stored.

The compiler storage map has many uses. The correct spelling of variable identifiers can be checked and misnamed variables caught (except when an array reference is misspelled, in which case the compiler thinks it is a function reference and, depending on how the reference is used, one of several error messages may result). As changes are made in the subprogram, new statement numbers and variable names can be checked for a previous use. The map is used extensively when interpreting dumps.

If the references parameter is not specified in the RUN statement, an abbreviated compiler storage map is produced which omits the GEN TAG and REFERENCES listing.

The last line of the compiler output indicates the 2600B locations were unused in compiling the main program and 3100B were unused in compiling the subroutine. Thus, the field length specified on the Job card could be reduced if it would not fall below the 40000B word minimum required by the compiler.

14.2.4 The Loader and the Load Map

Here we digress briefly to describe some aspects of the operation of the loader prior to describing the load map which it generates on the output. In addition to the loader discussion in this chapter, the writeup L3 CAL CLDR describes other loader features which allow more control of the loading process, use of auxiliary libraries, etc. Some of this information is also to be found in CALIDOSCOPE Control Statements.

The loader is a program in the operating system which accepts a translator-generated object code program as input and places it in the memory of the central processor in proper form for execution. The following information and concepts are useful in understanding how the loader performs its task.

14.2.4.1 What the loader loads and where it finds it

As each source subprogram in the source deck is successfully translated by RUN, a copy of the generated relocatable object code program is stored (usually in the fileset named LGO). Additional

object code subprograms generated at another time (using the P parameter on the RUN card, for example) may come from the user's input deck (from fileset INPUT) or other filesets generated by the user. Control statements (described in CALIDOSCOPE Control Statements) tell the loader where to find the relocatable object code subprograms which it is to use to construct the executable object program and specify the order in which they should be loaded. Loading is performed in the same order in which the filesets are specified and, within each fileset, in the order in which the object programs appear.

14.2.4.2 The relocation of subprograms and COMMON blocks

The object subprograms are read into contiguous locations in core, starting at a particular address, usually 100B, which is the initial load address.

As a subprogram is loaded, the code and each relative address reference within it is relocated. That is, the subprogram origin (load address for the particular subprogram) is added to the address on the left margin of the source listing to determine where the code generated for that statement is to be stored. The load address is also added to the memory addresses within the code. Thus the internal program references remain consistent but refer to the actual addresses in which the program is stored. The load address is the memory address (with respect to the beginning of the memory field assigned to the job) of the beginning location of that subprogram. For subprograms after the first, this is one plus the last address of the preceding subprogram or labeled common block.

Labeled common blocks are loaded as separate items. Each block is placed immediately in front of the first subprogram which declares it and is allotted the length declared for it in that subprogram. Therefore, if a longer block with the same label is declared in a subprogram which is loaded later, a fatal diagnostic message is issued. If a subprogram which declares a shorter block than actually allocated is loaded, a warning diagnostic message appears. The blank common block is loaded directly after the last subprogram loaded (including subprograms loaded from the library to satisfy external references). It is allotted the maximum length declared for it (which need not be the length given in the first subprogram in which blank common is declared).

14.2.4.3 Communication linkage between subprograms

During loading, each reference to an address which is external to a given subprogram (as in a CALL or FUNCTION reference) must be replaced by the relocated external address. This process is

termed linking. For example, in the sample program, the subroutine WORK, which has its origin at 3241B and ends at 3266B calls the function SQRT which comes from the standard subprogram library and has its origin at 7300B. SQRT is external to the program WORK.

The address to which an external reference is linked is called an entry point, i.e., a location in a subprogram which is the start of a given section of the executable code and can be referenced by other subprograms. A subprogram may have more than one entry point. Each PROGRAM, SUBROUTINE, FUNCTION, or ENTRY statement generates an entry point for the subprogram in which it appears[1].

14.2.4.4 Standard Subprogram Library

Any entry point which is referenced but missing from the user's program is automatically searched for in the standard subprogram library. A copy of the subprogram which contains that entry (if found) is then loaded into the user's memory field after the user's program and is linked to it. This program in turn may call for still others from the library fileset. Note that only a missing entry name will be requested from the library; thus, the SQRT library function will not be loaded if a subprogram named SQRT is supplied by the user.

Many library subprograms are referenced by the code generated by the RUN Fortran compiler as it compiles the object code for many of the Fortran statement types. For example, input/output (e.g., READ, WRITE) statements generate calls to library subprograms which perform I/O and format conversion. These can be seen (e.g., INPUTC, OUTPTC) in the load map. Section 15.6.1 lists all such names which may conflict with Fortran subprogram names. Chapter 11 contains a general description of the contents of the standard subprogram library.

14.2.4.5 Unsatisfied References

If a referenced entry cannot be found either among the user's subprograms or in the library, it is called an unsatisfied reference. Execution is allowed in such cases. However, in instances where a reference is unsatisfied unintentionally [e.g., if the user's program leaves out a subroutine, or neglects to dimension an array (in an expression, a reference to an element of

---

[1] In general, an entry point indicates what a subprogram has available; an external reference indicates what a subprogram needs but does not have available. The loader attempts to satisfy all of each subprogram's needs using what is available.

an array has the same syntax as a function call and will be interpreted as such by the Fortran compiler)], and an attempt to execute the non-existent subprogram is made a fatal error occurs (see section 15.4.2).

14.2.4.6 Memory Backgrounding

The locations in memory used to store the object code for the program's instructions and the locations whose contents are specified in DATA statements are initialized by the program. The contents of all other areas used by the program (e.g., COMMON areas, locations reserved for variables) are preset to a standard value, which has the following form: the upper 30 bits contain a negative indefinite real value (6000777777B) and the lower 30 bits contain a jump to an address of the form 400000B + xxxxxxx, where xxxxxxx is the address of the word itself. For example, the location 076340B would contain 6000777777702004763 40B. This value is intended to halt execution with a fatal error if the program uses it in a floating point calculation or attempts to execute it as an instruction. When such a halt occurs, the program is attempting to use 'undefined' numbers and the form of the number frequently helps identify the problem. There is no hardware facility to check for the use of an undefined integer quantity; so this error may manifest itself in obscure ways. For practical purposes this means a variable should <u>never</u> appear to the right of an equal sign unless it either appears in a prior statement to the left of an equal sign, has had a value assigned to it by a DATA statement, or appears in the list of a prior READ statement.

14.2.4.7 Error Detection in the Loader

Some error conditions are detected by the loader. Some are fatal (e.g., no main program), others are not (e.g., unsatisfied references). All detected errors are signalled by some (hopefully) explanatory message. However, some error conditions are not detected by the loader. For instance, the loader can not check to be sure that the actual parameter lists given to a subprogram agree with what is expected by the subprogram in its formal parameter list.

14.2.4.8 The Load Map

As the loader loads each subprogram, it writes a map of the program on the OUTPUT fileset. The first line of the load map:

```
BEGIN CLDR2.2F 6 OCT 73 16:48:06, CPU TIME USED IS .283 SECONDS
```

identifies the version of the loader in use, gives the date and time the load operation began and the amount of central processor time used within the job prior to the load. In this case, the

loader version is CLDR2.2F, the load began on 6 OCT 73 at 16:48:06 after using .283 seconds of CP time in the job.

The second line of the load map contains the column headings for the map itself. The significance of these headings is:

BLOCK — This column lists the names of the program blocks and COMMON storage blocks in the order in which their space is allocated in memory. Common block names are surrounded by slashes.

ORIGIN — This is the address within the central memory field where the block begins.

LENGTH — This is the length of the block.

DATE — This is the date of the creation of the object code, e.g., translation by RUN from a source program.

TIME — This is the time of the creation of the object code. Together with the DATE field, this field makes it possible to verify that the correct version of a program was actually used.

PROCESSOR — This is the identification of the processor used to produce the object code.

FILE — This is the name of the fileset from which the subprogram was loaded. The standard subprogram library is named SUBRLIB.

The DATE, TIME, PROCESSOR, and FILE fields appear only for program blocks. A special block named SYS.MAP is generated by the loader itself and is identified by the words "DIAGNOSTIC STORAGE MAP" in the above fields.

The blank COMMON block appears at the end of the load map and is identified by a block name which consists only of a double slash.

Error diagnostic messages may appear interspersed with the load map to indicate the points at which errors were detected.

Following the load map itself, a list of unsatisfied references appears in the sample program. This section will not appear if there are no unsatisfied references. In this case the reference to VARNM found at relative location 41B in program OVERFL is not satisfied. It has been given an arbitrary definition at 7461B which is a diagnostic printing procedure for this condition contained in library subprogram SYSTEM.

The final message from the loader states that the process of loading the sample program requires 27357B words of memory to hold the program, the loader, and the loader's working tables while the program is being loaded. However, once execution of the program begins, only 10617B words will be required. Since memory is allocated in blocks of 100B words the effective figures are 27400B and 10700B respectively.

All addresses used in the load map are in octal and are relative to the starting location of the user's memory field, considered as location 0.

14.2.4.9 Memory Allocation

The executable program is constructed within the central memory field. The sample program illustrates how large an area of memory is used for even a small program. In the diagram of the load map, the user program code and common blocks are seen to occupy only about 1100 cells. Three large areas occupy most of the diagram (see Figure 14.2). They are:

a. The input/output buffers

The size of the main program, OVERFL, is due to large blocks of memory for buffers for filesets declared on the PROGRAM card. These buffers are allocated within the main program area by the compiler. Determination of the buffer length is as described in Section 7.4. The purpose of the buffers is described in Section 15.2.

b. The library routines

The RUN Fortran compiler generates code which calls in large library subprograms to handle communication, data movement and conversion, etc. The bulk of the space shown for library routines in the diagram is used for reading cards and printing output.

c. The loader itself

The loader program and its tables effectively occupy the high addresses of the memory field during the loading process. No object program can be loaded into the same space. However, blank common can be placed there. The programmer who is pressed for core space during loading may possibly gain about 20000B words by placing arrays in a blank common area of at least that length.

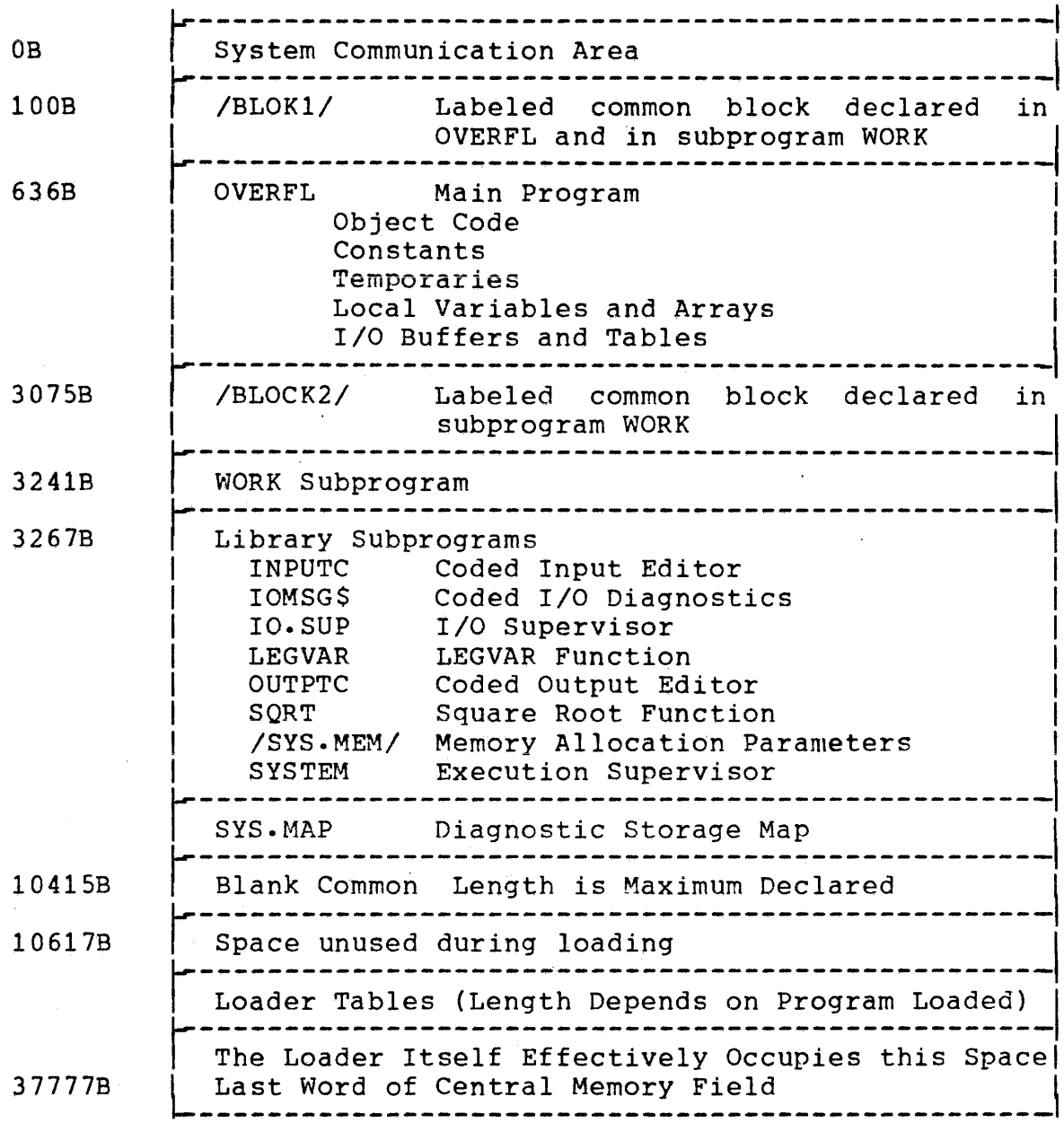


Figure 14.2 Central Memory Layout for Sample Program

14.2.5 Execution

If a Fortran program is loaded with no fatal errors, control is transferred to the first executable statement of the main program[1]. Execution then proceeds as dictated by the statements of the source program.

Caution: when execution begins, variables not assigned a specific value in the program (including variable subscripts and variables in labeled common), have a predetermined value as described in Section 14.2.4.6.

As part of execution data may be read from any fileset specified and information may be written on any unprotected fileset specified. Normally,

1. data (if any) are read (in the order in which they are arranged in the data deck) from the fileset INPUT as specified by READ and FORMAT statements;

2. information to be printed is placed on the fileset OUTPUT as specified by PRINT and FORMAT statements. Information is printed 56 lines per page, unless otherwise specified by a carriage control character (see Section 10.1). There is automatic skip over the page fold unless suppressed by a carriage control character.

Execution Printout: The Sample Program was designed to illustrate execution time errors. The execution print lines were generated at the points in the program noted below. Each point is identified by the subprogram name and the relative location (in parentheses) given in the source listing for that subprogram. Execution begins with the READ statement in OVERFL (relative location 3) and proceeds sequentially except as noted below. Blank lines are not itemized in the table.

---

[1] If more than one main program is loaded (a rather strange thing to do), control is transferred to the last one loaded unless the programmer explicitly directs otherwise.

Print line	Location of PRINT	FORMAT Used	Comment
1	OVERFL(11B)	6	The value read into variable A from a data card is printed out by OVERFL (as a check for correct input) using G format conversion. It is also printed with too little space under F conversion to show what happens when a field is too small.
2	OVERFL(23B)	2	The program reached this PRINT statement only because B is an infinite value as determined by the LEGVAR library subroutine. B is printed with both G and O format conversions to illustrate that an infinite prints as an R when a numeric specification (I,F,E,G) is used. (Section 9.3)
3-6	SQRT		OVERFL calls the subroutine WORK to compute the square root of the second parameter and places it in the first parameter. In the first call to WORK at relative location 43, the return is made without a message being printed since the call was legal. In the second call however, at relative location 47, the value whose square root is to be calculated is negative. This illegal argument was deliberately passed in order to illustrate this execution time error message, which is explained further in Section 15.4.2. SQRT returned an indefinite value as the function value.
7	OVERFL(51B)	1	An explanatory comment is printed by OVERFL. The indefinite value returned is printed with both G and O format conversions. Note that the indefinite prints as I when a numeric conversion specification is used.

The last call of WORK in OVERFL (61) caused execution to terminate with an arithmetic error - use of an infinite value (for variable B) in a real expression.

This error exit triggered the dump which follows. Note that no execution time error message was printed via the fileset OUTPUT for the ARITHMETIC ERROR because it was sensed by the 6400's

internal circuitry (hardware) rather than by a program (software). This error and interpretation of the dump are discussed in Sections 15.4.4. and 15.5.

14.2.6 Punched Output

Two types of punched card output may be generated during a job. The first is binary cards containing the compiled object program if requested; the second is output from the execution phase of the job, usually coded (Hollerith) cards, generated by the PUNCH statement. These are punched on orange top striped cards by the system, with "picture" cards at the beginning for identification purposes. The picture card has columns 1-3 and 78-80 completely punched, and the job identifier (the first 4 numbers of the job name plus a 2-digit sequence number) punched in block letters on the card so that they can be interpreted visually. The picture cards should be removed from the output decks. A sample picture card with the job identifier 99980Y is shown below.

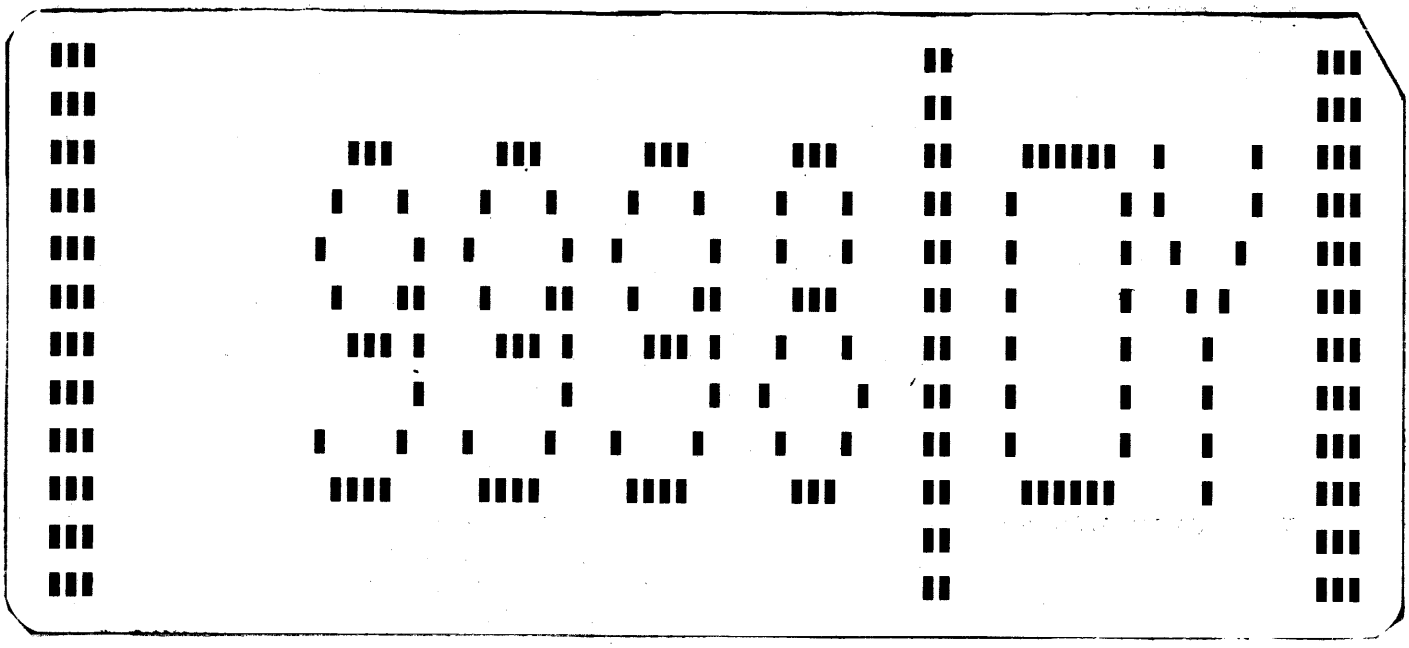


Figure 14.3 Picture Card Separator

Binary Cards: Binary cards containing object code from compilations, if any, are terminated by a 7-8-9 level 17B card, and a 6-7-8-9 card. The latter should be discarded. Each object code subprogram is terminated by a 7-8-9 card which must be left as part of the deck.

