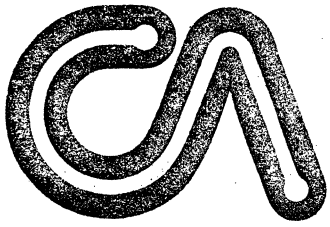


ComputerAutomation

NAKED MINI® Division

18651 Von Karman, Irvine, California 92715
Telephone: (714) 833-8830 TWX: 910-595-1767

FORTRAN IV REFERENCE MANUAL

90-96510-00B0

April 1976

PRINTED IN THE U.S.A.

REVISION HISTORY

Revision	Description	Date
A0	Original issue	
A1-A6	Various minor updates	
B0	New printing. Reflects capability of running Fortran on LSI-3/05	April 1976

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

Chapter Page

Chapter 5. INPUT/OUTPUT

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

List of Illustrations

List of Tables

INTRODUCTION

FORTRAN is an algebraic language designed primarily for use in scientific and mathematical applications. The name stands for FORmula TRANslation, because many of the statements are represented as formulas. For example, the formula

$$X = 8.1 + Y - a \cdot Y^2/Beta$$

can be written in FORTRAN as

```
X = 8.1 + Y - A*Y**2/BETA
```

The first FORTRAN was developed in the middle 1950's. It was soon followed by a version called FORTRAN II, in which several new features were added (notably user subroutines and common storage). FORTRAN IV appeared in the early 1960's, incorporating more new features, such as logical expressions, type declarations, double precision and complex data, data initialization, and labeled COMMON. As various manufacturers and universities continued to add other new features, a committee of the American Standards Association (now called the American National Standards Institute, ANSI) was formed to document "standard" FORTRAN. They documented two: Basic FORTRAN, which was similar to FORTRAN II; and FORTRAN, which was essentially FORTRAN IV.

This standard was intended to function, and has functioned, as a minimum acceptable standard. Virtually every FORTRAN IV in existence includes additional features beyond the standard.

Computer Automation FORTRAN IV contains ANSI FORTRAN as a subset. Some of the additional features are:

- In-line assembly language (particularly for real time)
- Simplified input/output (no FORMAT statement needed)
- Generalized subscripts (any integer expression)
- Alphanumeric strings
- Memory-to-memory data conversion (ENCODE/DECODE)
- End-of-file processing (END= option)
- Automatic double precision

A more complete list of extensions to ANSI FORTRAN may be found in appendix D.

The FORTRAN compiler accepts programs written in the FORTRAN IV language (called source programs) and translates them into machine language programs (called object programs), meanwhile producing a simulated assembly language listing of the object program (called an object listing) and diagnostics for any errors detected in the use of the FORTRAN IV language. The diagnostics and object listing in Computer Automation FORTRAN IV are designed to be readable and understandable, to assist in understanding what the compiler has done.

The Computer Automation FORTRAN IV compiler runs on a large LSI machine operating with a Computer Automation Operating System, but is optimized to produce small object programs that can run on small machines with RTX (the Computer Automation Real Time Executive). There is also a library of subroutines to provide support operations, such as input/output and floating point computations, as well as mathematical functions, such as logarithm and square root. The library is as modular as possible, so that only those portions actually needed will be loaded with the object program.

This reference manual describes the Computer Automation FORTRAN IV language and makes it possible to write FORTRAN programs. Further information on the use of the compiler, the run time library, linking, and system generation may be found in the FORTRAN IV Operations Manual (96510-01).

CHAPTER 1

PROGRAMS

KINDS OF PROGRAMS

A FORTRAN program may be one of three things: a main program, a subprogram, or a task. When loaded into memory for execution, there must be one and only one main program. Execution begins at the first statement of the main program or task. There may be any number (including none) of subprograms. A subprogram, which may be either a SUBROUTINE or FUNCTION, always has a name with which it is called by other programs. A task also has a name, but it is not called in the usual way; it is connected to a real time interrupt. Subprograms and tasks are described in subsequent chapters. All programs end with an END line.

CODING FORM

Lines of FORTRAN source language are prepared in 80-character, "card image" form. Each line has four fields:

Columns 1-5	Label. A statement may have a label in order to be referenced by other statements. A label is a decimal integer in which all blanks and leading zeros are ignored. Chapter 4 describes the use of labels.
Column 6	Continuation mark. Normally this column contains a blank (or zero). If a statement needs to be continued on more than one line, the succeeding lines must have some character other than blank or zero in this position. Digits appearing in columns 1-5 of a continuation line are ignored. Any number of continuation lines may be used.
Columns 7-72	Statement. The FORTRAN statement may begin anywhere in this field and may have blanks interspersed for readability (except within alphanumeric fields; see chapter 2).
Columns 73-80	Identification. These columns are ignored by the compiler and may be used for program/subprogram names and/or sequence numbers.

If the source lines are prepared on a medium other than cards, it is not necessary that all 80 columns appear. The line may be terminated at any point by a carriage return.

Figure 1-1 illustrates the use of these fields in preparing source language input.

COMPUTER AUTOMATION INC.
the NAKED MINI company
19651 Von Karman Irvine, Calif. 92664
tel 714 833 8830 twx 910 595 1767

FORTRAN CODING FORM

PROGRAM	QUADRATIC SOLUTION	GRAPHIC	Ø	O	ø	1	I	l		PAGE 1 OF 1
PROGRAMMER	S. POTTER	PUNCH	OH	OH	ZERO	EYE	EYE	ONE		DATE 9-74

↓—C FOR COMMENT

STATEMENT LABEL (1–5)	NOT (6)	FORTRAN STATEMENT (7–72)	IDENTIFICATION (73–80)
C		GIVEN COEFFICIENTS A, B, C	
C		SOLVE FOR X IN QUADRATIC	
C		A**X**2 + B*X + C = Ø	
C		READ COEFFICIENTS	
1Ø		INPUT A, B, C	
		VAL = B**2 - 4*A*C	
C		IF VAL IS NEGATIVE, ROOTS ARE UNREAL, OUT OF SIGHT	
		IF (VAL .GE. Ø) GO TO 3	
		OUTPUT 'NO REAL ROOTS',	
	1	'VAL = ', VAL	
		GO TO 7	
C		OTHERWISE COMPUTE ROOTS	
3		R1 = (-B+SQRT(VAL))/(2*A)	
		R2 = (-B-SQRT(VAL))/(2*A)	
C		PRINT ROOTS	
		OUTPUT 'ROOTS ARE', R1, R2	
C		WAIT FOR SIGNAL TO START OVER	
7		PAUSE	
		GO TO 1Ø	
		END	

1.2

COMPUTER AUTOMATION

Figure 1-1 Sample Program

COMMENTS

Any line with a C in column 1 is a comment line and has no effect on the program. The remaining columns are ignored and may contain anything in any position. We recommend liberal use of comments to document the operation of programs. Comment lines may appear anywhere except within a continued statement (i.e. preceding continuation lines).

CONDITIONAL COMPILATION

For debugging purposes, a statement may be written on a line with an X in column 1. How the statement is treated then depends on an option specified at the time the program is compiled (see chapter 9). If the XON option is specified, the X is ignored and the statement is processed normally. If the XON option is Not specified, the X is interpreted as if it were a C; that is, the line becomes a comment line.

During checkout, additional debugging lines can be included (for example, intermediate output) with an X in column 1. While the XON option is in force, these lines will take effect. When checkout is complete, the program can be recompiled without the XON option and these additional lines will become comments. As such, they both document the debugging that was used and can be reinstated quickly if needed.

A statement beginning on an X line may be continued only on an X line. A normal statement, however, may have a continuation line that either has an X or does not. For example:

C FOR COMMENT

STATEMENT LABEL	CONT	FORTRAN STATEMENT
1 5	6	7 10 15 20 25 30 35 40 45 50
		OUTPUT A,I,, RATE,, VALUE
X	2	,,'OLD VIALUE =',, OLDVAL
		BETA = 3.1* SQRT(VALUE)
X		IF (BETA .LT. .05) OUTPUT BETA
X	2	,'TOO SMALL.. WHAT DO I DO NOW?'

CHARACTER SET

Computer Automation FORTRAN IV accepts source programs (and data) in the ASCII (American Standard Code for Information Interchange) standard character set. The characters normally found in source programs are the 26 letters, the 10 decimal digits, and the following special characters:

+ - * / = < > ( ) . , : ' # $ @ ↑ blank

Other special characters may appear, for example in alphanumeric strings, but only the following are printable on all Computer Automation supported devices:

; ? % & ! " ←

The ASCII character set is shown in appendix C.

In some examples in this manual, the character blank is represented by b, so that it is possible to see exactly how many there are.

SAMPLE PROGRAM

Figure 1-1 shows a sample FORTRAN program prepared on a typical FORTRAN coding form. The lines with a C in column 1 are comments. Labels appear anywhere in columns 1-5. Their value does not imply any ordering of statements. It is simply an identification. The statements appear within columns 7-72. One of them has a continuation line, marked in column 6.

This program computes the roots of a quadratic equation, according to the formula

$$X = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

First it reads in the three coefficients, in a free form, separated by commas (or on separate lines). If the discriminant is negative, indicating no real roots, a message is printed to that effect. Otherwise the two roots are evaluated and printed. The program then waits for a signal from the computer operator to return to the beginning and read in another set of coefficients.

This program is a main program, because it does not begin with a SUBROUTINE, FUNCTION, or TASK statement. Therefore, when executed, processing automatically begins at the first statement, which is the INPUT statement with the label 10.