Each subprogram deck is sequentially numbered in binary in the lower six rows of column 79 and in column 80, beginning with the

number 1. Thus, card 1 of the deck has a 9 punch in column 80; card 2, an 8 punch; card 3, 8 and 9 punches; card 4, a 7 punch; card 5, 7 and 9 punches, etc.

Card 1 (a 9 punched in column 80) of each subprogram deck contains the subprogram name in display code (Appendix A, Table H), two characters per column, in columns 8-11. In the card shown below, column 8 contains 1,9 punches; column 9 contains 11,0,3,7 and 8 punches. Starting at the top (row 12) of column 8 and reading the holes in groups of 3, we obtain 000 100 000 001 (binary) or 0401 octal. Continuing similarly for the other columns, the subprogram name becomes 04013106B. Looking up the 2-digit octal codes in Appendix A, Table H, gives DAYF for the subprogram name.

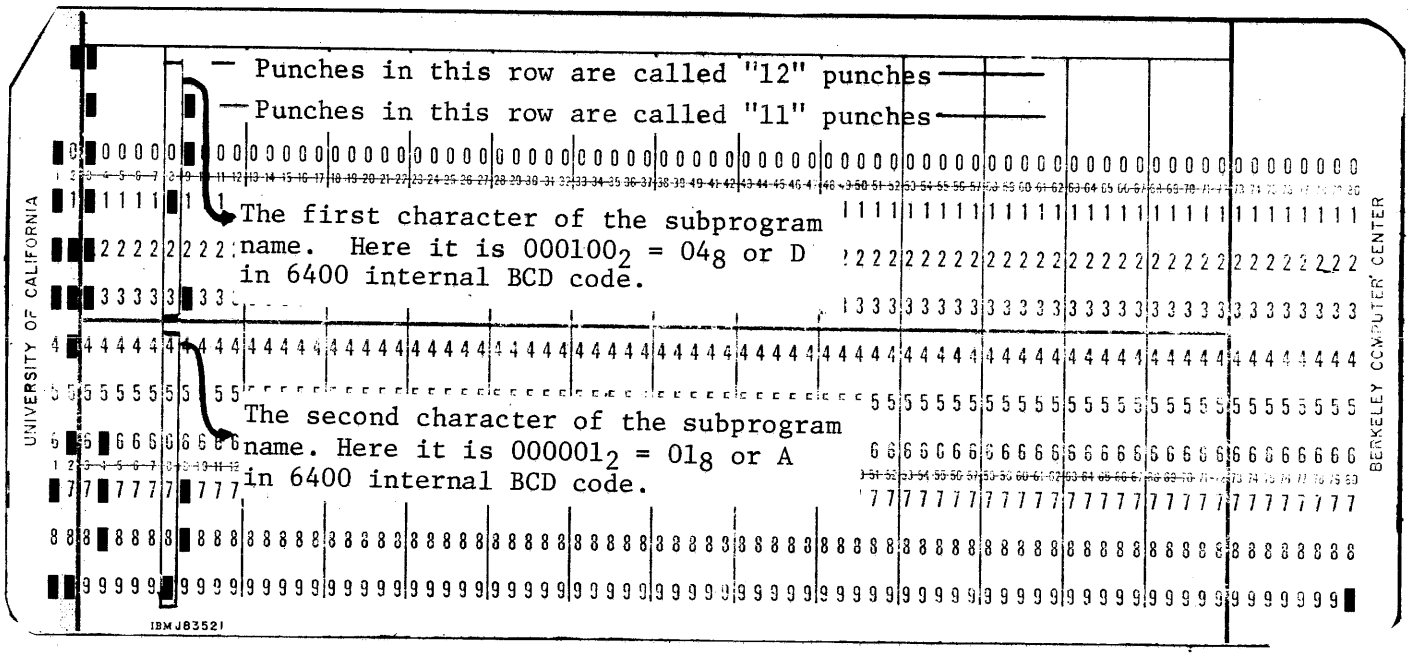


Figure 14.4 Sample Binary Card

Execution Output: The deck of coded cards produced by PUNCH statements (if there are any) on the fileset PUNCH during execution is terminated by a 6-7-8-9 card (6,7,8,9 punches all in column 1) which should be discarded.

It is also possible to produce binary punched output during execution by performing a binary (unformatted) write on a logical unit whose fileset name has been equivalenced to the fileset PUNCHB on the PROGRAM statement (or as described in Section 13.0.2). In this case, the binary output follows the binary cards from the compilation, if any, with no intervening picture card.

Binary (PUNCHB) cards are punched in this manner:

columns 1-2,	checksum and word count
columns 3-77	15 binary words, 5 columns per word,
columns 78-80,	binary card sequence number.

Note: All cards punched during a job are counted in the punched card limit specified in the job card. If this limit is exceeded, the job is aborted with a message, and the punched output is terminated with a 6-7-8-9 card. If the operator terminates the punching, the punched output ends with a picture card with ENDED in block letters.

14.2.7 Magnetic Tape Input/Output

The input/output statements, discussed in Chapter 10, are used to control the reading and writing of information on magnetic tape; the form in which information is recorded on tape is described in Section 13.2.3.

Information may be recorded in either binary (unformatted) or coded (formatted) mode on tape. For efficient use the binary form is recommended; since the information is written on the tape exactly as it appears in memory, no time-consuming conversions need to be performed. However, for compatibility with other computers, or initial entry of data to the system, coded is the preferred mode.

Temporary filesets (sometimes called "scratch" or "work" files) which are written and read within a particular job should also, for efficiency, be written in binary mode. Temporary files are usually not written on reels of magnetic tape, but are recorded on magnetic disk storage; they are discarded at the end of the job in which they are used. No special provision need be made for them except declaring them in a Fortran program on the PROGRAM statement (Section 7.4). As far as the programmer is concerned, they behave as actual tapes except that rewind is almost instantaneous.

Filesets which are assigned to actual tape, either for program input or output, must be declared in the PROGRAM statement and appear on a REQUEST card. (See CALIDOSCOPE Control Statements).

14.3 Other Job Setups

Besides the deck shown in Section 14.1 to compile and execute a Fortran program, the following sample decks may be useful to the Fortran programmer. A more complete description of control statements is given in CALIDOSCOPE Control Statements.

14.3.1 Load a program from pre-compiled binary decks and execute

Once a program has been successfully compiled, a deck may be set up to run the program without redoing the compilation. This saves the cost of compilation in cases where 'production' runs are to be made with different sets of data.

To do this, one first obtains a copy of the relocatable object code form of the program on cards by specifying the P option on the RUN card (see 15.1 and 14.2.6) (remember the punched output limit on the job identification statement). The following deck may be set up to execute the cards that the compiler punches:

Deck	Remarks
Job card	
LGO,INPUT.	Load binary decks from fileset INPUT (job deck) and execute.
7-8-9 card	End of control card record.
Binary program decks	
7-8-9 card	follows program deck in addition to the 7-8-9 card following the last binary card.
<Data cards>	If any are to be read from fileset INPUT.
6-7-8-9 card	EOI (End-of-Information)

14.3.2 Compile and load some routines, load others from pre-compiled relocatable binary decks, and execute

In some cases, a problem may have been broken up into a number of subprograms some of which may be gotten into a satisfactory state while others are still being debugged. In this situation, the cost of recompiling the subroutines which are already known to be operating correctly can be avoided. First, obtain relocatable binary decks for the subprograms which are correct by using the P option on the RUN card (see 15.1 and 14.2.6). Next, remove the source decks for these routines, leaving only the ones that need to be recompiled. Then use the following deck setup:

Deck	Remarks
Job card	
RUN.	Compile source statements.
LOAD,LGO.	Load compiled routines.
LGO,INPUT.	Load binary routines from INPUT fileset and execute.
7-8-9 card	End of control cards.
source deck	
7-8-9 card	End of source deck.
binary decks	
7-8-9 card	In addition to 7-8-9 following last binary card.
<data cards>	If any.
6-7-8-9 card	EOI (End-of-Information)

Note:

If duplicate program names are loaded via a LOAD card (from either LGO or INPUT filesets), only the first is used; a warning message is given. Thus, with this deck setup, the compiled subroutines are loaded first; they will override any decks of the same name which might already exist in the precompiled subprograms.

15.1 The RUN Control Statement

The general form of the RUN statement is:*

RUN,[*mode*],[*pl*],[*bl*],[*input*],[*output*],[*lgo*],[*epl*],[*asa*],[CR],[NA]

The most important parameters for most users are *mode* and CR. For the remaining parameters, the default is normally used.

mode Compiler mode option

- S compile and print out source listing (this is the default).
- P compile, print out source listing, and punch binary decks of the object code generated (see Section 14.2.6).
- L same as S with the generated object code for each source language statement listed using COMPASS mnemonics.
- M same as P with the generated object code for each source language statement listed using COMPASS mnemonics.

pl In the current implementation of the system, this parameter field is ignored.

bl standard object program input/output buffer lengths (if not overridden in PROGRAM statement). If omitted, 1001_8 is assumed.

input name of the fileset from which the RUN compiler program reads its input. If not specified, INPUT, which is the disk fileset consisting of information read in from the card reader, is assumed.

output The name of the fileset on which the RUN compiler is to write the source listing portion of its output. If omitted, OUTPUT is assumed.

lgo The name of the fileset on which the RUN compiler program is to write the binary machine code that is to be loaded, not that to be punched. If omitted, LGO is assumed. The fileset is not rewound prior to use by the compiler, thus allowing several compilations from different RUN statements to be placed on the fileset.

epl This field is ignored under the current system.

---

* This format is subject to change.

asa if zero, the normal ASA I/O list and format interactions are suppressed at execution. Use of this feature is not recommended. (See Section 9.7)

CR If specified, the reference table for each subprogram is printed after the source listing on the OUTPUT fileset. This is often convenient when the mode parameter is L or M.

NA If specified, and a fatal compilation error occurs, the job step will not set the current error flag and control statements will continue to be processed.

Examples:

RUN. Compile (i.e., load and execute RUN compiler program) with source list.

RUN,L,,,TAPE3,,,,,CR.

Compile with source listing and COMPASS mnemonics for the generated object code. The source language statements are read from fileset TAPE3 rather than INPUT (i.e., the card reader). A reference table is to be printed for each subprogram.

15.2 The FET and I/O Buffer Area

The compiler assigns to each distinct (not equivalenced) fileset named in the PROGRAM statement an area, following the machine code for the main program, to be used as a communication area and buffer for the fileset. The communication area is called the Fileset Environment Table (FET) and directly precedes the fileset buffer it describes. This table (currently 17 words long) provides the name of the fileset, an area where I/O requests and their statuses are posted, and pointers to indicate where information should currently be placed in the buffer or read from it. The buffer holds information waiting to be written on a fileset or input records awaiting a READ reference within the program. It serves to compensate for the difference in the speed at which the Central Processor executes the program and the slower speed at which the physical I/O device sends or receives data.

On output, when a record is sent to an I/O device, the area of the buffer which contained that record is freed. On input, an area for the physical record must be available within the buffer in order for data to be transmitted from the device.

The buffer for a fileset is emptied when an ENDFILE or REWIND is issued for it. When execution is terminated normally or for fatal library subprogram errors (Section 15.4.2), the information in the buffers of all output filesets is transmitted to the I/O devices. On an abnormal termination, such as ARITHMETIC ERROR (Section 15.4.4) or TIME LIMIT, information may still be in the buffers awaiting output. After an abnormal termination, the system empties (flushes) the buffers for any filesets with dispositions (such as OUTPUT), provided the FET is intact and no limit (such as print limit) is exceeded.

Buffer sizes for filesets are determined by the PROGRAM statement (Section 7.3). The standard buffer size for a fileset is 1001_8 words which is the minimum for a binary magnetic tape fileset. (The minimum buffer size for a disk fileset is 101_8 words.) But with a bigger buffer area, the program may not have to wait (i.e., stop the sequence of calculations) for space in the buffer to be freed, thus reducing the amount of PP time used.

15.3 RUN-COMPASS Subroutine Linkage

Since it may be convenient or necessary for some subroutines or functions called by a program to be written in COMPASS, the manner in which COMPASS subprograms should be constructed and how RUN compiled Fortran subprograms communicate with them is important.

If COMPASS subprograms are to be compiled along with Fortran-coded subprograms, the first line of each COMPASS subprogram must be one on which the first 10 columns are blank and columns 11-15 contain the characters IDENT. The last line is one with the characters END in columns 11-13 and the remainder blank. For efficient compilation, COMPASS subprograms should be grouped together after the subprograms written in Fortran language.

The RUN-COMPASS linkage is explained by an example followed by comments. The example consists of a Fortran main program and two subroutines. The Fortran statements are followed by the machine-language instructions which they generate, suitably commented. The Fortran statements are underlined to help distinguish them from the machine code.

```
PROGRAM MAIN(INPUT,OUTPUT)

        ENTRY   MAIN                    This declares the location
                                        MAIN to be an entry point,
                                        i.e., accessible from outside
+       VFD     42/OLMAIN,18/102B       entry/exit trace (see notes below)
MAIN    SB1     *-1
        SB2     C00001
        RJ      =XQ8NTRY                initialize FETs, etc.

CALL PHD(A,B,C)

        SB1     A                       address of first parameter to B1
        SB2     B                       address of second parameter to B2
        SB3     C                       address of third parameter to B3
+       RJ      =XPHD                   call PHD with three parameters.
-       VFD     6/07,6/3,18/MAIN-1      linkage and trace information (see
                                        notes below)
```

```
CALL PEN(M,N,O,P,Q,R,S,T,U,V)

      SB1     M                    The first six
      SB2     N                    parameters' addresses
      SB3     O                    are passed by way
      SB4     P                    of the
      SB5     Q                    first six
      SB6     R                    B-registers.
      SA1     -1+=XPEN             fetch entry/exit trace from
                                   PEN and
      SB7     X1-6                 calculate number of para-
                                   meters it expects less 6.
      SX6     S                    Pass address of parameter 7
      SA6     A1-B7                to indicated word in PEN.
      SX7     T                    Pass address of parameters
      SA7     A6+1                 8, 9, and 10 similarly.
      SX6     U                    Up to a total of 60 parameters
      SA6     A7+1                 can be passed using the
      SX7     V                    RUN calling
      SA7     A6+1                 sequence.
+     RJ      =XPEN                call with 10 parameters
-     VFD     6/07,6/10,18/MAIN-1  linkage and trace information
                                   (see notes below)
END

SUBROUTINE PHD(A,B,C)

      ENTRY   PHD                  This declares PHD to be an
                                   entry point, i.e., accessible
                                   from outside.
      VFD     42/OLPHD,18/3        entry/exit trace line (see
                                   notes below)
PHD   BSS     1                    Entry/exit word for RJ instruc-
      :                            tion in calling program
      :                            :
RETURN

      EQ      PHD                  RETURN

END
```

```
SUBROUTINE PEN(A,B,C,D,E,F,G,H,I,J)

      ENTRY    PEN                    This declares PEN to be an
                                      entry point

G     BSS                             address of parameter 7

H     BSS                             address of parameter 8

I     BSS                             address of parameter 9

J     BSS                             address of parameter 10

      VFD      42/OLPEN,18/10         entry/exit trace line (see
                                      notes below)

PEN   BSS      1                      Entry/exit word for RJ instruc-
      :                               tion in calling program
      :                               :

RETURN

      EQ       PEN                    RETURN

END
```

The linkage performs three functions: passing parameters, providing return information, and providing traceback information (to be used in case of error).

Parameter passing. In executing a call, the caller places the addresses of the parameters in B-registers and words in the called subprogram. The address of the first parameter (if any) is put into B1, the address of the second parameter (if it exists) is placed in B2, and so on up to the 6th parameter. If more than 6 parameters are being passed, the addresses of parameter 7,...,n < 61 are placed in words preceding the entry point of the called subprogram. Note that the number of parameters being passed is available in the linkage and trace information line in the caller and in the entry/exit trace line in the called subprogram (see description below). Also note that both of these parameters counts govern the storage of parameter addresses in the called subprogram; if the counts do not agree, something may be clobbered. RUN compiled subprograms do not automatically check that they are being passed the correct number of arguments but this may be determined by using the library subroutine NARG.

Return address. The RJ instruction executed by the caller transfers to the called subprogram at the location one greater than the entry point. It simultaneously places a transfer in the entry point location which, when executed, will transfer to the (location of the calling RJ) + 1. Thus, execution of the subprogram begins with the location after its entry point and the subprogram can return to the caller by transferring to the subprogram's own entry point.

Traceback information. Information is stored in the word preceding the entry point of all subprograms (the word labeled 'entry/exit trace line' in the example) and in the low 30 bits of each RJ word which calls a subprogram (labeled 'linkage and trace information' in the example) to allow automatic analysis of the paths leading to errors. An example of such analysis is the run-time error message traceback described in Section 15.4.2. The general formats of these words follow:

```
                    entry/exit trace line

                    VFD 42/0Lname,18/nparms
```

name is the name of the subprogram containing the entry/exit trace line (RUN-compiled subroutines contain the name of the subroutine here, for example). *nparms* is the number of parameters expected by this program when it is called (except in a main program, where it is the load address + number of fileset names declared on the PROGRAM statement).

```
                    linkage and trace information

                -   VFD 6/07,6/nparms,18/addr
```

nparms is the number of parameters being passed with this call. *addr* is the address of this subprogram's entry/exit trace line. The minus sign shown on this VFD causes the information to be stored in the low 30 bits of the RJ instruction where it belongs (without the minus, COMPASS would 'force upper' after the RJ and store the VFD information in the word following the RJ.)

Returning parameters. A subroutine returns parameters by resetting the values in the appropriate words whose addresses it has been passed. A function returns its value by placing the value in X6 (and X7 for double precision or complex functions) before returning.

15.4 Error Messages

Various classes of errors are detected during the running of a job. Compiler and assembler programs issue diagnostic messages when errors in rules for forming statements (syntax) are caught. Section 15.4.1 describes the diagnostics produced by the RUN compiler. Various library subprograms, called during the execution of a program, check for mistakes in their usage and issue error messages to the OUTPUT fileset if errors are found; these Fortran execution time error messages are listed in Section 15.4.2. The programs of the Operating System produce messages on the Job Log if they detect errors during execution; these errors are discussed further in Section 15.4.3. Finally, the internal circuitry of the 6400 detects a class of arithmetic errors described in 15.4.4.
It should be noted that some messages from RUN Fortran programs will normally appear in the JOBLOG and do not indicate an abnormal condition. Among these are:

Message	occurs when
END name	an END statement is executed or a RETURN statement is executed in a main program
STOP n name	a STOP n statement is executed
PAUSEn	a PAUSE n statement is executed
EXIT name	a CALL EXIT statement is executed

name indicates the program which executed the statement.

15.4.1 RUN Fortran Compiler Diagnostics

15.4.1.1 During a Fortran compilation by the RUN compiler, 2- or 3-character diagnostic mnemonics follow statements which are incorrect; other diagnostic mnemonics may follow the END statement or the storage map and indicate types of errors in the program which are not attributable to one given statement. RUN Fortran compiler diagnostics are of the form:

```
***XXY**************************************
```

where XX is a two-character error type indicator and Y is either blank or F; a blank signifies a warning diagnostic and F indicates a fatal diagnostic.

Warning diagnostics are given for statements that appear in the wrong order in a subprogram or are inconsistent with a previous statement. The compiler accepts the statements and produces the appropriate object code to perform the indicated operation. Statements so marked should be checked by the programmer for the intended meaning or sequence.