CHAPTER 2

ELEMENTS OF EXPRESSIONS

At the heart of the FORTRAN language is the computation of formulas. This is done with the assignment statement, which computes a value and assigns it to a variable. For example,

```
X = 4.5 + SIN(ALPHA)
```

The value to be computed is represented, on the right side of the equal sign, by an expression. The elements of an expression are connected by various operators, described in the next chapter. These elements may be one of three things: a constant, a variable, or a function reference. In the above example, 4.5 is a constant. It is referenced by value and never changes. There are various types of constants, as described below. X and ALPHA are variables. They are like "unknowns" in a formula. Their value can change, either by being input or by being assigned a value, as with X above. SIN(ALPHA) is a function reference, calling the function SIN with the one argument ALPHA.

NAMES

Variables and functions are identified by name (as are subroutines and COMMON blocks, described in subsequent chapters). A FORTRAN name must begin with a letter and may contain letters and digits. A name may be of any length, but only the first six characters are significant. Names longer than six characters may be used to improve readability, but care must be taken that no two of them have the same first six characters.

Examples:

```
X        A        R234        NUMBER        MASSACHUSETTS
```

In the last case, only the first six characters, MASSAC are recognized, so a name such as MASS ACTION would be considered identical. Note that within names (as in most places in FORTRAN), blanks are ignored, so MASS ACTION is the same as MASSACTION.

Computer Automation FORTRAN IV has no reserved names that are unavailable to the user. However, to avoid ambiguity we recommend that you avoid names that are the same as FORTRAN commands (e.g. READ, DO, IF).

DATA TYPES

Computations in FORTRAN may be done in various modes. For each mode there is a corresponding data type. In some cases types may be intermixed and in some they may not. This is discussed in the next chapter. The most important distinction to keep in mind is that between integer and floating point arithmetic.

Integer arithmetic deals only with integers (whole numbers) in a restricted range, and is used mainly for counting and subscripting. It is the fastest form of arithmetic. Floating point arithmetic handles the continuum of real numbers (including fractional values) over a wide range. However, the values are binary approximations to decimal numbers, which may or may not be exact. (There are two degrees of precision available, as described below.) Floating point computations are significantly slower than integer computations.

Computer Automation FORTRAN IV includes these six data types, as described below:

Integer	Complex
Real	Logical
Double Precision	Alphanumeric String

Integer

Integer values must lie in the range -32768 to +32767; that is, $-2^{15} \leq n \leq +2^{15} - 1$. (The value +32768 cannot be represented on a 16-bit 2's complement machine. Therefore, be careful to use the negative value -32768 only when it stands alone, such as on the right of an equal sign.) Examples of integer constants are:

1 27 -4197 0 30000

Normally, variables and functions whose names begin with I, J, K, L, M, or N are of integer type. (This is called the IJKLMN rule.) It is possible, however, to explicitly declare certain names to have a different type (see chapter 6). In the absence of such declarations, the following names would be integer:

NAKED MINI LSI2 I J KISMET

Integer constants may also be written in hexadecimal form or in Hollerith (alphanumeric) form. These are described below.

Integer values occupy one word (16 bits) of machine storage.†

† However, see Chapter 9 for the ANSI allocation option.

Real

Real arithmetic is the single precision form of floating point arithmetic. Real values must be in the range 1.7×10^{38} to 1.47×10^{-39}, that is, 2^{127} to 2^{-129}, or be zero. They may be positive or negative within this range. They have a precision of somewhat more than 7 significant digits, and they occupy two words (32 bits) of storage.

Real constants may be written in any of several ways. To be recognized as floating point, they must have either a decimal point or an exponent included. An exponent is a power of ten by which the value is to be multiplied. It follows the numeric value and consists of the letter E and a signed or unsigned integer. For example:

3874.73	12.	.0099	0.
6.601E15	9.E-7	.37E 31	93E+6

In the case of 93E+6, this represents 93×10^{6}, or 93,000,000, and could equally well be written as 93000000. or .93E8. Note that the plus sign preceding the exponent value is optional, and that the decimal point is not required if there is an exponent.

Real constants may be written with any number of digits, but only the first seven significant ones will be processed. It is permissible, however, to write real constants such as:

123456000000.0 or .0000054321

Unless declared otherwise, variables and functions whose names begin with anything other than I, J, K, L, M, or N are of real type. For example:

ALPHA HEIGHT OHMS Z SNEWO FROG

Double Precision

Double precision arithmetic is the same as real arithmetic, except that it carries about 17 digits of precision. A double precision value must lie in the same range as a real value. It occupies four words (64 bits) of storage.

A double precision constant must have a special exponent to identify it as double precision. This exponent is the same as for real except that a D is used instead of an E. Since the exponent is always present, a decimal point need not be. The following are double precision constants:

2.718281828459046D0 1.D 19 .3D-10 7148838830D-7

Note in the first case that an exponent of zero is used, since there must be an exponent. The mere presence of more than seven digits does not make a constant double precision. However, if a floating point constant appears in a double precision expression, it will automatically become a double precision constant, regardless of how many digits were written or whether a "D" exponent was used (see "Mixed Mode Expressions" in the next chapter). There is also an Automatic Double Precision option that will force all floating point constants into double precision (see below)

Since the IJKLMN rule classifies all names as either integer or real, a variable or function can only be double precision if it is explicitly declared to be so,† as described in chapter 6. In this case, it makes no difference what letter the name begins with, although "D" is mnemonically pleasing.

Complex

Complex numbers consist of a real and imaginary part, in that order. Each part is itself a single precision (real) value, with the precision and range of a real value. Complex quantities occupy four words (64 bits) of storage. The first two words are the real part, the second two are the imaginary part.

Complex constants are written with the real and imaginary parts separated by a comma and enclosed in parentheses. For example,

```
(3.37,2.0)
```

represents the complex number 3.37+2i, where $i = \sqrt{-1}$. Other complex constants:

```
(.001,1E-7)      (-4.,+7.5)          (1.,0.)      (0.,1.)
```

Note that the latter two examples, although represented in complex form, have values that are purely real and purely imaginary, respectively.

As with double precision, a variable or function can only be complex if its name is explicitly declared so (except for a few library functions). "C" is often used as the first letter of such names.

Logical

This is the last of the unique types. The others are variations of one sort or another. Logical is a very special type that is not numeric at all. It is used for the testing of

†Certain library functions, such as DSQRT, are automatically recognized to be double precision. See chapter 7.

conditions and has only the values "true" and "false". A logical quantity occupies one word (16 bits) of storage. However, only the first bit (the sign bit) is significant. It is 1 when the value is "true", 0 when it is "false".

To a large extent, logical type is used mainly for testing the result of relational expressions that compare numeric values. For example, the statement

```
IF (X+Y > 3.5) STOP
```

contains no variables or constants of logical type, but the relational expression (X+Y > 3.5) has a logical value of "true" or "false". However, elements of logical type may be used.

A logical constant may be either

```
.TRUE.    or    .FALSE.
```

written exactly as shown with the periods at either end. Variables and functions are logical only if their names are explicitly declared so. Logical quantities are combined by means of a different set of operators than those used with numeric quantities. The following chapter will describe logical expressions.

Logical operations should not be confused with Boolean operations, in which a whole string of 1's and 0's are ANDed or ORed together (see below, under "Boolean Operations").

Hexadecimal

Integer constants may also be written in hexadecimal form. This is mainly useful where the value is not numeric but is a bit pattern of some sort, e.g. for testing or masking, or to represent an unusual alphanumeric character that has no graphic representation. A hexadecimal constant may be written in either of two forms:

```
:xxxx      or      nZxxxx
```

where: x is a hexadecimal digit, i.e. one of 0 1 2 3 4 5 6 7 8 9 A B C D E F
n is the number of digits, between one and four

The :xxxx form is unique to Computer Automation and conforms with the representation of hexadecimal constants in our other software. Note that no count of the digits is required.

The nZxxxx form is used by some other FORTRAN systems, and is provided for consistency with those.

A hexadecimal constant, since it is a form of integer constant, occupies one word (16 bits) of storage. Thus there may be from one to four hexadecimal digits. If there are fewer than four, they are right justified. For example, :A8 is the same as 3Z0A8 or :00A8.

Chapter 6 explains the use of hexadecimal constants in DATA Statements, which is a little different. There, such constants are not necessarily assumed to be integer nor restricted to four digits.

Hollerith

As with hexadecimal, Hollerith constants are usually a special form of integer constant. (There are two exceptions, described below.) Integer quantities are usually used for representing alphanumeric characters, since there is no "alphanumeric" type of data. A Hollerith constant occupies one word (16 bits) of storage, and therefore can hold two alphanumeric characters of 8 bits each. Longer strings of characters can be written. These are called alphanumeric strings, but cannot be used as elements of expressions. (See the following section.)

A Hollerith constant is written:

nHaa

where: a is an alphanumeric character (see table C-1)
n indicates the number of characters, and must be 1 or 2

Note that the character blank is permissible in a Hollerith constant. Therefore, this is one of the few places where blanks are significant and cannot be introduced just for readability.

Note also that the alphanumeric character "0" is not the same as the binary value zero, so for example, 2H00 is not the same as :0000.

If a Hollerith constant has only one character, it is left-justified with a trailing blank. Examples of some Hollerith constants and their hexadecimal equivalents:

2HXY	=	:D8D9
1H$	=	:A4A0
2H00	=	:B0B0
1H	=	:A0A0

The two places where a Hollerith constant is not treated as an integer constant are standing alone on the right side of an equal sign (see chapter 3), and in a DATA statement (see chapter 6). In both of these cases, it is treated as an alphanumeric string.