Fatal diagnostics cause the object code for the current subprogram to be deleted from the relocatable binary fileset. If punched binary cards of the object code were requested, none will be produced for the subprogram. The compiler will translate succeeding subprograms, but the program as a whole will not be executed. Also, unless otherwise specified on the RUN card, control statements in the control record will be skipped as described in CALIDOSCOPE Control Statements.

The number of warning diagnostics (m) and fatal diagnostics (n) found in program name is indicated by a message in the JOBLOG such as:

n ERRORS m WARNINGS IN name

When n or m is 1 or 0, the text of the message is suitably modified. *name* is the name of the subprogram in which the errors were detected. The total number of fatal errors in all programs compiled with one RUN statement is indicate in a JOBLOG message thus:

FORTRAN FATAL ERROR TOTAL: n errors

A statement may have more than one diagnostic message associated with it. Also an error in one statement may cause diagnostics to be issued for succeeding correct statements.

For each type of diagnostic message found in a subprogram a further descriptive message appears at the end of the listing of source statements for that subprogram. This message gives a brief explanation of the error, the relative locations where it was found, and is of the form:

XX****** *Explanation of the error*

$rloc_1$, $rloc_2$, $rloc_3$, ...

where XX is the two-character error type indicator found in the diagnostic message and each $rloc_n$ is a relative location associated with the statement in which the error occurred.

The following alphabetical list contains each two-character error type indicator generated by the RUN Fortran compiler with its diagnostic message. Further explanation of the error and a possible reference to a section of this Guide accompanies each item.

Note that "SYNTAX ERROR" refers to a punctuation error such as the incorrect use or absence of a comma, period, slash or parenthesis.

AC******* ARGUMENT COUNT TOO HIGH

Indicates that the number of arguments in this reference to an arithmetic statement function is greater than the number of formal parameters. See Section 7.8.

AE******* ARITHMETIC STATMENT FUNCTION CALLS ITSELF

The arithmetic statement function being compiled references itself.

AF******* ARITHMETIC STATEMENT FUNCTION ERROR

The arithmetic statement function has a statement number or appears after the first executable statement. This diagnostic can also be caused by having a subscripted variable on the left side of a replacement statement without the variable appearing first in a DIMENSIØN, COMMON, or type statement. See Section 7.8 or 5.2.

AL******* SYNTAX ERROR IN ARGUMENT LIST

Indicates that the list of formal arguments of a subprogram is improperly constructed. It may be caused by an incorrect use of a library function. See Section 7.2.1 or Chapter 11. This diagnostic can also be caused by having a subscripted variable on the left side of a replacement statement without the variable appearing first in a DIMENSIØN statement. See Section 7.8 or 5.2. This message will also be issued if an integer constant used for the c parameter in an ENCØDE/DECØDE statement is greater than 150 (Section 10.6).

AS******* SYNTAX ERROR IN ASSIGNMENT STATEMENT

Indicates that the form of an ASSIGN statement is incorrect. See Section 6.1.3.

BC******* SYNTAX ERROR IN OCTAL CONSTANT

Indicates an error in the form of an octal constant; possibly an 8 or 9 occurs.

BX******* SYNTAX ERROR IN BOOLEAN STATEMENT

Indicates an error in the form of a FORTRAN Boolean expression. See Section 3.3.

CB******* LABELED COMMON BLOCKS EXCEED MAX OF 61

More than 61 blocks of storage have been declared labeled common. See Section 5.3.

CD******* VARIABLE DUPLICATED IN COMMON

Indicates that a variable currently being assigned to a common region has been previously assigned to this region. See Section 5.3.

CE******* VARIABLES ASSIGNED TO COMMON ARE IMPROPERLY EQUIVALENCED

Indicates that two variables assigned to common blocks are improperly equivalenced. See Sections 5.3 and 5.4.

CL******* SYNTAX ERROR IN CALL STATEMENT

Indicates that the form of a CALL statement is incorrect. See Section 7.5.

CM******* SYNTAX ERROR IN COMMON STATEMENT

Indicates that the form of a CØMMØN statement is incorrect. See Section 5.3.

CN******* TOO MANY CONTINUATION CARDS

Indicates that more than 19 continuation cards appear in succession or that one such card appears in an illogical sequence. This diagnostic may appear because of a previous diagnostic.

CT******* CONTINUE STATEMENT IS MISSING A STATEMENT NUMBER

All CØNTINUE statements must have a statement number. See Section 6.4.

DA******* DUPLICATE ARGUMENTS IN A FUNCTION DEFINITION STATEMENT

See Section 7.2.1.

DB******* ARRAY SIZE OUT OF RANGE

An array exceeds 131,071 elements.

DC******* SYNTAX ERROR IN A DECIMAL CONSTANT

A FORTRAN decimal constant is incorrectly formed. See Section 2.3.3.

DD******* VARIABLE BEING DIMENSIONED HAS BEEN PREVIOUSLY DIMENSIONED

Indicates a variable has appeared in more than one DIMENSIØN statement, or has been dimensioned previously in a type or CØMMØN statement.

DF******* DUPLICATE FUNCTION NAME

Indicates that the function name in the current arithmetic function definition statement has occurred as the name of a previously defined arithmetic function. May be caused by second use of an array variable which has not been dimensioned (the first use causes the AF diagnostic).

DI******* DO TERMINATOR PREVIOUSLY DEFINED

The terminator of this DØ loop has already been defined, i.e., the statement number terminating the DØ has occurred before the DØ statement.

DJ******* INDEX OF OUTER DO REDEFINED BY INNER DO

The index of an outer DØ loop has been used as the index of an inner DØ loop.

DL******* DECLARATIVE APPEARS AFTER FIRST EXECUTABLE STATEMENT

The declarative statement, a DIMENSIØN, CØMMØN, EQUIVALENCE or type statement, appears after the first executable statement. See Sections 5.1, 5.2 and 5.3.

DM******* SYNTAX ERROR IN DIMENSION STATEMENT

Indicates an error in the form of a DIMENSIØN statement. See Section 5.2.

DN******* ILLEGAL DO TERMINATOR

This statement cannot be used as a DØ terminator. Indicates the attempt to use a FØRMAT, GØ TØ, arithmetic IF, two-branch logical IF, or another DØ statement as the termination statement of a DØ. See Section 6.3.

DO******* SYNTAX ERROR IN A DO STATEMENT

Indicates an error in the form of a DØ statement, for example, when a previously dimensioned variable, or a zero or negative constant is used as an index parameter. See 6.3.

DP******* MULTIPLY DEFINED STATEMENT NUMBER

Indicates the current statement number has previously appeared in the statement number field. See Section 2.2.2.

DQ******* SYNTAX ERROR IN DATA STATEMENT OR APPEARANCE OF UNDIMENSIONED VARIABLE IN DATA STATEMENT

Self-explanatory. Dimensioning information for a variable must precede the DATA statement in which the variable occurs. See Section 5.5.

DR******* DATA RANGE ERROR

The dimension limits of an array are exceeded in a DATA statement. See Section 5.5.

DS******* UNDEFINED STATEMENT NUMBER IN A DO LOOP

The statement numbers in one or more DØ statements have not appeared in any statement number field.

DT******* SYNTAX ERROR IN DATA STATEMENT

Indicates an error in the form of a DATA statement. See Section 5.5.

DU******* AN ATTEMPT WAS MADE TO PRESTORE BLANK COMMON

Data may not be entered in blank common by DATA statements. See Section 5.3.

EC******* CONTRADICTION IN EQUIVALENCE STATEMENT

Indicates that a variable currently appearing in an EQUIVALENCE statement cannot be equivalenced because of an inherent contradiction in the statement.

EF******* END OF FILE CARD ENCOUNTERED, END CARD ASSUMED

Indicates that an EØR (7-8-9) or EØI (6-7-8-9) card is detected before an END card is encountered. This may be caused by an incorrect deck setup (see Section 14.1) or by a mispunch in column 1 of the card.

EM******* SYNTAX ERROR IN INDICATED EXPONENTIATION

Indicates the type of the base or the exponent of an indicated exponentiation process is improper. See Section 3.1.2.

EQ******* SYNTAX ERROR IN EQUIVALENCE STATEMENT

Indicates an error in the form of an EQUIVALENCE statement. See Section 5.4.

EX******* SYNTAX ERROR IN EXPONENT

Indicates an error in the exponent portion of an indicated exponentiation process. See Section 3.1.2.

FA******* FUNCTION HAS NO ARGUMENT

No argument was supplied in a function call.

FL******* SYNTAX ERROR IN EXTERNAL OR F-TYPE STATEMENT

Indicates an error in the form of an EXTERNAL statement. See Section 7.13.

FM******* UNRECOGNIZABLE STATEMENT

Indicates a statement whose type cannot be determined. This error can be caused by mispunching a card, by an erroneous punch in column 6 of a card, or by a missing punch in column 6 of an intended continuation card. It also can be given if there is no PRØGRAM, SUBRØUTINE or FUNCTIØN statement preceding a subprogram.

FN******* NO STATEMENT NUMBER ON FORMAT STATEMENT

Indicates that a FØRMAT statement is missing a statement number.

FP****** DUMMY PARAMETER USED IN COMMON STATEMENT

It is illegal in a given subroutine for a formal parameter and a variable in a common block to have the same name, since this would imply two different uses for the same name.

FS******* ERROR IN SPECIFICATION PORTION OF FORMAT STATEMENT

Indicates that the form of a conversion or editing specification in a FØRMAT statement is in error. See Section 9.3, 9.4, and 9.5.*

FT******* SYNTAX ERROR IN FUNCTION TYPE STATEMENT

Indicates an error in a FUNCTIØN statement. See Section 7.6.

The following nine diagnostics all refer to FORMAT statements:

F1******* LEFT AND RIGHT BRACKETS DO NOT MATCH, OR NEST IS DEEPER THAN FORMAT ((()))

F2******* POINT MISSING FROM D,E,F, OR G CONVERSION

F3******* COMMA NEEDED BETWEEN CONVERSIONS

F4******* COUNT NEEDED BEFORE H FIELD

F5******* ZERO, OR SIGNED, NUMBER ONLY ALLOWED BEFORE P

F6******* H COUNT TOO BIG, OR CLOSING QUOTE OR ASTERISK MISSING

F7******* UNRECOGNIZED CONVERSION CODE

F8******* REPEAT COUNT NOT ALLOWED BEFORE T, ASTERISK, RIGHT BRACKET, QUOTE, OR COMMA

F9******* NON-DIGIT IN FIELD WIDTH SPECIFICATION, OR ZERO FIELD WIDTH

---

* The format scan routines KØDER and KRAKER perform a further check on the validity of format specifications at execution time. See Section 15.4.2.

GF******* FORMAT NUMBER REFERENCED BY CONTROL STATEMENT

A GØ TØ or IF statement references a format statement number.

GO******* SYNTAX ERROR IN A GO TO STATEMENT

Indicates an error in the form of a GØ TØ statement. See Section 6.1.

HC******* HOLLERITH CONSTANT TOO LONG

IC******* CHARACTER NOT IN FORTRAN CHARACTER SET

A character has been used which is not one of the characters in the FORTRAN Source Language Character Set (see Appendix A). Not all occurrences of such characters are detected.

ID******* IMPROPERLY NESTED DO LOOPS

The sequence of DØ loops is improper. Also caused by an error in the construction of a DØ loop or of an implied DØ loop in I/O statements. See Sections 6.3.2 and 9.1.1.

IF******* SYNTAX ERROR IN AN IF STATEMENT

Indicates an error in the form of an IF statement. See Section 6.2.

IL******* SYNTAX ERROR IN AN INDEXED LIST OF I/O STATEMENT

Indicates an error in the form of an indexed list of the current input/output statement. See Section 9.1.

IN******* ILLEGAL FUNCTION NAME

The name of a function reference starts with a numeric character. See Section 7.7.

IO******* ILLEGAL I/O DESIGNATOR

An I/O designator has a variable name of more than 6 characters or a numeric value of more than 99.

IS******* ILLEGAL USE OF PROGRAM SUBROUTINE OR FUNCTION NAME

It is illegal to use the name given to a subroutine or a function as an array name within the subroutine or function definition. A subroutine name may not even be used as a simple variable name within the subroutine.

IT******* ILLEGAL TRANSFER TO DO TERMINATOR

A transfer to a DØ terminator is not allowed if no transfer to it appears before it appears in the physical deck sequence. This can be caused by illegally transferring in a DØ loop. See Section 6.3.3.

LN******* NAMELIST ERROR

Indicates that the form of a NAMELIST statement is in error or the usage of a NAMELIST name is incorrect. See Section 9.9.

LP******* EXPONENTIATION TO A LOGICAL POWER

See Section 3.1.2.

LS******* SYNTAX ERROR IN INPUT/OUTPUT LIST

Indicates an error in the form of an input/output list. See Section 9.1.

LU******* NON-LOGICAL EXPRESSION USED IN A LOGICAL CONTEXT

MA******* DUMMY PARAMETER MAY NOT APPEAR IN EQUIVALENCE STATEMENT

Indicates that a formal argument of the subroutine or function being compiled has been used in an EQUIVALENCE statement. See Section 5.4 and 7.2.1.

MO******* MEMORY OVERFLOW, FIELD LENGTH TOO SHORT

Indicates that the memory field, as specified on the Job identification statement or by an RFL statement is too short for compilation.

MS******* UNDEFINED STATEMENT NUMBER

Indicates that references have been made to statement numbers which did not appear anywhere in the statement label field of a line. May appear when the terminal statement of a DO loop is referenced from outside the loop. The missing statement numbers are listed above this diagnostic.

NC******* SUBROUTINE OR FUNCTION NAME CONFLICTS WITH A PRIOR USAGE

This diagnostic is usually the result of a subscripted variable not being dimensioned.

NL******* NAMELIST NAME NOT UNIQUE

NM******* IMPROPER HEADER CARD

Indicates that a PRØGRAM, SUBRØUTINE, or FUNCTIØN card is in error. This diagnostic may also be caused by a missing END statement in a previous subprogram.

NO******* NO OBJECT CODE GENERATED

The source program has generated no object code. This error will occur if a vacuous record is input to the compiler as when one too many 7-8-9 cards appear after the control cards or an EOF (7-8-9/17) is used in place of an EOR (7-8-9).

NP******* NO PATH TO THIS STATEMENT

The logic of the program is arranged so that this statment can never be executed.

NV******* VARIABLY DIMENSIONED ARRAY IN NAMELIST

A variably dimensioned array has been used in a NAMELIST statement, which is illegal.

OD******* REFERENCE TO AN ARRAY BEFORE IT IS DIMENSIONED

Indicates that a reference to an array was made prior to the appearance of the array in a DIMENSIØN statement. See Section 5.2.

PM******* FUNCTION PARAMETER MODE INCONSISTENCY

Indicates that the parameters in an arithmetic statement function reference do not agree in mode with the formal parameters of the statement function. See Section 7.8.

PN******* UNBALANCED PARENTHESIS

Indicates an unequal number of left and right parentheses in a statement. May be caused by punching past column 72.

PT******* SYNTAX ERROR IN AN ENTRY STATEMENT

The ENTRY statement being processed is labeled, has more than one name, is in a DØ loop, has a name starting with a number, or has a formal argument list. See Section 7.12.

RN******* SYNTAX ERROR IN A RETURN STATEMENT

Indicates an error in the form of a RETURN statement. See Section 6.7.

RW******* ILLEGAL USE OF RESERVED NAME

See Section 15.6.1

SB******* ERROR IN AN ARRAY SUBSCRIPT

Indicates an error in the form of a subscript of an array reference currently being processed. Can also be caused by using more subscripts in the array reference than are indicated in the dimensioning information for the array. See Sections 2.5 and 2.6.

SE******* SYNTAX ERROR IN SENSE STATEMENT

Statements which generate this diagnostic are using an obsolete statement form.

SF******* FIELD LENGTH OF ROUTINE BEING COMPILED EXCEEDS THE SPECIFIED FIELD LENGTH

A program takes more memory than is specified in the field length parameter of the Job identification statement. See CALIDOSCOPE Control Statements.

SM******* SYNTAX ERROR IN STATEMENT NUMBER

Indicates an error in the form of the statement number field. See Section 2.2.2.

SN******* ILLEGAL CHARACTER IN STATEMENT NUMBER USAGE

In a Fortran statement which may contain statement numbers, a statement number is incorrectly stated or positioned in the statement.

SY******* SYSTEM ERROR IN FORTRAN COMPILER

Show these to the programming consultant!

TM******* SUBROUTINE HAS MORE THAN 60 ARGUMENTS

Indicates that a subroutine reference has more than 60 arguments or that the subprogram being compiled has more than 60 parameters.

TN******* PROGRAM HAS MORE THAN 50 ARGUMENTS

Indicates that the PRØGRAM card has more than 50 arguments.

TT******* VARIABLE GIVEN CONFLICTING TYPES

A variable has appeared in more than one type statement. See Section 5.1.

TY******* SYNTAX ERROR IN A TYPE STATEMENT

Indicates an error in the form of a type statement. See Section 5.1.

UA******* REFERENCE MADE TO AN AS YET UNDIMENSIONED ARRAY

Indicates reference was made to an array which has not previously appeared in a DIMENSIØN statement.

UE******* LOGICAL UNIT NUMBER IS NOT AN INTEGER

The logical unit number in an I/O statement must be a constant or simple variable of integer type. See the introduction to Chapter 10.

VC******* VARIABLE NAME CONFLICTS WITH A PRIOR USAGE

Indicates that a variable name appears which conflicts with some prior usage, where, for example, the prior usage was a mispunched subscripted variable mistaken as a function.

VD******* ARRAY WHOSE DIMENSIONS ARE ARGUMENTS TO THE SUBROUTINE OR FUNCTION HAS BEEN MISUSED

Indicates improper use of an array with variable dimensions. See Section 7.11.

XF******* SYNTAX ERROR IN THE EXPRESSION BEING PROCESSED

Indicates an error in the form of the expression currently being processed. An error in the form of a complex constant is also indicated by this diagnostic. See Chapter 3.

ZY******* SYSTEM ERROR-UNKNOWN TWO LETTER CODE

See the Computer Center programming consultant.

This list of diagnostic messages may be changed in future versions of the CAL RUN Fortran Compiler. The Computer Center programming consultant should be seen if the cause of a diagnostic cannot be found.

15.4.1.2 Errors Undetected by Compiler

Not all of the errors that may be in a source language program are diagnosed by the RUN compiler. These errors may result in execution time error messages (see 15.4.2) or in incorrect answers to the problem being programmed.

The compiler can only check the correctness of the syntax of a statement or group of statements, not their intent or meaning. (As a parallel, an English language sentence may be structurally correct but convey no meaning.) Thus a programmer must carefully examine his program to be sure that it conveys the meaning he intended, i.e., solves the given problem.

One way to check for meaning is to follow the program through with a sample set of data, performing by hand the operations prescribed by each program statement. This procedure is commonly called "dry running" the program.

The CAL RUN Fortran compiler itself does not completely check the program for correct syntax. One particular area where few checks are made is program flow, i.e., can the program logically proceed at each branch statement (Chapter 6) to the sections of the program indicated on the statements? Thus no diagnostics are given if there are transfers to statements within a DØ loop from outside the loop. Each branch statement should be checked by the programmer to be sure that all indicated transfers in control are valid. In addition, it may not detect instances where a subroutine (or function) name is used incorrectly in the subroutine (or function) definition.

The compiler does not check the use of subscripted variables very carefully. No check is made either at compilation time or at execution time to insure that the referenced element of an array is within the bounds of the array as allocated by the DIMENSIØN statement. Exceeding the bounds of an array is likely to destroy the contents of locations stored after it and cause other errors, such as ARITH errors (see Section 15.4.4). Also the compiler does not give any diagnostics if the number of subscripts used with an array name is less than the number indicated by the DIMENSIØN statement.

One inherent feature of the language that may cause undiagnosed errors is the ability to have a mixed-mode arithmetic expression (Section 3.1.2). Thus, for example, forgetting to type a variable as CØMPLEX could result in an erroneous calculation of a complex expression. Further difficulties may arise when real and integer variables and constants are mixed, because integer arithmetic produces truncated results. A careful examination of the program should be made to verify that an arithmetic expression is evaluated as intended and that the types of all variables are those intended.