Alphanumeric String

An alphanumeric string may be written in either of two ways:

nHs or 's'

where: s is a string of alphanumeric characters, of length ≤ 255.
n is the number of characters

An alphanumeric string is not considered to have a data type and cannot be used in the ordinary way as an element of an expression. It is simply a string of characters, which occupies consecutive words in memory at two characters per word. It can be used in the following situations, all of which are described in later chapters:

1. Standing alone on the right side of an equal sign (see chapter 3)
2. In a DATA statement (see chapter 6)
3. As an argument to a subroutine or function (see chapter 7)
4. In the list of an OUTPUT statement, 's' form only (see chapter 5)

Within a string of the form 's', the quote character (') can be represented by two consecutive quotes. For example, the characters 'it's done' can be written

```
'''IT''S DONE'''
```

As with Hollerith constants, blanks are significant within alphanumeric strings.

Examples:

```
'MISCELLANEOUS CHARACTERS'
'(A+B)/C:  '
22HACCORDING TO  HIS NEED
5HAZAM!
1H?
```

Note that the last example could be used as either an alphanumeric string or a Hollerith constant.

Boolean

Boolean operations are those in which logical operations (AND, OR, etc.) are performed on a whole word full of 1's and 0's. This is not a standard mode in FORTRAN, but can be accomplished using hexadecimal data to set up the bits and the Boolean functions (described in chapter 7) to perform the operations.

AUTOMATIC DOUBLE PRECISION

A compiler option is available to automatically convert all real (single precision) quantities and operations to double precision, so that constants, variables, and functions do not have to be changed in the source program in order to obtain more than 7 digits accuracy. The ADP option is described in chapter 9.

VARIABLES

Variables are identified by name and can change value during the program. A variable always has one of the five types, integer, real, double precision, complex, or logical, and can only assume values within the range specified for that data type. (Any type of variable may contain an alphanumeric string, though integer variables are recommended.) Unless declared otherwise by a type statement (see chapter 6), a variable is integer if it begins with I, J, K, L, M, or N, and real otherwise.

There are two kinds of variables, simple variables (also called scalar variables) and arrays. A simple variable is a single value and is referenced by its name, as illustrated in previous examples. E.g. N, ROOT, DHO, VOLUME, CAPACITY.

Arrays

An array is a set of values. It has a name and a type, just like a simple variable. Each value is identified by its position within the array. For example, the weights of ten items might be contained in an array called WEIGHT with ten positions. The first value would then be WEIGHT(1), the second WEIGHT(2), and so on to WEIGHT(10). An array may have more than one dimension. A matrix is a two-dimensional array, and its values are identified by two positions, the first within the column and the second within the row. For example, the temperatures at twelve points on a 3x4 grid could be assigned to a 3x4 array called T. Its elements would then be:

T(1,1)	T(1,2)	T(1,3)	T(1,4)
T(2,1)	T(2,2)	T(2,3)	T(2,4)
T(3,1)	T(3,2)	T(3,3)	T(3,4)

In Computer Automation FORTRAN IV an array may have any number of dimensions. Chapter 6 discusses the declaration of array dimensions, including lower and upper bounds and how arrays are stored in memory.

Array Elements

An individual element of an array is called an array element. It is identified by the name of the array followed by subscripts enclosed in parentheses and separated by commas. There must be the same number of subscripts as the array has dimensions. Thus an array element looks like:

$v(s_1, s_2, \ldots, s_n)$

where v is the name of the array
s is a subscript expression
n is the number of dimensions declared for the array

In most cases where a simple variable can appear, such as on the left side of an equal sign, an array element is equivalent.

Subscripts

A subscript may be any integer expression. (Many FORTRANs restrict subscripts to a limited form of expression.) In particular, a subscript may itself be subscripted. This allows an entry in one array to identify the position of an entry in another array, and so on. Examples of array elements, with subscripts:

```
A(3)          MM(J)        TEMP(3,1+K-2*LAST)
COORD(I,J,K,L)             THREAD(LIST(MM(J)-K)+2)    LIMIT(-1)
```

Negative or zero subscripting, as shown in the last example, requires special dimensioning with a lower bound less than 1 (see chapter 6).

FUNCTIONS

Functions are subprograms that can be referenced as elements of an expression. A function acts on one or more quantities, called its arguments, and produces a single quantity, called the function value. For example, ATAN2 is the name of a library function that computes an arctangent, given the ordinate and abscissa. ATAN2(Y,1.0) is a function reference representing a specific value, namely the arctangent of Y and 1.0.

A function reference, then, consists of the function name followed by the arguments enclosed in parentheses:

$f(a_1, a_2, \ldots, a_n)$

where: f is the name of the function
n is the number of arguments
a is an argument.

Arguments may be constants, variables, expressions, or the names of arrays or subprograms (see chapter 7).

Except for certain library functions, the IJKLMN rule applies to the type of function names. For convenience, the double precision and complex library functions (e.g. DATAN2) are automatically recognized as having a special type. The type of a function indicates the type of its resultant value.

In addition to library functions, the user can define his own functions, either in FORTRAN (with the FUNCTION statement or the statement function definition, described in chapter 7) or in assembly language (as most library functions are written). If they are to have a type other than integer or real, they will have to be explicitly declared in a type statement.

Examples of function references:

```
F(X)          SQRT(7*A+BETA)          DISTANCE(RATE,TIME)
MAX0(N+5,J**2,1000)          F(F(X))
```

CHAPTER 3

EXPRESSIONS AND ASSIGNMENTS

There are three quite different kinds of expressions: arithmetic, relational, and logical. Each is made up of operands separated by operators. The operands may be constants, variables, or function references, or they may be subexpressions. A subexpression is an expression enclosed in parentheses. In some cases, an operator can be unary and act on only one operand, rather than separating two operands (for example, "-" to indicate a negative value).

ARITHMETIC EXPRESSIONS

An arithmetic expression is made up of integer, real, double precision, and/or complex operands, combined by arithmetic operators, which are:

Operator	Meaning
+	Addition or Positive
-	Subtraction or Negative
*	Multiplication
/	Division
** or ↑	Exponentiation †

Two operators may not appear in a row. To express Y*-3, you must write Y*(-3) or -3*Y. ** is not considered two operators, but one.

Expressions can range from a single operand to long formulas of any complexity. Some examples:

```
F(X)
3.1417
X + Y
(A+B) * (A-B)
RATE(J-1)+(GAMMA+1/RATE(J)-8.72*(P+SQRT(R**2+T**2)))/DIST
```

Evaluation Hierarchy

Does the expression X+Y/Z mean (X+Y)/Z or X+(Y/Z)? In FORTRAN it means X+(Y/Z) and this is determined by the hierarchy of operators, which is:

1.	**	(highest)
2.	* and /	
3.	+ and -	(lowest)

† Complex exponentiation is only permitted to an integer power. See "Mixed Mode Expressions".

This means that the expression

```
T-U ** V * W
```

is interpreted as

```
T - ((U**V) * W)
```

Parentheses take precedence over all operators. Any subexpressions enclosed in parentheses are evaluated first and then treated as single operands.

Successive operators of the same precedence are evaluated left to right, so that J/K/M*L means ((J/K)/M)*L. This includes **, the exponentiation operator. For example,

```
2 ** 3 ** 2
```

is interpreted as

```
(2 ** 3) ** 2
```

which is the same as

```
2 ** (3 * 2)
```

and has the value 64, whereas

```
2 ** (3 ** 2)
```

would have the value 512. This is not consistent among FORTRANs, so we recommend the use of parentheses to show exactly what you mean.

Note that when the results are equivalent, the compiler may reorder operations to obtain more efficient object code. For example, E+F+G/H might be evaluated as G/H+E+F. To preserve desired groupings, use parentheses.

Mixed Mode Expressions

The type of an expression depends on the type of the operands in it. If it contains only operands of one type, then it has that type. If it contains operands of more than one type, it is called a mixed mode expression. Most mixed expressions are allowed, but some are not. Some are allowed but not recommended, because of the varying ways in which other FORTRANs treat them.

There is a precedence of types that determines the type of a mixed mode expression. It is:

1. Complex (highest)
2. Double precision
3. Real
4. Integer (lowest)

With one exception, the type of an expression is the same as the highest type appearing in it. The exception is that function arguments are independent of the expression in which the function reference appears. They have no effect on the mode of the outer expression.

Within a mixed mode expression, each operation is done in the higher mode of its two operands. In general this means that the lower type operand is converted to the higher type before being used. (In integer exponentiation, however, this is not necessary. See below.) The order in which the operands are selected depends on the precedence of the operators connecting them. (The order of evaluation is not affected by the precedence of the types of the operands.) Higher precedence operations are always done first. For example, in:

```
X + J/K
```

the division is done first, in integer, with the fractional part truncated. Then the result is converted to real and added to X. The same would be true if the expression had been written:

```
J/K + X
```

When there is a succession of operands connected by operators of equal precedence, they are grouped from the left, regardless of type. For example, in:

```
J + K + X
```

J is added to K in integer, then this is floated and added to X. All these operations can be done "on the fly", without having to store intermediate results in temps. On the other hand, if the expression had been written:

```
X + J + K
```

the J would be floated and added to X, and this result would be stored away so that K could be floated and then added to it. This can affect not only the speed but the results, so keep it in mind when writing mixed mode expressions. Usually it is a good idea to start with the lowest type operands on the left and proceed to the highest type on the right.

Parentheses have the highest precedence and can be used to control the modes in which operations get done. For example:

```
X + (J+K)
```

causes the J+K to be done in integer.

Usually each mixed mode operation requires the lower type operand to be converted first. Exponentiation to an integer power is an exception. For example:

```
X ** K
```

is done by repeated multiplication of X by itself K times, rather than by using logarithm and exponential, which would be required by:

```
X ** FLOAT(K)
```

The ordinary numeric types, integer, real, and double precision, may be mixed in any way, using all of the operators. Complex quantities may be mixed with the other three when using add, subract, multiply, or divide, but the only complex exponentiation allowed is complex to an integer power. Assuming CPX is complex:

```
3*CPX**(J+K)        is legal
CPX**2.0
A**(1.0,1.0)        are not legal
CPX**CPX
```

The latter is the only case where two operands of the same type may not be combined.