IMPORTANT: "Dry running" a program not only aids finding undetected errors, but facilitates the debugging phase of a program. Flow charting a program before coding aids spotting bug-prone logic.

15.4.2 RUN Fortran Execution Time Error Mesages

The Fortran library subprograms test for many of the more common cases of incorrect arguments and indicate these errors to the programmer by a standard error printout. The form of the listing on the fileset ØUTPUT is:

Message indicating the nature of the error

ERROR NUMBER n DETECTED BY LSUB AT y
CALLED FROM SUBn AT Yn=m_n IN SUBn
CALLED FROM SUBn-1 AT Yn-1=m_{n-1} IN SUB$_{n-1}$
Trace information
CALLED FROM SUB1 AT Y1 = m_1 IN SUB1
CALLED FROM main AT Ym = m_0 IN MAIN

where n is the error number associated with the detected error; LSUB is the library subprogram in which the error was detected at address y. SUB1,...,SUBn are subprogram names and main is the name of the main program. The Y1,...,Yn,Ym are absolute (octal) addresses in the program, and the m_i are the corresponding relative addresses within the subprogram.
On those errors which are fatal, execution is halted and an error skip of control statements is initiated (see CALIDOSCOPE Control Statements). Also the message,

FATAL ERROR N

where N is the error number, is placed in the Job Log.

The trace information shows (in reverse order) the way in which program control was transferred from the main program to subprogram SUB1 to subprogram SUB2 to ... to subprogram SUBn which referenced LSUB. $m_0,m_1,...,m_n$ indicate the place in each subprogram where the subprogram above it was CALLed or referenced. This information is useful to the programmer in establishing how the error occurred.

Example: The following error printout occurred in the sample output shown in Chapter 14 as compiled by the RUN Fortran compiler.

```
NEGATIVE ARGUMENT
ERROR NUMBER     39 DETECTED BY SQRT AT ADDRESS 007323
CALLED FROM WORK    AT 003251=000010 IN WORK
CALLED FROM OVERFL  AT 000706=000050 IN OVERFL
```

Here the main program OVERFL called subroutine WORK which used the SQRT function with a negative argument. If WORK has been called from several different places in OVERFL the address given with OVERFL would locate the CALL used when the error occurred. The addresses given in the error printout are first the absolute then the relative.

When a program terminates normally or with a library detected error (not on an ARITHMETIC ERROR, LIMIT EXCEEDED, etc., exit), an error summary is provided, and each error number is given with the number of times it occurred in the job. Those error numbers which did not occur in the job are not listed.

FORTRAN ERROR MESSAGES AND ASSOCIATED NUMBERS

Listed below are the error conditions checked for by various Fortran library subprograms, the standard recovery action (either the value returned, or the word "fatal" for fatal errors), and the associated error number.

The symbols INF and IND denote the infinite and indefinite forms of a floating-point number described in Appendix A.

When an error condition is preceded by "also" it indicates that the subprogram in question calls on a subordinate library subprogram, giving it the arguments indicated. Therefore, the subordinate subprogram may detect some errors of its own and report them under its own error number.

Routine	Condition	Standard Recovery	Error Number
ACGOER	This routine is only called upon detection of a computed or assigned GO TO error.	Fatal	1
ACOS(R)	R = INF or R = IND or abs (R) .GT. 1.0	+IND +IND	2
ALOG(R)	R = INF or R = IND or R .LT. 0 R = 0	+IND -INF	3
ALOG10(R)	R = INF or R = IND or R.LT.0 R = 0	+IND -INF	4
ASIN(R)	R = INF or R = IND or abs (R) .GT. 1.0	+IND	5
ATAN(R)	R = INF or R = IND	+IND	6
ATAN2 (R1, R2)	(R1 or R2) = (INF or IND) R1 = R2 = 0	+IND +IND	7
CABS(Z)	(real (Z) or imag (Z)) = (INF or IND)	+IND	8
CBAIEX:Z**I	(real (Z) or imag (Z)) = (INF or IND) Z= (0,0) and I.LE.0	(+IND,+IND) (+IND, +IND)	9

Routine	Condition	Standard Recovery	Error Number
CCOS(Z)	(real (Z) or imag (Z)) = (INF or IND) also: COS(real (Z)) and EXP (imag (Z)) and imag (Z) .LT. -675.82	(+IND,+IND)	10
CEXP(Z)	real (Z) or imag (Z)) = (INF or IND); also: SIN (imag (Z)) and EXP (real (Z))	(+IND,+IND)	11
CLOG(Z)	(real (Z) or imag (Z)) = (INF or IND); also: ALOG (CABS(Z)) and ATAN2 (imag (Z), real (Z))	(+IND,+IND)	12
COS(R)	R = INF or IND or abs (R) .GT. 2.2E14	+IND	13
CSIN(Z)	(real (Z) or imag (Z)) = (INF or IND) ALSO: SIN (real(Z)) and FXP (imag (Z)) and imag (Z) .LT. -675.82	(+IND,+IND)	14
CSQRT(Z)	(real (Z) or imag (Z)) = (INF or IND)	(+IND,+IND)	15
DABS (D)	D = INF D = IND	+INF +IND	16
DATAN (D)	D = INF or D = IND	+IND	17
DATAN2(D1, D2)	(D1 or D2) = (INF or IND) D1=D2=0	+IND +IND	18
DBADEX: D1**D2	(D1 or D2) = (INF or IND) D1=0 and D2 .LE. 0 D1 .LT. 0	+IND +IND +IND	19
DBAIEX: D1**I2	D1 = INF or D1 = IND D1 = 0 and I2.LE.0	+IND +IND	20
DBAREX: D1**R2	(D1 or R2) = (INF or IND) D1 = 0 and R2 .LE. 0 D1 .LT. 0	+IND +IND	21
DCOS(D)	D = INF or D= IND or abs (D) .GT. 2.2E14	+IND	22
DEXP(D)	D = INF or D = IND D.GT. 741.67	+IND +INF	23

Routine	Condition	Standard Recovery	Error Number
DLOG(D)	D = INF or D = IND or D .LT. 0 D = 0	+IND -INF	24
DLOG10(D)	D = INF or D = IND or D .LT. 0 D = 0	+IND -INF	25
DMOD(D1,D2)	(D1 or D2) = (INF or IND) D2 = 0 D1/D2 .GE. 2**96	+IND +IND +IND	26
DSIGN(D1,D2)	D1 = IND or D2 = (0 or INF or IND) D1 = INF	+IND INF with sign of D2	27
DSIN (D)	D = INF or D = IND or abs (D) .GT. 2.2E14	+IND	28
DSQRT (D)	D = INF or D = IND or D .LT. 0	+IND	29
EXP (R)	R = INF or R = IND R .GT. 741.67	+IND +INF	30
IBAIEX: I1**I2	I1 = 0 and I2 .LE. 0 I1**I2 .GE. 2**48	0 0	31
IDINT(D)	D = +INF or D = IND or D .GE. 2**59 D = -INF or D .LE. -2**59	2**59-1 1-2**59	32
RBADEX: R1**D2	(R1 or D2) = (INF or IND) R1 = 0 and D2 .LE. 0 R1 .LT. 0	+IND +IND 	33
RBAIEX: R1**I2	R1 = INF or R1 = IND R1 = 0 and I2 .LE. 0 R1**I2 = INF	+IND +IND 	34
RBAREX: R1**R2	(R1 or R2) = (INF or IND) R1 = 0 and R2 .LE. 0 R1 .LT. 0	+IND +IND +IND	35
SIN(R)	R = INF or R = IND or abs (R) .GT. 2.2E14	+IND	36
SLITE(I)	I .GT. 6 or I .LT. 0	PROCEED	37
SLITET (I1,I2)	I1 .GT. 6 or I1 .LE. 0	I2 = 2	38

Routine	Condition	Standard Recovery	Error Number
SQRT(P)	R = INF or R = IND or R .LT. 0	+IND	39
SSWTCH(I1, I2)	I1 .GT. 6 or I1 .LE. 0	I2 = 2	40
TAN(R)	R = INF or R = IND or abs (R) .GT. 8.4E14	+IND	41
TANH(R)	R = INF or R = IND	+IND	42
Unused			43

The following routines whose names are followed by a colon and a sample statement or expression are not explicitly called for by the programmer in his program. They are called for when the indicated statement is used in the program. Currently, all errors detected by these subprograms, except as noted, are fatal errors causing termination of the job.

Routine	Condition	Error
IOCHEK IF(UNIT, i)	IF(UNIT, i) should only be used on a BUFFER I/O file. The file name given in the message is = i.	44 NF

LENGTH:
LENGTH (i)
IFENDF:
IF(EOR,I)
IOCHEC:
IF(IOCHECK,I)
	Status of a BUFFER I/O file must be checked by IF(UNIT,I) in the instance where an error would occur if no check were made.	45 NF

Unused		46
SETPRU	The physical record unit size should not be changed since the current process is not completed. However the PRU size is changed as requested.	47 NF
SETPRU	The physical record unit size cannot be changed on this type of device. Returns without changing the PRU size.	48 NF
INPUTN: READ(I,X)	NAMELIST error: precision lost in converting integer constant. NAMELIST data terminated by 7-8-9 or 6-7-8-9 card, not $. Too few constants for unsubscripted array.	49 NF

Routine	Condition	Error
OVERLAY	Improper use of OVFPLAY	50 F
SYSTEMP	User program detected fatal error.	51 F
SYSTEMP	User program detected non-fatal error. Execution is allowed to continue but results should be suspect.	52 NF
CKFLAG.	Parity error on previous read not checked for. IF(IOCHECK,I)N1,N2 should be used to check for parity errors (except on buffer I/O files). N1 is the return indicating a parity error, N2 is the return indicating no error.	53 F
CKFLAG.	Number of elements in the list for the previous unformatted (binary) read was more than were contained in the binary record and the LENGTH function was not used to check for this condition.	54 F
CKFLAG.	An end-of-file was encountered on the last read on this file and neither IF(EOF,I) nor IF(UNIT,I) were used to check for this condition.	55 F
CKFLAG.	A read was attempted after a write, without an intervening BACKSPACE or REWIND. This is not allowed.	56 F
BUFFEI: BUFFER IN	Starting address greater than terminal address.	57 F
CKFLAG.	An I/O operation was attempted on a buffered I/O file without checking its status first with an IF(UNIT,I).	58 F
BUFFEO: BUFFER OUT	Starting address greater than terminal address minus 1.	59 F
INPUTN: READ(I,X)	NAMELIST name not found. Wrong type constant. Incorrect subscript. Too many constants. (, $, or = expected.	60 F
SETPRU	The PRU size requested exceeds the maximum allowed for magnetic tape.	61 F
MEMORY	Illegal request for memory.	62 F
SIO$	Physical record too big, i.e., longer than 512 words if binary or 1280 characters if coded (unless SETPRU has been called).	63 F

Routine	Condition	Error
GETBA	Undefined file. There is no buffer defined for the file being accessed; i.e., the fileset name appearing in the diagnostic was not specified in the PROGRAM statement list. See Section 7.4. If the file is OUTPUT, the message NO OUTPUT FILE FOUND appears in the job log. May also indicate that values have been improperly assigned to a variable tape number, or that lower core has been destroyed by the program.	64 F
Unused		65
INPUTS: DECODE	Attempt was made to transfer more than 150 characters per record on DECODE processing.	66 F
Unused		67
KRAKER/KODER: Used in BCD	Illegal letter used as format specification.	68 F
I/O CONVERSION*	Improper parenthesis nesting in format specification.	69 F
	Format exceeds maximum record length currently set on input or output.	70 F
	Field width specified as zero.	71 F
	Field width specified as less than or equal to the specified fraction width	72 F
	List was used with an I/O statement with only Hollerith specifications in the format.	73 F
Unused		74
Unused		75
Unused		76
FBPUT/FBGET	Wrong number of arguments.	77 F
KRAKER: USED in BCD input Conversion*	Illegal character in data item in record being read.	78 F
	Data converted is too large.	79 F
INPUTB: Blocking BACKSP	An attempt has been made to read an unblocked fileset as if blocked or blocking control word mismatch, or incorrect trailing blocking control word.	80 F

* See footnote on next page.

Routine	Condition	Error
IF(IEOI)	End of Information on previous read.	81 F
INPUTB	Machine, system, disk or tape checksum error. It is important to show the job output to the Computer Center Consultant in order to enable the Systems Staff to pin down the machine failure. The parity error flag is set and the blocking control word is checked. If it is all right, the program continues.	82 NF
	The function code is not allowed for the device; the file is closed or not allowed to be written on or read follows a write.	83 F
	An unexpected error status has been detected in the FET. Possible system error. See the Computer Center Consultant.	84 F
OUTPTS: ENCODF	Attempt to transfer more than 150 characters per record on ENCODE processing.	85 F
FBPUT	LRECL greater than blocksize.	86 F
KODER: Used in BCD output Conversion*	Attempt to output a single precision variable under "D" format.	87 F
SYS.UEX	Attempt to reference a missing subprogram.	93 F

* The message indicating the nature of an error found by the subroutines KODER (formatted output) and KRAKER (input) is followed by supplemental messages which localize the error. The FORMAT statement number and up to one line of the FORMAT statement will be printed with an up arrow to indicate the position in the FORMAT statement where the error was detected. In the case of input data errors, the position of the card in the data deck and one line of the record (card) in which the error was detected will be printed with an up arrow pointing to the spot at which the error was found.

15.4.3 Operating System Error Messages

The operating system produces messages in the Job Log to warn of a possible error, to indicate a definite error, and/or to report hardware malfunctions. The exact messages depend on the version of the operating system. Here, the general form of CALIDOSCOPE error messages is explained and a few of the more common examples are discussed briefly. Section 15.4.4 discusses arithmetic errors, which are very common errors that also generate messages in the Job Log. An example of an arithmetic error message occurs in the sample program in 14.2 and is discussed in 15.5.

All system error messages are printed on the Job Log and are usually (the exceptions are discussed below) prefixed with the time and a character pair, the first of which indicates the severity of the error.

- I: indicates an informational message. Sometimes such messages are associated with a fatal or warning error and give additional information; other times they pertain to an error which was automatically recovered by the system.
- F: indicates a fatal error that causes the job to be aborted.
- W: is a warning message which indicates that a possible error was detected, but processing continues. The cause of the message should be investigated.

Approximate text of some common messages

These three messages indicate that the job tried to exceed the indicated limit as specified on the job identification statement (job card). It is well to ascertain that the indicated limit is not being exceeded because of an error in the program before resubmitting the job with a higher limit. aaaaaa indicates an octal address.

```
hh.mm.ss F:CENTRAL PROCESSOR TIME LIMIT EXCEEDED AT ADDRESS aaaaaa

hh.mm.ss F:PRINT LIMIT EXCEEDED
hh.mm.ss I:FET ADDRESS = aaaaaa, FILESET = name

hh.mm.ss F:PUNCH LIMIT EXCEEDED
hh.mm.ss I:  FET ADDRESS = aaaaaa, FILESET = name
```

The following messages are printed by HYDRA, the program which supervises reading jobs into the computer. They are the most common messages which don't have a prefix containing the time, etc. These errors prevent the job from being run:

BAD JOB NUMBER

indicates that the job number specified is out of funds or was never funded at all; or the two check characters necessary to use the indicated number were not specified correctly.

BAD JOB CARD

covers a variety of sins: an illegal character occurs in the statement, such as an 8 or 9 in an octal number field; a parameter has been specified in excess of the maximum allowed for that parameters on this job number; the job is an S job but specifies a parameter which is too large for S.

CHECKSUM ERROR RECORD n CARD m

indicates that binary card m in record n* has been punched improperly or read improperly.

SERIAL CHK RECORD n CARD m

indicates that binary card m in record n* is out of order.

The following possible errors are reported but the job is allowed to run;

HOLLERITH CHECK RECORD n CARD m

indicates that card m in record n* has an illegal combination of punches for a Hollerith card. The job is allowed to execute with blank(s) substituted for the character(s) in question.

The message

MODE CHANGE RECORD n CARD m

means that both Hollerith and binary cards occurred in the same record.

---

* n and m are decimal. The control statement record is counted as record 0. Cards within a record are numbered starting with 1. If you cannot analyze these problems, see the programming consultant. Checksum errors which disappear when a job is simply resubmitted should be reported to the consultant in any case.

15.4.4 Arithmetic Errors

Computation is halted (error exit) when the internal circuitry of the 6400 detects certain "arithmetic" errors. A dump is then initiated and a message of the form:

F:ARITHMETIC ERROR MODE X AT ADDRESS Y

is placed in the Job Log. X is the type (or mode) of error which occurred and Y is the absolute address (usually plus one) (in octal) of the instruction word causing the error. Several lines of I: messagesmay follow the F: message giving some explanatory information and the locations of some values which might have caused the error.

A dump of the 6400's registers is automatic (i.e., no DMP statement is needed) with an arithmetic error. The system communication area, showing the names of the filesets and the locations of the FETs for the filesets, is given, followed by about 100B locations on each side of the offending instruction. An error skip of control statements (see CALIDOSCOPE Control Statements) is initiated after an arithmetic error.

The sample job output in Section 14.2 shows an example of an arithmetic error. Section 15.5 discusses the error from the sample job.

The types of errors detected are:

MODE 0 — An ARITHMETIC ERROR with a MODE of zero may be generated by an erroneous transfer to location zero, e.g., the 6400's internal circuitry has found no error. This wild transfer can be caused by the program storing data where instructions should be (e.g., subscripts of a variable are incorrect or labelled COMMON exceeded) or by having the actual parameters to a subprogram not agree with their intended use in the subprogram. COMPASS programmers note that a MODE 0 ARITHMETIC error can also be generated by an attempt to execute an illegal instruction or having a PS instruction in the lower 15 bits of a word.

MODE 1 — Address out of bounds. The instruction has tried to reference a location not contained within a job's field length. In the dump one of the address registers will contain an address greater than the program length, which is shown in the FL register. This error can occur when an operand is being fetched for a calculation or stored as a result of a calculation. In a Fortran program, subscripts of an array which exceed the dimensions of the array can cause this error.

In this case the address given in the Job Log message is around 400000B and a reference appears under "UNSATISFIED EXTERNALS" at the end of the load map.

MODE 2 — Operand out of range. The instruction has tried to use an infinite operand in a floating-point operation. In the dump, at least one of the working registers will contain an infinite form (37770...0 or 4000.....0). An infinite operand is usually created by a division by zero in a Fortran program. However only its subsequent use in another calculation causes this error. Refer to Appendix A for a further explanation and for the form of the infinite operand.

MODE 4 — Indefinite operand. The instruction has tried to use an indefinite operand in a floating-point operation. In the dump, at least one of the working registers will contain an indefinite form (17770...0 or 60000....0). Because the loader sets unused cells in memory to indefinite values (see p. 14.12), the most common source of indefinite operands is referencing variables which have not been set. An array reference using a wild index is a frequent cause of using such an 'undefined' variable. An indefinite operand is generated either when an error is detected in one of the library mathematical functions (15.4.2) or when zero is divided by zero. However, only the subsequent use in another calculation causes this error. Refer to Appendix A for a further explanation and the form of the indefinite operand.

MODE 6 — The instruction has tried to use both infinite and indefinite operands in a floating-point operation. See Modes 2 and 4 above.

15.5 Debugging and Memory Dump Interpretation

Debugging is a term used to describe the process of trying to figure out why a program did not do what it was supposed to do. It sometimes happens that an error cannot be found simply by inspecting the program or its normal output and in such cases one may be forced to seek clues in a memory dump.

Dumps of memory (i.e., a printout of the contents of locations in memory are taken when a DMP statement (see CALIDOSCOPE Control Statements) is encountered in the control statement record or when a fatal error is detected by the operating system. Section 15.5.1 describes the information given in a dump, using the dump produced by the sample program of Chapter 14 for examples.