Logical quantities may not appear as operands in arithmetic expressions, since they have no numeric value.

Here are some guidelines about using mixed mode expressions:

1. Integer operations are the fastest, so to take advantage of this, all operands in an expression should be integer.

2. For maximum efficiency when operands are of various types, group the lower types together, either left to right or with parentheses. For example:

```
33*N/X*CPX
```

is more efficient than:

```
CPX*33*N/X
```

On the other hand, if 33*N is liable to overflow maximum integer size, it may be preferable to sacrifice speed and do the multiplication in floating point by writing:

```
N/X*33*CPX
```

3. Constants that need to be converted to a higher type will be converted at compile time, rather than during execution. For example:

```
3/X+10          is interpreted as          3.0/X+10.
```

This also means that constants that need to be double precision will automatically be double precision, even though they do not have a D exponent. For example, if DP is double precision:

```
.3 + DP         is equivalent to           .3D0 + DP
```

and the .3 will have the full 16 digits of accuracy.

4. When variables or function references of a lower type are used, they will have to be converted during execution, at some cost in space and time.

5. If complex and double precision quantities are mixed, the double precision ones will be converted to complex, thus losing their extra precision.

6. Be aware that other FORTRAN systems may handle mixed mode arithmetic differently, particularly in cases such as:

```
J/K + X
```

Other FORTRANs may do all operations in the highest type in the whole expression, rather than in the higher type of their two operands. Thus in the above case the division would be done in real mode, not integer. We think it best to avoid situations of this sort.

Arithmetic Overflow

Chapter 2 discussed the ranges of values for numeric quantities. If a value exceeds the proper range, one of the following actions is taken, depending on the type and the context:

1. In source programs, constants that are too large or too small are diagnosed as errors during compilation.

2. Input values read in at run time are also diagnosed if out of range.

3. Integer overflow resulting from calculations at run time is ignored. The computer automatically returns the lower 16 bits. Therefore, if you use large integer values, test them where necessary to avoid overflow.

4. Floating overflow at run time, either from arithmetic operations (add, multiply, etc.) or from mathematical functions (e.g. exponential), produces a diagnostic. In addition, the maximum possible value (of the appropriate sign) is substituted and execution continues.

5. Floating underflow at run time (magnitude too small) results in a zero value and no error message.

RELATIONAL EXPRESSIONS

A relational expression compares two arithmetic expressions and produces a logical result, i.e. true or false, according to whether the values have the relationship specified. The relational operators are:

Operator	Meaning
.LT. or <	Less than
.GT. or >	Greater than
.LE.	Less than or equal
.GE.	Greater than or equal
.EQ.	Equal
.NE.	Not equal

The relational expression has the form:

$$e_1 \; r \; e_2$$

where: e_1 and e_2 are integer, real, double precision, or complex expressions
r is one of the relational operators shown above

For example:

```
J .EQ. KEY
RADIUS**2 > 1
X+Y .LE. X*Y
```

In the first example, if J equals KEY the relational expression is true, otherwise it is false.

If the two arithmetic expressions have different types, each one is evaluated in its own type and then the one with the lower type is converted to the higher type for the comparison. If the value is a constant, the conversion is done at compile time; otherwise it must be done at run time. Thus:

```
23  .GT. X      is equivalent to      23. .GT. X
```

while

```
I/J  .GE. SQRT(G)
```

causes the division to be performed in integer and the result converted to real in order to compare with SQRT(G).

A complex value may only be compared for equal or not equal, since the others are not meaningful. It may be compared with a non-complex value, in which case, the latter acquires an imaginary part of zero.

Be careful about comparing floating point values for equality. Most values are binary approximations, so during computations inaccuracy will creep into the low order bits. This will make values that are essentially equal appear unequal. We can guarantee, however, that constants that have an exact binary representation will be exactly translated.

It is not permissible to concatenate relational operations, such as in
(A .LT. B .LT. C).

Relational expressions are a subset of logical expressions. They most often appear in logical IF statements, such as

```
IF (N < 0) GO TO 5
```

as described in the next chapter.

LOGICAL EXPRESSIONS

Logical expressions are made up of logical operands and the three logical operators:

Operator	Meaning
.AND.	True if both operands are true
.OR.	True if either or both operands are true
.NOT.	True if single operand is false, false if operand is true.

The first two are binary operators, while the third is a unary operator.

Each element of a logical expression has the value true or false, and each logical operation produces one of those values. An element of a logical expression may be:

1. A relational expression
2. A logical variable or function reference
3. A logical constant
4. Another logical expression enclosed in parentheses
5. Any of the above, preceded by .NOT.