Two problems are commonly associated with debugging and memory dumps. One is to identify the variable or code associated with an absolute address given in a dump or error message. The other is the inverse problem of finding the absolute address of some variable or code so that it can be inspected in a dump. These are discussed in sections 15.5.2 and 15.5.3.

15.5.1 Dump Format

A dump shows the contents of registers and memory words in octal. Each octal digit represents 3 bits of the binary number contained in a word. The correspondence between the octal digit and binary number is as follows:

Octal	Binary	Octal	Binary
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

The first line of the dump gives the time and circumstances of the dump. The second line gives the values of the location count (P = 000000), the reference address* (RA = 101100), the field length (FL = 010700), the exit mode (EM = 07), the reference address for ECS (ECS RA = 01053000), the field length for ECS (ECS FL = 00000000), and the monitor address* (MA = 003567).

The next 8 lines give the contents of the machine registers and related information in one of the forms:

X = Y or C(X) = Y ,

where X is the name of a register (X0-X7, A0-A7, or B0-B7) and Y is either the contents of that register in the first form or the contents of the cell specified by the address contained in the register in the second form. For example, if A1 = 005517 appears followed by C(A1) = 00000 00000 00000 02053, then the contents of address 5517 is 2053B. C(A1) = ///// ///// ///// ///// would indicate that the address in A1 points outside the job's memory field.

The next 2 lines give the contents of the system communication registers.

Following these registers, the contents of memory locations are given. The dump of memory is in the form of five columns. The first column comprises 6 octal digits which specify the absolute location (in the user's portion of memory) of the word whose contents are shown next. The contents of 4 consecutive memory locations then follow in columns 2-5. Each location is represented by 20 octal digits broken up by an intervening space into four groups of five digits each. For example, the sample dump shows that location 64 contains 14071700000000000000B, location 65 contains 10647B, etc. When consecutive locations contain the same value, a line of the following form is printed:

---

* The reference address is the hardware location of the beginning of the memory field. It is not relevant to the user's program but may be significant in checking for possible machine failure.

The monitor address is a pointer internal to the generating system and is also not relevant to the user's program.

WORDS N1 TO N2 ALL CONTAIN N3

Thus, the line before the one giving the contents of location 64 in the sample dump tells one that locations 10 through 63, inclusive, are all zero.

The first few words in the system communication area may be useful to the programmer. Starting in location 2 are the fileset names for the filesets in the order that they appear in the PROGRAM statement. The high order 42 bits (first 14 octal digits of each word) contain the name of the fileset in 6400 internal BCD code; the low order 18 bits (last 6 octal digits) contain the address of the Fileset Environment Table (FET) used for that fileset. Thus in the sample output these cells are shown following the label 'SYSTEM COMMUNICATION AREA'. Cell 2 contains the name INPUT (1116202524B) and the Fileset Environment Table (FET) is located at 1031B; cell 3 contains OUTPUT (172524202524B), and its FET is at 2053B; cells 4 contains TAPE5 (2401200540B), and since it is equivalenced to INPUT, its FET is also 1031B.

The contents of other locations in a dump can be interpreted by knowing the octal forms of each type of variable or constant used in the Fortran language. These octal forms as well as methods or tables for their conversion to decimal or alphanumeric forms are given in Appendix A.

15.5.2 Identifying Code or Variables Starting from Absolute Addresses

The addresses given in memory dumps and most addresses given in diagnostic messages are absolute. That is, the user program's memory is considered as one long block of cells numbered from 0 to the field length less one and the absolute address merely identifies one of these cells. On the other hand, the addresses given on compiler listings are relative to the beginning of the program being compiled and are referred to here as relative locations. To convert an absolute address AA to a relative location RL involves two steps:

1. Identify the subprogram containing the absolute address and determine its load address LA. To do this compare AA with the loading addresses of the subprograms indicated on the load map. The largest load address which does not exceed AA is LA and the corresponding subprogram contains AA.

2. RL = AA - LA. Remember that the numbers are octal. RL may now be used to identify a variable or code in the compiler listing in the appropriate subprogram.

Example:

In the sample program shown in Chapter 14, the following arithmetic error occurred:

```
F:ARITHMETIC ERROR MODE 2 AT ADDRESS 003250
I:  INFINITE OPERAND USED
I:  OPERAND MAY HAVE RESULTED FROM A DIVISION BY ZERO
I:  REFERENCE TO WORD 001026 WHOSE VALUE IS INFINITE
```

We will now identify the statement associated with address 3250. Because the address given in the error message is one greater than the address of the offending instruction, we take 3247 as AA. From the load map, we identify WORK, whose LA is 3241, as the subprogram containing the instruction. Now RL = 3247 - 3241 = 6 and inspection of the compiler listing for WORK identifies the statement associated with the arithmetic error:

```
X2 = 2.*X1
```

which indicates fairly strongly that X1 has somehow become out-of-range. It should be noted that because there can be up to 4 instructions in one computer word, it sometimes happens that instructions from two adjacent statements 'share' the same word, in which case there is an ambiguity as to the identification of the statement.

Example:

Now we identify the variable associated with address 001026 given in the error message above. With AA = 1026 we inspect the load map and find that OVERFL, whose LA is 636, contains the cell in question. So RL = 1026 - 636 = 170 and inspection of the compiler listing for OVERFL identifies the variable B as occupying 170.

15.5.3 Finding Absolute Addresses Starting with Addresses Given in Compiler Listings

The addresses given on compiler listings are relative to the beginning of the individual program being compiled. Since the loader relocates each subprogram so that its origin is at the load address rather than at 0, the addresses of variables, etc., given in the listing must be modified in order to inspect them in memory. The addresses where things actually get put in memory are referred to as absolute addresses. An absolute address AA is easily calculated from a relative location RL given in the compiler listings:

1. Look up in the load map the load address LA where the loader put the given subprogram.

2. AA = RL + LA. Remember that the numbers are octal. AA may now be used to inspect the variable (or whatever) in a memory dump.

Example:

We find the variable DEMO in the subroutine WORK in the sample program of Chapter 14, starting from its RL of 23. The load map gives 3241 as the LA of WORK, so the absolute address of DEMO is 23 + 3241 = 3264. Cell 3264 is shown in the sample dump as 60007 77777 02004 03264. This is an example of the values which the loader puts in unused cells in memory, so we deduce that DEMO had not been set at the time the dump was taken.

Labeled COMMON blocks are not subprograms, but the method for finding variables contained in labeled COMMON is the same as that for finding other variables except that LA must be taken as the location of the block.

15.6 Hazardous Names

There are several aspects of the RUN Fortran implementation which restrict the kinds of name that may be given to subprograms (subroutines, functions, main programs, etc.). This section discusses the causes and effects of such restrictions and gives as much explicit information as possible as to what the reserved names are. The names of less than 7 characters listed in this section are all perfectly legal ANSI Fortran names and indicate substandard implementation of RUN.

The clumsy addition of literals to FORMATs by CDC causes the possibility of ambiguous statements; so the name FORMAT should not be used for arrays.

15.6.1 Subprogram Names Illegal Because of Implied Calls

Some Fortran statements other than CALL generate code which calls a subprogram. An example is the PRINT statement, which generates a call of the library subprogram OUTPTC. Consequently, if a programmer names a subroutine of his own OUTPTC, the loader has no way to link a given call to the correct subprogram. RUN detects a statement which calls a subprogram by one of these names and flags it as a fatal error. However, it does not detect a subprogram named one of these names. Consequently, a main program could be given one of these names, no error would be detected, and an infinite loop might result. A complete list of such names follows:

ACGOER	DBADEX	IBAIEX	INPUTS	OUTPTN	RBAIEX
BACKSP	DBAIEX	IFENDF	IOCHEC	OUTPTS	RBAREX
BUFFEI	DBAREX	INPUTB	IOCHEK	PAUSE	REWINM
BUFFEO	END	INPUTC	OUTPTB	Q8NTRY	STOP
CBAIEX	ENDFIL	INPUTN	OUTPTC	RBADEX	

15.6.2 Names Which Collide with Library Subprogram Linkage

A general problem arises during loading when the same entry point exists in two or more subprograms. When this happens, the loader will link calls to the first subprogram containing the name. This sort of confusion is usually under the control of the programmer, who can avoid the problem in a variety of ways. Unfortunately, some subprograms in the RUN library call other subprograms in the library using linkage names which are legal Fortran subprogram names. If the programmer defines a subprogram of his own with such a name, he may 'derail' the internal linkage of the library into his own routine. No error is detected by RUN or the loader and havoc is the normal result.

The following are ~~(otherwise legal) names of library subprograms called~~ internally in the library:

ABNORML	IOIO
ALOG	IOSAV
ATAN2	IOZW
CABS	IOZZ
COS	KODER
CPC	KRAKER
CPC02	
CPC03	PRINTRG
CPC04	RESTORE
CPC999	SAVEREG
DCQR	SIN
DEXP	SQRT
DLOG	SYSTEM
EXP	
GETBA	

Many of these subprograms are only required under special circumstances, but the ones underlined are loaded with every RUN Fortran program which does formatted input and output.

Note that some of the above are Fortran functions, e.g., SIN, COS, SQRT, etc. Thus, if a programmer supplies his own version of such a Fortran function, it will affect calculations other than the calculation of the function in question. For example, the SIN function is used by the library functions CSIN, CCOS, CEXP. So supplying a different version of SIN (perhaps faster but less accurate than the library version) may well have unexpected and undesired side-effects.

15.6.3 Subprogram Names Which Collide with Library Deck Names

The relocatable binary form of a subprogram has a name associated with it called a deck name. The loader insists that only one subprogram of a given name be present in any collection of subprograms that it puts into memory at one time. When it finds a subprogram with the same name as one already loaded, it effectively ignores the latest one and proceeds. Generally, the deck name is the same as the name by which a subprogram is called. (Subprograms compiled by RUN are given the name specified on the SUBROUTINE, FUNCTION, etc., statement as a deck name.)

Unfortunately, the RUN library harbors some subprograms whose deck names differ from the names by which they are called. An example is ASIN, which has a deck name of ASINCOS. Thus, if a programmer accidentally names a subroutine ASINCOS in a collection of programs which uses ASIN somewhere, trouble results at load time. The loader will not load ASINCOS from the library and ASIN will be an undefined external unless the programmer provides his own ASIN subprogram.

Chapter 11, which lists RUN library subroutines and functions, indicates deck names when they differ from the name used to call the subroutine.

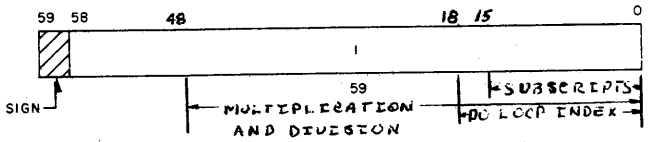
A.1 6400 Word Structure

Use of machine-dependent or compiler-dependent coding within a Fortran program requires familiarity with the 6400 word structures shown below.

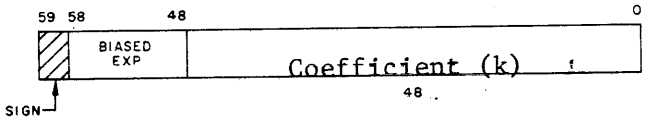
TABLE A Internal Data Formats

Floating-point Forms

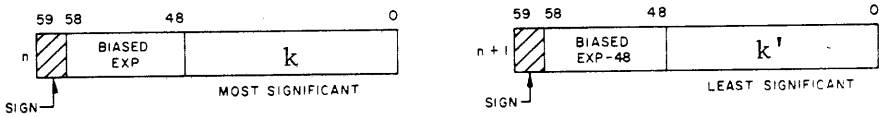
INTEGER



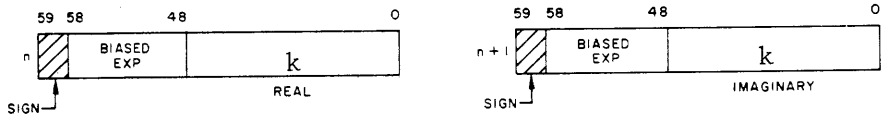
REAL



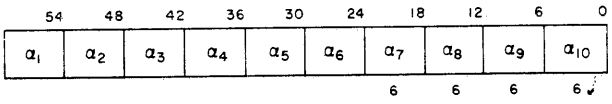
DOUBLE-PRECISION



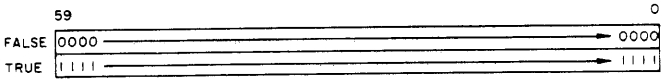
COMPLEX



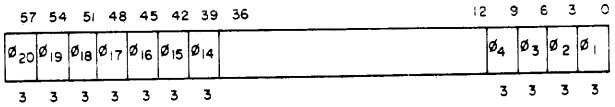
HOLLERITH (DISPLAY CODE)



LOGICAL



OCTAL



This knowledge is used in constructing octal masks, Ø or A formats; in manipulating alphanumeric information; and in performing special arithmetic calculations.

A.2 INTEGER ARITHMETIC

In fixed point addition and subtraction of 60 bit numbers, negative numbers are represented in one's complement notation* and overflows produce no error condition. The sign bit is in the high-order bit position (bit 59) and the binary point is at the right of the low-order bit position (bit 0). A zero sign bit indicates a positive number; a one, a negative number.

Examples (in octal):

0	0000000000000000000
-0	7777777777777777777
1	0000000000000000001
-1	7777777777777777776
25	0000000000000000031
-25	7777777777777777746

Further examples are shown in Table C.

During addition and subtraction the sign bit is treated as just another bit of the number. This results in what is termed 'wrap-around' instead of an overflow condition; thus if 1 is added to the largest positive number, the result is the largest negative number:

```
  3777777777777777777   (largest positive number)
+ 0000000000000000001
  4000000000000000000   (largest negative number)
```

- The 6400 does not have integer multiply and divide instructions: these operations are programmed using the floating-point instructions. As a result, operands for these operations may be no larger than $2^{48}-1$; larger operands lose their high-order bits. Programs using multiplication-division to shift bits or characters in a word are affected by this feature. Also, on output conversion, a variable greater than $2^{48}-1$ is flagged as out-of-range (R).

---

* The one's complement of a binary number is found by subtracting the number from 2^N-1, where N is 60 for the 6400.

A.3 REAL, COMPLEX, DOUBLE PRECISION ARITHMETIC

Floating-point word representation and arithmetic are used for the Fortran types real, double precision and complex.

Floating-point arithmetic takes advantage of the ability to express a number with the general expression kB^n, where:

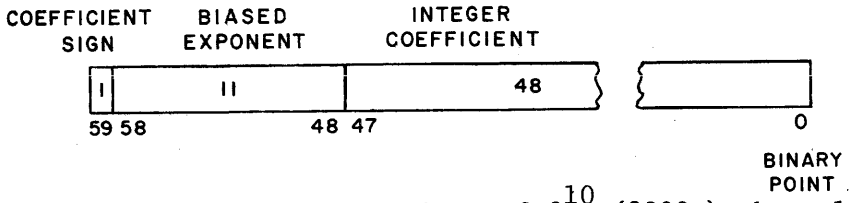
k = coefficient

B = base number

n = exponent, or power to which the base number is raised

For binary information the base number is the constant 2. The base does not appear in the number representation.

The 60-bit floating-point format used by the CDC 6400 is shown below. The binary point is considered to be to the right of the coefficient, thereby providing a 48-bit integer coefficient, the equivalent of about 15 decimal digits. The sign of the coefficient is carried in the highest order bit of the word; a zero bit indicates a positive number and a one bit, a negative number. Negative numbers are represented in one's complement notation.



The 11-bit exponent carries a bias of 2^{10} (2000_8) when placed in the floating-point word (a biased exponent is sometimes referred to as a characteristic). Thus, a number with a true exponent of 342_8 would appear as 2342_8; a number with a true exponent of -160_8 would appear as 1617_8. Exponent arithmetic is done in one's complement notation.

The coefficient is normalized; that is, the most significant bit is in bit 47 of the word. Since the binary point is to the right of the normalized coefficient, the coefficient is an integer between 2^{47} and 2^{48}. Therefore all numbers less than 2^{47} will have negative exponents.

Examples (in octal):

1.0	1720400000000000000
-1.0	6057377777777777777
1.0995×10^{12}	1770400000000000000
1.4074×10^{14}	2000400000000000000

Octal Floating-Point to Decimal Conversion Tables

Several tables are included in this appendix to assist the programmer who wishes an approximate conversion of octal floating-point values to decimal:

Table

- C illustrates some full word octal forms
- D gives conversions for values near unity
- E gives rough scale conversions for all floating-point numbers.

A decimal value which appears in the body of either Table D or Table E is equivalent to the octal number whose high-order digits appear as the row and column headings, respectively, with zeros to the right. For example:

Table	Row	Col.	Octal Number	Decimal Number
D	17154	0	17154000000000000000	.12500
D	17337	3	17337300000000000000	3776.0
E	17	504	17504000000000000000	1.677×10^{7}
E	05	704	05704000000000000000	2.4099×10^{-181}

Conversion of a Number from Decimal to Floating-Point Form

The operation of converting the decimal number 12.5 to its exact octal floating-point representation is as follows:

a. Convert the number to binary: $12.5_{10} = 1100.1_{2}$

b. Express the number as a 48-bit integer coefficient normalized (.i.e, with a one as the leftmost bit) times the appropriate integral power of two:

$$1100.1_{2} = 110010000000000000000000000000000000000000000000 \times 2^{-44}$$

(Octally seen as 6200000000000000 in the coefficient field.)

c. Convert this power to octal : $-44_{10} = -54_{8}$

d. If the power is positive, add it to 2000_8. If the power is negative, subtract it from 1777_8; therefore $1777_8 - 54_8 = 1723_8$.

e. And obtain the result as the octal word:

1723620000000000000.

Reversing the process gives the decimal version of the number.

Overflow, Underflow and Indeterminate Forms

The magnitude of a floating-point number is limited by the 11 bits allocated for the characteristic. Its range, from 0000_8 to 3777_8, is equivalent to about 10^{-294} to 10^{322}. The result of a floating-point operation (usually division by zero) which exceeds the upper limit (overflow case) is treated as an infinite quantity. The result is zero if the exponent falls below the lower limit (underflow case). Indefinite (i.e., indeterminate) results may follow the use of infinity (or possibly zero) as operands or may be returned by mathematical functions (refer to Section 15.4.4) which are given illegal arguments (such as SQRT (-1.)).

The floating-point circuitry of the computer assigns the following special bit configurations to indicate indefinite and infinite operands:

+ ∞	3777XXXXXXXXXXXXXXX	(causes MODE 2 errors)
- ∞	4000XXXXXXXXXXXXXXX	(causes MODE 2 errors)
+indefinite	1777XXXXXXXXXXXXXXX	(causes MODE 4 errors)
-indefinite	6000XXXXXXXXXXXXXXX	(causes MODE 4 errors)

(X = any octal digit)

Computation is not halted when the indefinite or infinite result is generated. If either form may be generated as the final value of the expression in an arithmetic replacement statement, the program may test for these exceptional conditions by using the LEGVAR function (see section 11.2.10) before the result is used for further computation.

For Underflow or Overflow Use:

LEGVAR(*vname*)	The function value returned will be -1, +1, or 0 if the variable *vname* is indefinite, infinite, or within range, respectively.

The infinite or indefinite result may be printed (printing as R for infinite, I for indefinite) or moved from one location to another with no error condition arising.

Computation is halted (error exit) when an attempt is made to use either form in a floating-point operation (see Section 15. 4.4). Use of an indefinite or infinite form which is the result of a sub-expression evaluation within an expression may trigger the error exit before the above test can be made. (Thus if A/B results in an infinite form, the evaluation of the expression (A/B)*C will cause an error exit.)

The operating system may be instructed to process infinite and indefinite operands without taking an error exit (see MODE in CALIDOSCOPE Control Statements).
Shown below are the results of all calculations involving infinite or indefinite forms.

Arithmetic Involving Infinite and Indefinite Forms

∞ - ∞ = I	∞ / N = ∞	∞ + ∞ = ∞	0 / ∞ = 0
∞ / ∞ = I	∞ + N = ∞	∞ * ∞ = ∞	0 * 0 = 0
∞ * 0 = I	∞ - N = ∞	∞ / 0 = ∞	0 / N = 0
0 / 0 = I	N / 0 = ∞	∞ * N = ∞	N / ∞ = 0
I θ X = I			0 * N = 0

where ∞ = infinity
I = indefinite
0 = zero
N = any finite nonzero floating-point number
X = I, N, 0 or ∞

A.4 ALPHANUMERIC WORDS

Table H lists the display codes stored for character string information on the 6400. Character strings are often compared and sorted by means of arithmetic or masking operations which depend on these octal values.

Several new methods of defining and handling character data are available:

1. Internal format conversions may be made by means of the FORTRAN statements ENCØDE and DECØDE (Section 10.5).
2. Two new forms, nRf and nLf, have been added as Hollerith constant specifications.
3. The Hollerith constants may be used in executable statements as well as in DATA statements, for example, A=9HHØLLERITH. (Note that the end-code for Hollerith constants in argument lists is a binary zero word.)

4. One new form, Rn, has been added as an alphanumeric format conversion specification.

Table B clarifies the use of these different forms.

Note that the octal form, which may be used to define any word structure, is useful for internal definition of the additional printer characters shown in Table G. (These printer characters do not appear on a key punch and must otherwise be multiple-punched on a card.)

A.5 LOGICAL OPERANDS

Logical operands are defined as 7777777777777777777 octal (minus 0) for true and binary zero (plus 0) for false. The logical operators .AND.,.ØR. and .NØT. are used with these operands as discussed in Section 3.3.

However, it must be remembered that operators .AND., .ØR. and .NOT. perform a dual function on the 6400, acting as masking operators as well as logical operators. The same internal evaluation is used for both types of operations: a bit-by-bit masking. That is, each binary bit of an operand is complemented in the .NOT. operation, and the pairs of corresponding bits in two operands are masked bit-by-bit in the .AND. and .ØR. operations. Section 3.4 defines these maskings evaluations. For example, given

$A = 0000000000000000001_8$ (a non-zero word)

$B = 0000000000000000010_8$ (a non-zero word)

then

$A.AND.B = 0000000000000000000_8$ (a plus zero word)

Yet, in a logical operation the word as a whole must be the logical unit of information. This unit is obtained by using as operands only the words in which all bits are the same, the plus zero (all zero bits) for false, or the minus zero (all one bits) for true. Logical operands which are defined within the FORTRAN language will be assigned these values.

If the definitions for logical operands are not followed, not only will logical operations be faulty but also the logical IF statement will give erroneous results. The test in a logical IF statement provides a true branch if the result of the expression is anything other than plus zero; that is, if any bit is one. Thus, in the example shown above, the expression A.AND.B gives a false value (a plus zero) and a false branch even though A and B have non-zero values (true values).

TABLE B Character String Data

Type of Definition	Form when less than full word is specified: Characters are justified	Form when less than full word is specified: Unused positions filled with	Example Definition	Example of information stored in machine word: As Characters (ten/word)[3]	Example of information stored in machine word: As octal digits (20/word)	Reference Section
Hollerith constant nHf [1]	left	blank characters	7HTABLEbZ	TABLEbZbbb	24010214055532555555	2.3.6
Hollerith constant nRf	right	binary zeros	7RTABLEbZ	000TABLEbZ	00000024010214055532	
Hollerith constant nLf [2]	left	binary zeros	7LTABLEbZ	TABLEbZ000	24010214055532000000	
Format conversion spec. Aw	left	blank characters	A4	RATSbbbbbb	22012423555555555555	9.3.12
Format conversion spec. Rw[2]	right	binary zeros	R4	000000RATS	0000000000022012423	9.3.15
Octal constant Øn	right	binary zeros	Ø53232515	000000$SUM	0000000000053232515	2.3.2
Octal format conversion Øw	right	binary zeros	Ø20	(P∧Q∨¬R)bb	51206721667622525555	9.3.12

1. All definitions must be for ten characters or less except the nHf Hollerith constant when it appears in a DATA statement. (Caution: the nHf is also used as an editing specification in a format statement. Such use does not define a Hollerith constant.)

2. Caution: The nLf form is a Hollerith constant, but the Lw form is a logical format conversion specification (see Section 9.3.17).

3. 0 represents a true binary zero, but not a display code zero. b represents a blank.

Moreover, unless a logical value (X) is properly defined, both "X" and ".NØT.X" will be treated as true in a logical IF statement. For example, if X = 1.0 = 1720400000000000000 octal, then .NØT. X = -1.0 (the complement of X) = 6057377777777777777 octal, and since neither of these is plus 0, they are both true.

The L input format conversion assigns the value .TRUE. to a variable when the field on the data card contains T as the first non-blank character. Otherwise the value .FALSE. is assigned. A diagnostic is never given, no matter what characters appear in the data field; that is, the input is alphanumeric. Note that each of the following data items will be assigned the value false if read in and converted under an L format:

F, 1, 0, -0, -1, .TRUE.

A.6 Summary of Numeric Representations

The smallest nonzero positive REAL which fits in one computer word is .3131513062514-293 or OCTAL 0001400000000000000

The largest positive REAL which fits in one computer word is .1265014083171+323 or OCTAL 3776777777777777777

Positive zero, also used as logical FALSE, is stored as 0. or OCTAL 0000000000000000000

Negative zero, also used as logical TRUE, is stored as -0. or OCTAL 7777777777777777777

The smallest nonzero negative REAL which fits in one computer word is -.3131513062514-293 or OCTAL 7776377777777777777

The largest negative REAL which fits in one computer word is -.1265014083171+323 or OCTAL 4001000000000000000

The largest positive INTEGER which can be output as a decimal number is 281474976710655 or OCTAL 0000777777777777777

Positive octal INTEGERS from 0001000000000000000 to 1776777777777777777 may be stored and added or subtracted but not multiplied or printed as decimals. For example

R or OCTAL 0001000000000000000

R or OCTAL 1776777777777777777

The largest negative INTEGER which can be output as a decimal number is -281474976710655 or OCTAL 7777000000000000000

Negative octal INTEGERS from 7776777777777777777 to 6001000000000000000 may be stored and added or subtracted but not multiplied or printed as decimals. For example

R or OCTAL 0777577777777777777

R or OCTAL 6001000000000000000

Four special forms are reserved for infinite or indefinite real results, and may be stored or moved but not used as real operands. They are determined by the first four ocal digits, and the other digits may be any octal characters.

A.6.1 Limiting Values for Integers

The largest integer which can be handled directly by the hardware is $\pm 576460752303423487 = 2^{59} - 1$. More stringent limits are sometimes imposed on integers used in certain contexts. If these limits are exceeded, no error message is given and results are unpredictable. The contexts and limits are:

CONTEXT	MAXIMUM INTEGER VALUE
integer multiplication or division	$\pm 281474976710655 = 2^{48} - 1$
index of DO-loop	$\pm 131071 = 2^{17} - 1$ (and greater than 0!)
integer mode to real mode conversion	$\pm 281474976710655 = 2^{48} - 1$
real mode to integer mode conversion	$\pm 576460752303423487 = 2^{59} - 1$ (but the real has less precision than this)
integer to be converted for output	$\pm 281474976710655 = 2^{48} - 1$
integer addition, subtraction, or input conversion	$\pm 576460752303423487 = 2^{59} - 1$

A.6.2 Limiting Values for Floating-Point (Reals, Complex, Double-Precision)

The magnitude of non-zero floating-point values which can be handled directly by the hardware must lie in the range .313151306251401E-293 to .126501408317068E323. Generating a value smaller than the lower limit results in a zero with no other indication that accuracy has been lost. Generating a value larger than the upper limit results in a special value called infinite. Infinite values may be tested for with the library subroutine LEGVAR. Use of a infinite value in a floating-point calculation will cause an arithmetic error detected by the hardware.

Reals are directly represented in floating point, so the above limits apply directly. Complex values are represented by a pair of floating-point values, so the real and imaginary parts of a complex value are each constrained by the same limits. Double-precision values are represented by a pair of floating-point values. The MSP (most significant part) is a regular floating-point value and is constrained by the given limits. The LSP (least significant part) is also a floating-point value, but it has a characteristic which is 48 less than that of its corresponding MSP. Thus, the LSP will be set to 0 when the MSP is 2^{48} times larger than the given lower limit for floating-point. In other words, half the accuracy of a double-precision value will disappear when the magnitude gets down to 2^{48}*.313151306251401E-293. At the other extreme, an LSP larger than 2^{-48}*.126501408317068E323 would imply that the corresponding MSP has become infinite.

TABLE C Samples of Powers of Ten

10^n	Octal Integer	n	-10^n	Negative Octal Integer
0	0000000000000000000	0	-0	7777777777777777777
1	0000000000000000001	0	-1	7777777777777777776
10	0000000000000000012	1	-10	7777777777777777765
100	0000000000000000144	2	-100	7777777777777777633
1000	0000000000000001750	3	-1000	7777777777777776027
10000	0000000000000023420	4	-10000	7777777777777754357
100000	0000000000000303240	5	-100000	7777777777777474537
1000000	0000000000003641100	6	-1000000	7777777777774136677
10000000	0000000000046113200	7	-10000000	7777777777731664577
100000000	0000000000575360400	8	-100000000	7777777777202417377
1000000000	0000000007346545000	9	-1000000000	7777777770431232777
10000000000	0000000112402762000	10	-10000000000	7777777665375015777
100000000000	0000001351035564000	11	-100000000000	7777776426742213777
1000000000000	0000016432451210000	12	-1000000000000	7777761345326567777
10000000000000	0000221411634520000	13	-10000000000000	7777553663143257777
100000000000000	0002657142036440000	14	-100000000000000	7775120635741337777

10.0^n	Octal Floating Point		-10.0^n	Negative Octal Floating Point
.0000001	1670655376246536257 3	-7	-.0000001	6107122401531241520 4
.0000010	1674414336750132755 5	-6	-.0000010	6103363441027645022 0
.0000100	1677517426542161551 0	-5	-.0000100	6100260351235616226 7
.0001000	1702643334272616103 2	-4	-.0001000	6075134443505161674 5
.0010000	1706406111564570652 0	-3	-.0010000	6071371666213207125 7
.0100000	1711507534121727024 4	-2	-.0100000	6066270243656050753 3
.1000000	1714631463146314631 5	-1	-.1000000	6063146314631463146 2
1	1720400000000000000 0	0	-1	6057377777777777777 7
10	1723500000000000000 0	1	-10	6054277777777777777 7
100	1726620000000000000 0	2	-100	6051157777777777777 7
1000	1731764000000000000 0	3	-1000	6046013777777777777 7
10000	1735470400000000000 0	4	-10000	6042307377777777777 7
100000	1740606500000000000 0	5	-100000	6037171277777777777 7
1000000	1743750220000000000 0	6	-1000000	6034027557777777777 7
10000000	1747461132000000000 0	7	-10000000	6030316645777777777 7
100000000	1752575360400000000 0	8	-100000000	6025202417377777777 7
1000000000	1755734654500000000 0	9	-1000000000	6022043123277777777 7
10000000000	1761452013710000000 0	10	-10000000000	6016325764067777777 7
100000000000	1764564416672000000 0	11	-100000000000	6013213361105777777 7
1000000000000	1767721522450400000 0	12	-1000000000000	6010056255327377777 7

TABLE D Octal Floating Point to Decimal Conversion for Numbers Near Unity

	0	1	2	3	4	5	6	7
17014	.00003	.00003	.00003	.00003	.00003	.00004	.00004	.00004
17015	.00004	.00004	.00004	.00004	.00004	.00004	.00004	.00004
17016	.00005	.00005	.00005	.00005	.00005	.00005	.00005	.00005
17017	.00005	.00005	.00006	.00006	.00006	.00006	.00006	.00006
17024	.00006	.00006	.00006	.00007	.00007	.00007	.00007	.00007
17025	.00008	.00008	.00008	.00008	.00008	.00009	.00009	.00009
17026	.00009	.00009	.00010	.00010	.00010	.00010	.00010	.00010
17027	.00011	.00011	.00011	.00011	.00011	.00012	.00012	.00012
17034	.00012	.00013	.00013	.00013	.00014	.00014	.00014	.00015
17035	.00015	.00016	.00016	.00016	.00017	.00017	.00018	.00018
17036	.00018	.00019	.00019	.00019	.00020	.00020	.00021	.00021
17037	.00021	.00022	.00022	.00023	.00023	.00023	.00024	.00024
17044	.00024	.00025	.00026	.00027	.00027	.00028	.00029	.00030
17045	.00031	.00031	.00032	.00033	.00034	.00034	.00035	.00036
17046	.00037	.00037	.00038	.00039	.00040	.00040	.00041	.00042
17047	.00043	.00043	.00044	.00045	.00046	.00047	.00047	.00048
17054	.00049	.00050	.00052	.00053	.00055	.00056	.00058	.00060
17055	.00061	.00063	.00064	.00066	.00067	.00069	.00070	.00072
17056	.00073	.00075	.00076	.00078	.00079	.00081	.00082	.00084
17057	.00085	.00087	.00089	.00090	.00092	.00093	.00095	.00096
17064	.00098	.00101	.00104	.00107	.00110	.00113	.00116	.00119
17065	.00122	.00125	.00128	.00131	.00134	.00137	.00140	.00143
17066	.00146	.00150	.00153	.00156	.00159	.00162	.00165	.00168
17067	.00171	.00174	.00177	.00180	.00183	.00186	.00189	.00192
17074	.00195	.00201	.00208	.00214	.00220	.00226	.00232	.00238
17075	.00244	.00250	.00256	.00262	.00269	.00275	.00281	.00287
17076	.00293	.00299	.00305	.00311	.00317	.00323	.00330	.00336
17077	.00342	.00348	.00354	.00360	.00366	.00372	.00378	.00385
17104	.00391	.00403	.00415	.00427	.00439	.00452	.00464	.00476
17105	.00488	.00500	.00513	.00525	.00537	.00549	.00562	.00574
17106	.00586	.00598	.00610	.00623	.00635	.00647	.00659	.00671
17107	.00684	.00696	.00708	.00720	.00732	.00745	.00757	.00769
17114	.00781	.00806	.00830	.00854	.00879	.00903	.00928	.00952
17115	.00977	.01001	.01025	.01050	.01074	.01099	.01123	.01147
17116	.01172	.01196	.01221	.01245	.01270	.01294	.01318	.01343
17117	.01367	.01392	.01416	.01440	.01465	.01489	.01514	.01538
17124	.01563	.01611	.01660	.01709	.01758	.01807	.01855	.01904
17125	.01953	.02002	.02051	.02100	.02148	.02197	.02246	.02295
17126	.02344	.02393	.02441	.02490	.02539	.02588	.02637	.02686
17127	.02734	.02783	.02832	.02881	.02930	.02979	.03027	.03076
17134	.03125	.03223	.03320	.03418	.03516	.03613	.03711	.03809
17135	.03906	.04004	.04102	.04199	.04297	.04395	.04492	.04590
17136	.04688	.04785	.04883	.04980	.05078	.05176	.05273	.05371
17137	.05469	.05566	.05664	.05762	.05859	.05957	.06055	.06152
17144	.06250	.06445	.06641	.06836	.07031	.07227	.07422	.07617
17145	.07813	.08008	.08203	.08398	.08594	.08789	.08984	.09180
17146	.09375	.09570	.09766	.09961	.10156	.10352	.10547	.10742
17147	.10938	.11133	.11328	.11523	.11719	.11914	.12109	.12305
17154	.12500	.12891	.13281	.13672	.14063	.14453	.14844	.15234
17155	.15625	.16016	.16406	.16797	.17188	.17578	.17969	.18359
17156	.18750	.19141	.19531	.19922	.20313	.20703	.21094	.21484
17157	.21875	.22266	.22656	.23047	.23438	.23828	.24219	.24609
17164	.25000	.25781	.26563	.27344	.28125	.28906	.29688	.30469
17165	.31250	.32031	.32813	.33594	.34375	.35156	.35938	.36719
17166	.37500	.38281	.39063	.39844	.40625	.41406	.42188	.42969
17167	.43750	.44531	.45313	.46094	.46875	.47656	.48438	.49219
17174	.50000	.51563	.53125	.54688	.56250	.57813	.59375	.60938
17175	.62500	.64063	.65625	.67188	.68750	.70313	.71875	.73438
17176	.75000	.76563	.78125	.79688	.81250	.82813	.84375	.85938
17177	.87500	.89063	.90625	.92188	.93750	.95313	.96875	.98438

TABLE D Octal Floating Point to Decimal Conversion for Numbers Near Unity

	0	1	2	3	4	5	6	7
17204	1.0000	1.0313	1.0625	1.0938	1.1250	1.1563	1.1875	1.2188
17205	1.2500	1.2813	1.3125	1.3438	1.3750	1.4063	1.4375	1.4688
17206	1.5000	1.5313	1.5625	1.5938	1.6250	1.6563	1.6875	1.7188
17207	1.7500	1.7813	1.8125	1.8438	1.8750	1.9063	1.9375	1.9688
17214	2.0000	2.0625	2.1250	2.1875	2.2500	2.3125	2.3750	2.4375
17215	2.5000	2.5625	2.6250	2.6875	2.7500	2.8125	2.8750	2.9375
17216	3.0000	3.0625	3.1250	3.1875	3.2500	3.3125	3.3750	3.4375
17217	3.5000	3.5625	3.6250	3.6875	3.7500	3.8125	3.8750	3.9375
17224	4.0000	4.1250	4.2500	4.3750	4.5000	4.6250	4.7500	4.8750
17225	5.0000	5.1250	5.2500	5.3750	5.5000	5.6250	5.7500	5.8750
17226	6.0000	6.1250	6.2500	6.3750	6.5000	6.6250	6.7500	6.8750
17227	7.0000	7.1250	7.2500	7.3750	7.5000	7.6250	7.7500	7.8750
17234	8.0000	8.2500	8.5000	8.7500	9.0000	9.2500	9.5000	9.7500
17235	10.000	10.250	10.500	10.750	11.000	11.250	11.500	11.750
17236	12.000	12.250	12.500	12.750	13.000	13.250	13.500	13.750
17237	14.000	14.250	14.500	14.750	15.000	15.250	15.500	15.750
17244	16.000	16.500	17.000	17.500	18.000	18.500	19.000	19.500
17245	20.000	20.500	21.000	21.500	22.000	22.500	23.000	23.500
17246	24.000	24.500	25.000	25.500	26.000	26.500	27.000	27.500
17247	28.000	28.500	29.000	29.500	30.000	30.500	31.000	31.500
17254	32.000	33.000	34.000	35.000	36.000	37.000	38.000	39.000
17255	40.000	41.000	42.000	43.000	44.000	45.000	46.000	47.000
17256	48.000	49.000	50.000	51.000	52.000	53.000	54.000	55.000
17257	56.000	57.000	58.000	59.000	60.000	61.000	62.000	63.000
17264	64.000	66.000	68.000	70.000	72.000	74.000	76.000	78.000
17265	80.000	82.000	84.000	86.000	88.000	90.000	92.000	94.000
17266	96.000	98.000	100.00	102.00	104.00	106.00	108.00	110.00
17267	112.00	114.00	116.00	118.00	120.00	122.00	124.00	126.00
17274	128.00	132.00	136.00	140.00	144.00	148.00	152.00	156.00
17275	160.00	164.00	168.00	172.00	176.00	180.00	184.00	188.00
17276	192.00	196.00	200.00	204.00	208.00	212.00	216.00	220.00
17277	224.00	228.00	232.00	236.00	240.00	244.00	248.00	252.00
17304	256.00	264.00	272.00	280.00	288.00	296.00	304.00	312.00
17305	320.00	328.00	336.00	344.00	352.00	360.00	368.00	376.00
17306	384.00	392.00	400.00	408.00	416.00	424.00	432.00	440.00
17307	448.00	456.00	464.00	472.00	480.00	488.00	496.00	504.00
17314	512.00	528.00	544.00	560.00	576.00	592.00	608.00	624.00
17315	640.00	656.00	672.00	688.00	704.00	720.00	736.00	752.00
17316	768.00	784.00	800.00	816.00	832.00	848.00	864.00	880.00
17317	896.00	912.00	928.00	944.00	960.00	976.00	992.00	1008.0
17324	1024.0	1056.0	1088.0	1120.0	1152.0	1184.0	1216.0	1248.0
17325	1280.0	1312.0	1344.0	1376.0	1408.0	1440.0	1472.0	1504.0
17326	1536.0	1568.0	1600.0	1632.0	1664.0	1696.0	1728.0	1760.0
17327	1792.0	1824.0	1856.0	1888.0	1920.0	1952.0	1984.0	2016.0
17334	2048.0	2112.0	2176.0	2240.0	2304.0	2368.0	2432.0	2496.0
17335	2560.0	2624.0	2688.0	2752.0	2816.0	2880.0	2944.0	3008.0
17336	3072.0	3136.0	3200.0	3264.0	3328.0	3392.0	3456.0	3520.0
17337	3584.0	3648.0	3712.0	3776.0	3840.0	3904.0	3968.0	4032.0
17344	4096.0	4224.0	4352.0	4480.0	4608.0	4736.0	4864.0	4992.0
17345	5120.0	5248.0	5376.0	5504.0	5632.0	5760.0	5888.0	6016.0
17346	6144.0	6272.0	6400.0	6528.0	6656.0	6784.0	6912.0	7040.0
17347	7168.0	7296.0	7424.0	7552.0	7680.0	7808.0	7936.0	8064.0
17354	8192.0	8448.0	8704.0	8960.0	9216.0	9472.0	9728.0	9984.0
17355	10240.	10496.	10752.	11008.	11264.	11520.	11776.	12032.
17356	12288.	12544.	12800.	13056.	13312.	13568.	13824.	14080.
17357	14336.	14592.	14848.	15104.	15360.	15616.	15872.	16128.
17364	16384.	16896.	17408.	17920.	18432.	18944.	19456.	19968.
17365	20480.	20992.	21504.	22016.	22528.	23040.	23552.	24064.
17366	24576.	25088.	25600.	26112.	26624.	27136.	27648.	28160.
17367	28672.	29184.	29696.	30208.	30720.	31232.	31744.	32256.

TABLE E FULL RANGE OCTAL FLOATING POINT TO DECIMAL CONVERSION

OCTAL	004	104	204	304	404	504	604	704
00	0.	4.0083-292	1.0261-289	2.6269-287	6.7249-285	1.7216-282	4.4072-280	1.1282-277
01	2.8883-275	7.3941-273	1.8929-270	4.8458-268	1.2405-265	3.1757-263	8.1299-261	2.0812-258
02	5.3280-256	1.3640-253	3.4918-251	8.9389-249	2.2884-246	5.8582-244	1.4997-241	3.8392-239
03	9.8284-237	2.5161-234	6.4411-232	1.6489-229	4.2213-227	1.0806-224	2.7665-222	7.0821-220
04	1.8130-217	4.6413-215	1.1882-212	3.0417-210	7.7869-208	1.9934-205	5.1032-203	1.3064-200
05	3.3444-198	8.5618-196	2.1918-193	5.6110-191	1.4364-188	3.6772-186	9.4137-184	2.4099-181
06	6.1694-179	1.5794-176	4.0432-174	1.0351-171	2.6497-169	6.7833-167	1.7365-164	4.4455-162
07	1.1381-159	2.9134-157	7.4583-155	1.9093-152	4.8879-150	1.2513-147	3.2033-145	8.2005-143
10	2.0993-140	5.3743-138	1.3758-135	3.5221-133	9.0166-131	2.3082-128	5.9091-126	1.5127-123
11	3.8726-121	9.9138-119	2.5379-116	6.4971-114	1.6633-111	4.2580-109	1.0900-106	2.7905-104
12	7.1437-102	1.8288E-99	4.6817E-97	1.1985E-94	3.0682E-92	7.8545E-90	2.0108E-87	5.1476E-85
13	1.3178E-82	3.3735E-80	8.6362E-78	2.2109E-75	5.6598E-73	1.4489E-70	3.7092E-68	9.4956E-66
14	2.4309E-63	6.2230E-61	1.5931E-58	4.0783E-56	1.0440E-53	2.6728E-51	6.8423E-49	1.7516E-46
15	4.4842E-44	1.1479E-41	2.9387E-39	7.5232E-37	1.9259E-34	4.9304E-32	1.2622E-29	3.2312E-27
16	8.2718E-25	2.1176E-22	5.4210E-20	1.3878E-17	3.5527E-15	9.0949E-13	2.3283E-10	5.9605E-08
*17	1.5259E-05	3.9063E-03	1.0000E+00	2.5600E+02	6.5536E+04	1.6777E+07	4.2950E+09	1.0995E+12
20	1.4074E+14	3.6029E+16	9.2234E+18	2.3612E+21	6.0446E+23	1.5474E+26	3.9614E+28	1.0141E+31
21	2.5961E+33	6.6461E+35	1.7014E+38	4.3556E+40	1.1150E+43	2.8545E+45	7.3075E+47	1.8707E+50
22	4.7890E+52	1.2260E+55	3.1386E+57	8.0347E+59	2.0569E+62	5.2656E+64	1.3480E+67	3.4509E+69
23	8.8342E+71	2.2616E+74	5.7896E+76	1.4821E+79	3.7943E+81	9.7133E+83	2.4866E+86	6.3657E+88
24	1.6296E+91	4.1718E+93	1.0680E+96	2.7341E+98	6.9992+100	1.7918+103	4.5870+105	1.1743+108
25	3.0061+110	7.6957+112	1.9701+115	5.0435+117	1.2911+120	3.3053+122	8.4615+124	2.1661+127
26	5.5453+129	1.4196+132	3.6342+134	9.3035+136	2.3817+139	6.0972+141	1.5609+144	3.9958+146
27	1.0229+149	2.6187+151	6.7039+153	1.7162+156	4.3935+158	1.1247+161	2.8793+163	7.3710+165
30	1.8870+168	4.8307+170	1.2367+173	3.1658+175	8.1045+177	2.0748+180	5.3114+182	1.3597+185
31	3.4809+187	8.9110+189	2.2812+192	5.8399+194	1.4950+197	3.8273+199	9.7978+201	2.5082+204
32	6.4211+206	1.6438+209	4.2081+211	1.0773+214	2.7578+216	7.0600+218	1.8074+221	4.6269+223
33	1.1845+226	3.0323+228	7.7626+230	1.9872+233	5.0873+235	1.3023+238	3.3340+240	8.5351+242
34	2.1850+245	5.5935+247	1.4319+250	3.6658+252	9.3844+254	2.4024+257	6.1502+259	1.5744+262
35	4.0306+264	1.0318+267	2.6415+269	6.7622+271	1.7311+274	4.4317+276	1.1345+279	2.9043+281
36	7.4351+283	1.9034+286	4.8727+288	1.2474+291	3.1933+293	8.1750+295	2.0928+298	5.3575+300
37	1.3715+303	3.5111+305	8.9885+307	2.3010+310	5.8907+312	1.5080+315	3.8605+317	9.8829+319

* Coordinates represent the five leftmost octal digits. Example: The first column of the starred line represents the octal word: 17 004 00000 00000 00000. Powers of 10 greater than +99 or less than -99 are represented as ± nnn, rather than E± nn.

TABLE F CONVERSION TABLE OF OCTAL-DECIMAL INTEGERS

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

Octal	Decimal
0000	0000
to	to
0777	0511
(Octal)	(Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

Octal	Decimal
1000	0512
to	to
1777	1023
(Octal)	(Decimal)

OCTAL/DECIMAL INTEGER CONVERSION TABLE (Cont'd)

2000 (Octal) to 2777 (Octal) = 1024 (Decimal) to 1535 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1519	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

3000 (Octal) to 3777 (Octal) = 1536 (Decimal) to 2047 (Decimal)

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

OCTAL/DECIMAL INTEGER CONVERSION TABLE (Cont'd)

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

4000	2048
to	to
4777	2559
(Octal)	(Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

5000	2560
to	to
5777	3071
(Octal)	(Decimal)

OCTAL/DECIMAL INTEGER CONVERSION TABLE (Cont'd)

Octal	Decimal
6000	3072
to	to
6777	3583
(Octal)	(Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

Octal	Decimal
7000	3584
to	to
7777	4095
(Octal)	(Decimal)

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3496	3497	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	4022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

TABLE G Powers of Two

2^n	n	2^{-n}
1	0	1.0000000000000000
2	1	.5000000000000000
4	2	.2500000000000000
8	3	.1250000000000000
16	4	.0625000000000000
32	5	.0312500000000000
64	6	.0156250000000000
128	7	.0078125000000000
256	8	.0039062500000000
512	9	.0019531250000000
1024	10	.0009765625000000
2048	11	.0004882812500000
4096	12	.0002441406250000
8192	13	.0001220703125000
16384	14	.0000610351562500
32768	15	.0000305175781250
65536	16	1.52587890625000000000000000D-05
131072	17	7.62939453125000000000000000D-06
262144	18	3.81469726562500000000000000D-06
524288	19	1.90734863281250000000000000D-06
1048576	20	9.53674316406250000000000000D-07
2097152	21	4.76837158203125000000000000D-07
4194304	22	2.38418579101562500000000000D-07
8388608	23	1.19209289550781250000000000D-07
16777216	24	5.96046447753906250000000000D-08
33554432	25	2.98023223876953125000000000D-08
67108864	26	1.49011611938476562500000000D-08
134217728	27	7.45058059692382812500000000D-09
268435456	28	3.72529029846191406250000000D-09
536870912	29	1.86264514923095703125000000D-09
1073741824	30	9.31322574615478515625000000D-10
2147483648	31	4.65661287307739257812500000D-10
4294967296	32	2.32830643653869628906250000D-10
8589934592	33	1.16415321826934814453125000D-10
17179869184	34	5.82076609134674072265625000D-11
34359738368	35	2.91038304567337036132812500D-11
68719476736	36	1.45519152283668518066406250D-11
137438953472	37	7.27595761418342590332031250D-12
274877906944	38	3.63797880709171295166015625D-12
549755813888	39	1.81898940354585647583007812D-12
1099511627776	40	9.09494701772928237915039062D-13
2199023255552	41	4.54747350886464118957519531D-13
4398046511104	42	2.27373675443232059478759765D-13
8796093022208	43	1.13686837721616029739379882D-13
17592186044416	44	5.68434188608080148696899414D-14
35184372088832	45	2.84217094304040074348449707D-14
70368744177664	46	1.42108547152020037174224853D-14
140737488355328	47	7.10542735760100185871124267D-15
281474976710656	48	3.55271367880050092935562133D-15
562949953421312	49	1.77635683940025046467781066D-15
1125899906842624	50	8.88178419700125232338905334D-16

TABLE H 6000 SERIES CHARACTER CODES

FORTRAN Source Language Character	Display Code	Card Punch Code
A	01	12-1
B	02	12-2
C	03	12-3
D	04	12-4
E	05	12-5
F	06	12-6
G	07	12-7
H	10	12-8
I	11	12-9
J	12	11-1
K	13	11-2
L	14	11-3
M	15	11-4
N	16	11-5
O	17	11-6
P	20	11-7
Q	21	11-8
R	22	11-9
S	23	0-2
T	24	0-3
U	25	0-4
V	26	0-5
W	27	0-6
X	30	0-7
Y	31	0-8
Z	32	0-9
0	33	0
1	34	1
2	35	2
3	36	3
4	37	4
5	40	5
6	41	6
7	42	7
8	43	8
9	44	9
+	45	12
-	46	11
*	47	11-8-4
/	50	0-1
(	51	0-8-4
)	52	12-8-4
$	53	11-8-3
=	54	8-3
blank(space)	55	space
,	56	0-8-3
.	57	12-8-3
Additional Printer Characters		
≡	60	0-8-6
[	61	8-7
]	62	0-8-2
:	63	8-2
≠	64	8-4
←	65	0-8-5
∨	66	11-0
∧	67	0-8-7
↑	70	11-8-5
↓	71	11-8-6
<	72	12-0
>	73	11-8-7
≤	74	8-5
≥	75	12-8-5
¬	76	12-8-6
;	77	12-8-7
not used	00	

Display codes are the octal codes stored in central memory to represent the above characters. Those characters which are not in the FORTRAN source language are only printable on the on-line printer.

TABLE I Summary of Control Codes for Keyboard Terminals

Keyboards come in several flavors; as a result there are variations in the key used to transmit a particular code to the computer. This table lists the keys used on the most common devices, model 33 and model 35 teletypes. If "shift" or "CTRL" appears it means that the shift or CTRL key is to be held down while the accompanying key is struck. "(CS)" means that both the shift key and the CTRL key are held down while the indicated key is struck.

Key	Action
RETURN	Return printer to column one and inform computer that the line is complete as sent
LINEFEED	Space printer to next line
SHIFT /	Delete previous character (prints "?")
SHIFT 1	Restart line (prints "!", no carriage return)
CTRL A	Exit from bell mode (characters sent in bell mode are discarded)
SHIFT 0	Prepare to enter 8-bit mode (prints left arrow)
(CS) P	Sign on code
CTRL Q	Enter or leave echo suppression mode (characters are not printed in this mode - used for passwords, etc.)

The following codes are sent only at the beginning of a line during input and are followed by a RETURN.

CTRL B	End of record (may be followed by two digit octal level. Level 17 is end of file). If sent during output, CTRL B means suppress multiple blanks.
CTRL D	End of job (i.e., end of information of fileset INPUT). If sent during output, CTRL terminates the current fileset.
CTRL I	Cancel job currently being entered

Further details are in the Remote Terminal System manual.

The following problems still exist in RUN 2.3B3:

1. Registers may be confused in a DO loop within a subroutine if the subroutine has six or more formal parameters and the DO loop does not end with a CONTINUE statement.

2. Blank cards preceding or following a COMPASS subroutine may cause trouble. Avoid them.

3. A statement such as

```
READ 91,I,J,X(I,J),I,J,X(I,J)
```

   does not execute properly: when the subscripts for the second X(I,J) are calculated, a temporary value is used which was calculated for the first X(I,J) and involves the first value of J. In other words, it behaves like

```
READ 91, I,J,X(I,J),I,ITEMP,X(I,J)
J=ITEMP
```

4. If the program name is included on the END card of a Fortran deck and the name begins with the letter F an erroneous no path diagnostic is given although the right code seems to be generated.

5. One program has been reported which produces an erroneous undefined statement number diagnostic and fails to detect a NO PATH to this statement error. Fixing this error eliminates the spurious diagnostic.

6. Expressions involving the in line functions REAL(Z) and AIMAG(Z) may not compile correctly. This may be avoided by using temporaries to hold the in line function values and substituting the temporaries in the required expression. In the case observed, the Washington RUN (RUNW) compiler produced correct code.

7. When IF statements using compound conditionals are used, the mode of arithmetic in one component of the compound may incorrectly dominate other components. Enclosing the components in parentheses appears to avoid the problem, e.g.

```
IF ((A .EQ. 5.3) .AND. (I .EQ. 10H          ))  STOP
```

   compiles correctly (but does not do so without the inner parentheses).

8. If a variable is mentioned in both a TYPE statement and a NAMELIST statement, the TYPE statement must appear first in the program.

9. An illegal branch into the range of a DO of the form:

```
      DO 1 I=1,N
      IF(BLOB) GO TO 1
      DO 1 J=1,M
1     CONTINUE
```

is not detected as an error. Do not use this construction. Time will not be refunded to you if you do.

10. The material concerning subprogram names discussed in 15.6 should properly be considered as bugs.

11. A subscripted formal parameter may not be used as a FORMAT declaration identifier.

Example:

```
      SUBROUTINE SUB(I)
      DIMENSION I(10)
          .
          .
          .
      WRITE (1,I(5)) J
          .
          .
          .
      END
```

produces the spurious diagnostic:

```
      UNRECOGNIZABLE STATEMENT
```

12. RUN sometimes generates a bad object deck (won't load) for a subprogram with twenty or more formal parameters.

13. Difficulties have been reported with ENCODE/DECODE statements using formal parameters.

14. Use of an undefined value as a subscript of an I/O list item sometimes causes spurious error messages, e.g.. Illegal format or list-format conflict.

15. Loop indices should be integer values.

GLOSSARY

All page, chapter and section references are to be found in this Guide.

Underlining is used to indicate terms which are described elsewhere in this glossary.

Absolute Address
: The adjusted address, after loading is completed, used to reference a given word in the storage of a computer.

Address
: A number used to locate a specific word in the storage of a computer.

Allocatable device
: A storage medium which can be shared by several jobs simultaneously; a disk is allocatable, a magnetic tape is not.

Alphanumeric
: An alphabetic or numeric character (A-Z and 0-9).

ANSI
: American National Standard Institute (Formerly USA Standard Institute and American Standards Association).

ASA
: See ANSI

ASCII
: American Standard Code for Information Interchance - a binary representation of characters which has been adopted as a national standard for transferring characters between electronic devices.

Assembler
: A translator which operates on computer-oriented symbolic language statements, generating the corresponding object code. COMPASS is an example.

Base
: Base of exponentiation in floating point notation. The base is often implicitly the same as the radix being used.

BCD
: An abbreviation for Binary Coded Decimal which refers to any one of several systems for representing each of the 10 decimal digits in addition to the 26 letters and sundry special characters by a unique sequence of bits.

Commonly but inaccurately used to describe the mode of recording information on tape in even parity.

Binary

The number system expressed to the base (radix) 2, consisting solely of combinations of the digits 0 and 1. This is the system employed internally by the CDC 6000 Series computers.

Binary Record

The data read or written by a single unformatted READ or WRITE statement.

Bit

One binary digit, either 0 or 1.

Block

A group of contiguous entities recorded on and read from a storage medium as a unit; it may contain a partial, one, or more record(s) and is synonymous with Physical Record.

Blocking

A method for making more efficient use of a storage medium by grouping or dividing the data being read from or written onto the medium into standard physical units. See Chapter 13.

Buffer

An internal temporary storage area in which data are accumulated during input and output operations. It serves to compensate for the difference in speed at which two communicating components perform operations. The storage of data in a buffer is analoguous to the storage of energy by a flywheel.

Byte

A group of adjacent bits which constitute a subdivision of a computer word. In CDC 6000 Series machines, a 60 bit word is composed of 5 bytes of twelve bits each.

CAL

The identifier of the Computer Center installation within VIM, an organization comprised of installations using CDC 6000 and 7000-series computers.

CALIDOSCOPE

CAL Improved Design of SCOPE. The name given to the operating system installed at CAL in 1971.

Carriage Control

The process of regulating the position of the paper in the printer to control the spacing of the printed lines. Double spacing and skipping to the top of a new page are examples.

Carriage Control Character

The first character of each line to be printed is used for carriage control and does not appear on the printout. See Section 10.1.

CDC 6400 Computer System
: A computer complex manufactured by the Control Data Corporation consisting of one central processor and ten peripheral processors plus assorted peripheral devices connected to the processors.

Central Memory (CM)
: See Memory

Central processor (CP or CPU)
: A computing unit which performs the computational operations within a CDC 6000 series machine.

Characteristic
: In floating point notation, the characteristic is the exponent of the base plus a constant used to eliminate negative values.

Character Set
: A set of unique visual representations (graphics) and its correspondence to a given set of sequences of bits. A character set usually includes the letters from A to Z, the digits from 0 to 9, and a selection of punctuation marks and mathematical symbols. In the CDC 6000 series machines there are 63 characters in the character set, each of which is represented by a given configuration of six bits (excluding the all-zero configuration). (See Appendix A, Table H.)

Character
: May refer to either a graphic or its corresponding bit sequence. See Character Set.

Coded
: Encoded in characters as in formatted I/O as opposed to binary form. (See Chapter 13.)

COMMON fileset
: A user created fileset which remains in the system after the job which created it is completed. It is available to subsequent jobs until it expires, a system deadstart occurs, or the fileset is released by the user.

Common Storage
: An area for storing data within the memory field of a Fortran program such that its contents may be referenced by more than one subprogram. Common blocks which are labelled are stored just before the first subprogram which references them. The unlabelled or blank common block is placed at the end of all the subprograms loaded. See Section 5.3.

COMPASS
: A hardware oriented symbolic programming language for CDC 6000 series computers which provides a COMPrehensive ASsembly System, designed for efficient usage of computer resources and maximum flexibility in program construction at the price of tedious and detailed coding.

Compiler

A translator, which operates on procedure- or application-oriented source language statements and generates the object code which corresponds to them.

Compiler Storage Map

A table generated by a compiler giving the relative addresses assigned to elements declared in a source program. See Section 14.2.3.

Control Record

The first logical record of the input for a job containing the job identification statement followed by the other control statements necessary to run the job.

Control Statement

Any one of several statements which are placed at the beginning of the job input and which indicate to the Operating System an operation to be performed. (See CALIDOSCOPE Control Statements.)

Core

A donut-shaped piece of magnetic material used to record one bit of information. Many computer memories are fabricated using cores so that "core" and "memory" are often used interchangeably.

Cpi

Characters per inch (see Density.)

Cross-Reference Table

A program-generated directory giving the relative address assigned to elements in a program and the relative addresses of the references to each of them.

Data Channel

The interface between a peripheral controller and a peripheral processor.

Dayfile

See Job Log and System Log.

Deadstart

The procedure of loading the Operating System from tape and placing it in execution. All previous information in the machine is erased, and the system is totally reinitialized. See recovery.

Debug

To locate errors in a program or hardware device.

Deck

A collection of punched cards; usually a set of related cards which have been punched for a definite purpose. See Job.

Deck Name

Seven or fewer characters associated for identification purposes with the object code generated for a subprogram. See Section 15.6.3 and Chapter 11.

Default

The value of an item which is assumed in the absence of a specific indication of what value the item should have. Usually applied to parameter values on control statements.

Density

A number which describes the physical spacing of information on a storage medium. On magnetic tape the unit of measure is characters per inch (cpi) and equals the number of frames which are recorded per inch within each physical record. Densities used on CDC tape units are 200 cpi (LO), 556 cpi (HI), and 800 cpi (HY).

Diagnostic

A message issued by a program indicating an error in its input or environment.

Direct-access fileset

See Random Access Fileset.

Disk

A circular metal plate coated with a material which provides a magnetic recording surface. Information is recorded onto concentric rings called tracks, each of which has an address. Reading and writing is done by means of one or more read/write heads mounted on movable or fixed arms. The operating system keeps filesets on disk and performs all the necessary bookkeeping so that filesets can be created, read, etc., by name rather than by disk address.

Disk driver

The set of peripheral processor programs which perform the read/write operations from and to the disk. Also called the stack processor because requests for disk I/O operations may be queued in a stack.

Disk fileset

A fileset whose storage medium is disk.

Disk pack

A removable set of disks.

Display code

The encoding of characters as six bit sequences used to represent coded information internally in the CDC 6400. See internal BCD.

Disposition code

A 12-bit code in the FET for a disk fileset which designates the manner in which the fileset will be disposed of when the job is terminated, e.g., print, punch (Hollerith), punch (binary), etc. Zero indicates the fileset is not to be further processed.

Dump

A print-out of the contents of locations in memory in a specified format, usually octal. (See Section 15.5). More generally, the print-out of the contents of any storage medium, e.g., a tape dump.

ECS

Extended Core Storage: An auxiliary storage device accessible via specific central processor instructions which serves as a relatively inexpensive and slower access core memory.

ECS Field

That part of the ECS allocated to a single job for access by instructions contained in the program.

ECS Fileset

A fileset whose storage medium is ECS.

End of file

A recorded configuration used to indicate the end of a file (see Chapter 13).

End of record

A recorded configuration (see Chapter 13) used to indicate the end of a SCOPE logical record. On punched cards, it is a card with 7,8, and 9 punches in column one.

Entry point

A labeled instruction in a subprogram which marks a point at which references may be made to the subprogram from other subprograms.

EOF

Abbreviation for End of file.

EOR

Abbreviation for End of record.

Error message

A message generated by a program to provide information about an error or a program malfunction. It is like a diagnostic except that is usually more serious. See Section 15.4.

Execution

The state describing the operation of a program in which the central processor of the computer functions according to the instructions of the program. Also, the state of a job while its control statements are being processed and the steps they call for are being performed.

External

An address referenced by a given subprogram which is not part of the same subprogram, e.g., a CALL or function reference (see entry point).

External BCD

A particular BCD representation used to represent a character set on even parity magnetic tape. The encoding used is the ANSI standard BCD encoding.

FET

An abbreviation for Fileset Environment Table.

Field

A set of one or more adjacent entities, such as memory words, card columns, printer positions, or bit positions, which are treated as a whole.

Field Length

The physical extent of a field. In particular the number of words in the Central Memory (or ECS) field of a job.

File

A collection of related information terminated by a unique mark (called an EOF) having a logical beginning and ending, existing in the computing system as an element of a fileset.

Fileset Environment Table

A table set up in Central Memory serving as a communication link between the system and the user's program. The system and user's program indicate the state of the processing of a fileset by setting fields in this table.

Fileset

A named collection of information which is accessible via the Operating System. It consists of one or more files.

Fileset name

The name by which a fileset is known to the Operating System. A fileset name is a letter followed by up to 6 alphanumeric characters. Sometimes called a logical fileset name or lfn.

Floating Point Notation

A notation similar to scientific notation by which a number is expressed as a signed decimal mantissa times some integral power (exponent) of ten (the base), e.g., $345.123 = .345123 \times 10^3$. This notation is used with binary radix in computers to increase the range of numbers which can be stored (see Appendix A).

Format

1. Any specification for arranging elements.
2. A Fortran statement specifying the arrangements of characters, fields, lines, punctuation, etc.

Formatted

Describes an I/O operation performed under the specifications of a Fortran FORMAT statement.

Frame

See Magnetic Tape.

Graphic

The visual representation of a character (as produced, for example, by a line printer or teletype).

Half Track

Another name for a disk record block used because the operating system allocates half of a track for a record block.

Hardware

1. The magnetic, mechanical, electrical and electronic devices or components of a computer.
2. (Slang) Any piece of automatic data/processing equipment.

Hollerith Code

Coding system for data in punched cards using one column per character. Named for inventor Herman Hollerith.

Input

Information or data transferred or to be transferred from an external storage medium into the internal storage of the computer. In general, anything that is received from an external source; e.g., input to a subprogram (i.e., parameters), input to a card reader, (i.e., a punched card), or input from a teletype line, etc.

Input/Output (I/O)

The process of transmitting information into or out of a computer via peripheral devices (see Chapters 9, 10, and 13).

Interface

A common boundary between automatic data-processing systems or between two devices of a single system.

Internal BCD

A BCD system used to store characters in a computer's memory. In particular, display code, the character code used in CDC 6000 series computers. (See Appendix A.)

Inter-record gap

An interval of space or time containing no information and signalling the end of a physical record on a storage medium such as tape. Also referred to as a record gap. The size of the inter-record gap for magnetic tape is about 3/4".

I/O

See Input/Output.

Job

1. The unit of work, consisting of a deck of cards or the equivalent input supplied via another input medium, beginning with a job identification statement and presented to the Operating System.
2. More specifically, a collection of one or more filesets, one of which is classified as the INPUT fileset, which the Operating System recognizes as an entity and which contains a description of the operations the computer has to perform. Possible elements include local filesets and COMMON filesets.

Job card

See job identification statement.

Job deck

See Job 1.

Job identification statement

The first statement of any job, containing the number of the account to be billed and various limit and control parameters. Called a job card when the job is punched on cards. See Guide to Computer Center Services and CALIDOSCOPE Control Statements for details.

Job log

A chronological summary of significant occurrences during the processing of a job. It is printed at the beginning of each job's output.

Job step

One of the divisions in the execution of a job which begins when a control statement is processed and (except for load sequences) ends when a subsequent control statement is processed or job execution terminates.

Justify

To position the contents of a field such that they either begin in the left-most position (left-justify) or end in the right-most position (right-justify).

Keyword

1. An informative word in a title or text.
2. An identifier on a control statement specifying one of several possible parameters.

Level number

A number (from 1 to 17B) associated with each logical record which serves to organize the records in a fileset in a hierarchy (see Chapter 13).

Lfn Abbreviation for logical fileset name.

Library

A collection of subprograms which are selectively loaded, as required, to satisfy the external references in a program already loaded.

Line See Unit record

Linking

During loading, the process of supplying for each external reference occurring in a subprogram loaded into memory, the absolute address of the corresponding entry point. More generally, the act of establishing a connection between two elements.

Listing

A printed representation of coded information.

Literal

A data item which is made up of a string of characters, e.g., a Hollerith constant.

Load Address

The absolute address of the first word of a subprogram after it has been loaded.

Load map

The directory produced on the output fileset by the Loader showing the absolute addresses at which the object code for various subprograms and Common Storage blocks were placed in memory. A reference listing may also be included. See example in Chapter 14.

Load sequence

A sequence of one or more control statements which load and (optionally) execute a user's program.

Loader

A program which accepts translator generated object code subprograms as input and places them in central memory in proper form for execution as one absolute program (see Section 14.2.4). It may also produce program overlays on secondary storage media (see Chapter 8).

Loading

The action performed by a Loader.

Location

See Address.

Logical fileset name.

See Fileset name.

Logical Record

A data structure within a fileset which is recognized by the operating system and which consists of a collection of information terminated by an end of record. The collection begins with the preceding end of record or, for the first logical record in a fileset, the beginning of the fileset. See Chapter 13.

Machine language

The set of instructions (and/or the binary representation of a particular sequence of instructions) which is directly interpretable by the machine hardware.

A programming language which represents the hardware instructions more or less directly.

See Object code.

Magnetic tape

A ribbon of plastic, coated with a metallic oxide that accepts and holds magnetism. The recording surface of the tape is divided lengthwise into tracks and crosswise into frames. These define a matrix in which the presence or absence of tiny spots of magnetism called bits represent the information. See Chapter 13. See entries for parity.

Main program

The program element (defined in Fortran by a PROGRAM statement) to which control is transferred after loading has been completed. A Fortran main program contains the definitions of the filesets to be used for input and output (see Section 7.4).

Mantissa

The coefficient which is to be multipled by the base raised to the exponent power to determine the value of a number in floating-point notation. The mantissa is usually in the form of a normalized fraction. See Floating Point Notation.

Memory

The internal hardware device used to store information to be used by the central processor. Also referred to as Central Memory, core or CM.

Memory Field

That portion of memory assigned to the processing of a job.

Microsecond

One millionth of a second.

Millisecond

One thousandth of a second.

Normalized

A fraction is normalized with respect to a radix if its positional representation in that radix has a non zero digit following the radix point. A number in floating point notation is normalized if its mantissa is a normalized fraction or the number is zero.

Object code

A sequence of instructions needing only linking and relocation to be directly comprehensible to the computer, to perform a given set of arithmetic and logic operations which solve a given problem. Also referred to as object language, machine language, or object program.

Octal

The number system to the radix 8. Octal numbers may be derived from binary numbers by grouping the bits into sets of three going away from the binary point and converting each group to the corresponding octal digit.

Off-line

Operation of input/output and other devices not under direct computer control.

On-line

Operation of an input/output device as a component of the computer, under programmed control.

Operating System

An organized collection of programs under whose supervision and control jobs are processed.

Output

Information transferred from the internal storage of a computer to any one of several external storage media, e.g., paper, cards, magnetic tape. In general anything which is communicated to the outside.

Overlay

The technique used to run a job in a smaller area of memory than would be required if the entire program resided in memory simultaneously. Overlays are stored on a secondary storage device and are read into memory when needed, replacing program elements which are no longer needed (see Chapter 8).

Parity

The remainder modulo 2 (i.e., even or odd) of the count of the non-zero bits in a given group.

Parity: block, track, or longitudinal

The parity of the bits in a given track of a block on tape, also, by extension, the parity of all tracks in the block.

Parity bit

The seventh bit in a frame on magnetic tape representing the parity check information.

Parity: character or frame

The parity of the bits in each frame across the width of the tape.

Parity check

A method for detecting errors in stored data (caused on tape, for example, by dust or skew while reading/writing). Also, an error detected by performing a parity check, i.e., a parity error.

Parity: even

A checking method which counts one bits in a given group and appends a 1 or 0 as the parity bit such that the total of ones is even.

Parity: odd

A checking method which counts the one bits in a given group and appends a 1 or 0 as the parity bit such that the total of ones is odd.

Peripheral controller

A piece of equipment which acts as an intermediary control device, linking a peripheral unit to a data channel or in the case of off-line operation, to another peripheral unit.

Peripheral processor (PP)

A computer which performs input/output or monitor functions within a CDC 6000 series computer system.

Peripheral unit or device
: One of various machines used in combination or conjunction with the computer but not part of the computer itself, e.g., card reader, line printer, magnetic tape unit.

Physical record
: A block of information recorded on a physical medium and constituting the amount of information which was recorded on that medium as a unit. It is the minimum amount of information which can be read from the medium. (see Blocking.)

Physical Record Unit (PRU)
: The maximum size for physical records to be read from a particular medium. For most media this is fixed but it may be adjusted in the case of magnetic tape (see Chapter 13).

Print-out
: Output produced on a printer.

Process
: A generic term that may include compute, assemble, compile, interpret, execute, generate, etc., i.e., perform a specific computation action.

Program
: The collection of subprograms needed to solve a given problem on a computer. Also used commonly to refer to a single subprogram.

PRU
: See Physical Record Unit.

Public Fileset
: A fileset which is permanently available to jobs in the system, i.e., is guaranteed to be available whenever user jobs are being processed.

Radix
: The base of a number system, i.e., a quantity that defines a system of representing numbers by positional notation.

Random access fileset
: A fileset in which individual logical records are directly accessible regardless of their position. See Section 13.0.1.

Record
: See Binary Record, Logical Record, Physical Record and Unit Record.

Record block
: The unit in which space on an allocatable device is allocated to a fileset, e.g., a disk record block is fifty sectors of 64 words each.

Reference Address (RA)
: The hardware address in Central Memory which serves as the origin for the loading of a job. It defines the first word of the memory field for the job (the job uses 0 to address this word).

Relative Address
: The address used to identify a word in a subprogram with respect to its relative position in that subprogram. The starting address is relative location 0. A relative address is translated into an absolute address by addition of the specific load address serving as the origin of that subprogram.

Relocatable binary
: The form of the object code produced by many translators, RUN in particular.

Relocation
: The method by which the object code for a subprogram is placed in memory starting at a load address.

Register
: A device for the temporary storage of one or more words to facilitate arithmetic, logical, or transfer operation. CDC 6000 series computers have eight address registers (A0-A7), eight index registers (B0-B7), and eight operand registers (X0-X7). Also, a memory cell devoted to a specific function.

Remote Terminal System
: The CALIDOSCOPE subsystem which communicates with keyboard terminals.

Ring
: See Write Enable Ring.

Rollout
: A technique used to improve turnaround for high-priority jobs by temporarily saving the contents of the memory field of a lower priority job on disk or ECS to allow other jobs to get the memory they require. When the higher priority tasks are complete, the lower priority job is "rolled in" and its processing resumed.

RTS
: See Remote Terminal System.

RUN compiler
: The name given to the particular Fortran compiler documented in this manual.

Run
: The act of submitting a job to the computer for processing - including all phases of processing.

RUNW
: A version of RUN produced by the University of Washington.

SCOPE
: The manufacturer's Operating System for the CDC 6000 series computers. CALIDOSCOPE is derived from version 3.2 and 3.3 of SCOPE.

Sector

One of the 100 equal length areas into which a track on a disk is divided; a disk PRU.

Sequential access fileset

A fileset in which individual logical records are accessible only after the preceding records have been accessed, regardless of whether all the information or only some of it is desired. See Section 13.0.1.

Skew

Misalignment of the bits in a frame on tape due to some physical distortion of the tape.

Software

Various internal programs or routines professionally prepared to facilitate the user's efficient operation of the computer equipment (hardware); e.g., compilers, assemblers, operating systems. More generally, any programs used on a computer.

Source language

A language suitable for input to a translator (as opposed to object code).

Source listing

Output generated by a compiler or assembler in which each statement of the source language input is given. Diagnostics for any detected errors which occur may also be given.

Source program

The sequence of statements which are input to a translator (e.g., Fortran). See Section 1.2.

Stack processor

See Disk Driver.

Storage map

See Compiler Storage Map and Load Map.

Storage medium

A material on which information may be recorded; in particular, a material on which information may be both recorded and retrieved through I/O devices attached to a computer.

Subprogram

A program element which can be compiled separately and may be linked with other program elements to create an executable program. Used to refer to a subroutine, a main program, a function, a block data subprogram or a library subprogram which is not in any of these categories. Subroutines, functions, and main programs are called "procedural subprograms". See Chapter 7.

Subroutine

A subprogram which depends upon other subprograms for its activation. In Fortran a subroutine is activated by a CALL statement.

Syntax

Rules governing statement structure in a language.

System log

Combined job log for all jobs run during a given period, also containing certain operational information.

Tape

See Magnetic Tape.

Tape Drive (Unit)

The hardware mechanism that moves magnetic or paper tape past read/write heads for purposes of information transfer.

Tape Driver

The set of peripheral processor programs which perform the input/output operations from and to magnetic tape.

Tape Mark

A pattern recorded on tape to signal the end of valid information. It consists of an octal 17 written in even parity followed by a block parity check character.

Traceback

A listing of the sequence of subprogram linkages in the path from the main program to the current subprogram, i.e., an ordered listing of CALL, function references, or implicit library calls which have been executed and for which no corresponding RETURN has been executed.

Tracing

A technique for debugging a routine whereby during the execution of its instructions, information concerning the status of registers, storage locations, location counters, variables, etc., are transmitted to an output device in the same sequence in which the traced instructions are executed.

Track

The portion of a moving-storage medium, such as a magnetic tape or disk, onto or from which sequences of bits may be written or read one at a time in a serial fashion.

Translator

A program which operates on statements written in one language (called the source language) and generates a set of corresponding statements in another language (called the target language or in some cases, object code).

Turnaround

The length of time from when a job is submitted for processing until the results are available.

Unformatted

Describes an operation performed on data taken just as it is found. Used specifically to refer to a Fortran read or write operation which has no FORMAT statement associated with it.

Unit record

In central memory, a sequence of coded characters whose end is indicated by a terminator (twelve or more bits of zero) which fills out a CM word. Externally, a unit record may be a printed line or a punched card.

Unsatisfied external

An external reference which appears in a subprogram and which does not correspond to an entry point in any subprogram loaded with it or in any subprogram in the library. A list of unsatisfied externals, if any were found, appears at the end of the load map.

USASI

See ANSI.

Volume

A physical unit of a storage medium, e.g., a reel of magnetic tape, a disk pack.

Write Enable Ring

A ring physically inserted into a reel of magnetic tape to allow information to be written on it.

Write Ring

See Write Enable Ring.

Word

A unit of information within memory. The information may be data to be operated on by the central processor or instructions for the central processor to execute a program. In CDC 6000 series machines, a word consists of 60 bits.

INDEX TO RUN GUIDE

-A-

-B-

-C-

-D-

-E-

-F-

-M-

COMMENT SHEET

DOCUMENT TITLE_____

Publication: Number_____ Date_____

FROM: Name_____ Phone_____

Address: _____

_____

_____

Please check one:

Faculty __ Student __ Graduate __ Undergraduate __ Staff __ Other _____

COMMENTS:

Your evaluation of this document will be welcomed by the Computer Center. Any errors (Please indicate page number), suggested additions or deletions, or general comments may be made below.

FOLD HERE

FOLD HERE

Fold in half, staple and return to address on reverse side of this form.

Staple

University of California
Computer Center Library
239 Evans Hall
Berkeley, Calif. 94720