Logical expressions most often contain relational expressions and are used in logical IF statements, such as the one shown in the preceding section. A more complicated one, using some logical operators, would be:

```
IF (A > B .AND. (J .EQ. KEY .OR. J .EQ. NEWKEY)) GO TO 23
```

This logical expression has the value true if A is greater than B and J equals either KEY or NEWKEY. This double test on J cannot be performed by writing:

```
J .EQ. (KEY .OR. NEWKEY)
```

because, first of all, KEY and NEWKEY are not logical values and so cannot be connected by .OR., and secondly, if they were logical values the subexpression (KEY .OR. NEWKEY) would have a logical value, not the integer value required by the .EQ. operator.

The only time two logical operators may appear next to each other is when the second is .NOT.. For example, assuming L is a logical variable:

```
N .EQ. 3 .AND. .NOT. L
```

Although less common than their use in IF statements, logical expressions may also have their values assigned to logical variables (with the assignment statement, described below), and these variables, as well as the constants .TRUE. and .FALSE., may then be used in logical expressions.

Evaluation Hierarchy

As with arithmetic expressions, there is a hierarchy that determines in which order logical operations will be performed. For example, the expression

```
.NOT. L1 .OR. L2 .AND. L3
```

might be interpreted as:

```
.NOT. (L1 .OR. (L2 .AND. L3))
```

or:

```
(.NOT. (L1 .OR. L2)) .AND. L3
```

or various other ways. Actually it means

```
(.NOT. L1) .OR. (L2 .AND. L3)
```

because the precedence of logical operators is:

1.	.NOT.	(highest)
2.	.AND.	
3.	.OR.	(lowest)

Parentheses may of course be used to define how operands are to be grouped. Also, logical expressions may contain relational expressions, which are evaluated first.

The relational expressions may contain arithmetic expressions which, in turn, must be evaluated first. Thus the overall hierarchy of all operations can be expressed as:

1. Parenthesized arithmetic subexpressions, from innermost out.
2. **
3. * and /
4. + and -
5. The relational operators.
6. Parenthesized logical subexpressions, from innermost out.
7. .NOT.
8. .AND.
9. .OR.

Let us apply this hierarchy to an example containing all of the above operations. Here L, P, Q, and R are logical:

```
L.OR..NOT.P.AND.(Q.OR.R).OR.A>B+C/D**(E-F)
```

At the final step, this is the OR of three operands, as shown below:

```
L .OR. ((.NOT.P).AND.(Q.OR.R)) .OR. (A>(B+(C/(D**(E-F)))))
```

ASSIGNMENT STATEMENT

The assignment statement is the most important statement in FORTRAN. It specifies most of the computations that are to be performed by a program. It is written:

v = e

where: v is a variable (simple or subscripted)
e is an expression

This computes the value of e and assigns it to v. It is not exactly an equation, since it does not declare that v is equal to e; it sets v equal to e. Thus a statement such as:

K = K - 3

is not a contradiction; it simply decreases the current value of K by 3.

Some examples:

```
X = Y
N = 3*MAX0(I,J)
MM(I) = MM(I-1) + K*2
FLAG = .TRUE.
TIME(LIME) = GOODOLD*GONEBY
E = M * C**2
```

Usually the expression has the same type as the variable. If it does not, then it is computed independently of the variable (i.e. in its own mode) and converted to the variable's type before assigning. This is called a mixed mode assignment and, as with mixed mode expressions, some cases are allowed and others are not. In particular, a logical expression can be assigned only to a logical variable. A complex value cannot be assigned to a lower numeric type (such as real), because this involves the loss of its imaginary part and, since this might happen inadvertently, a warning diagnostic is more useful here. There is a library function provided for doing complex to real conversions.

If the entire expression on the right of an equal sign consists of a single constant (of a different type), then the constant will be converted at compile time. Otherwise the conversion must be done at run time. For example:

X = 0 is equivalent to X = 0.0

A special case is made for alphanumeric string constants that appear alone to the right of an equal sign. These are considered to have no type and are simply stored into the variable regardless of its type. The string constant must not be longer than can be contained in the variable. Since character strings have two characters per word, this means the maximum size is two characters for integer and logical variables, four characters for real, and eight for double precision and complex. If the string is shorter than the maximum length, it is stored beginning at the left (first word, first byte) of the variable, and the rest of the variable is filled out with blanks.

We recommend integer variables (or arrays) for working with alphanumeric characters, for several reasons:

1. It is hard to work with the individual words of a multi-word floating point variable.

2. The arithmetic operations, such as addition and multiplication, are not meaningful in floating point, since part of the word is a mantissa and part an exponent.

3. The Boolean functions, which can be used for masking out certain characters, operate only on integer quantities.

Note that in this situation, a Hollerith constant is considered a string constant, so the statement:

```
X = 2HAB
```

is quite different from the two statements:

```
J = 2HAB
X = J
```

since in the second case J will be converted to floating point, destroying any resemblance to alphanumeric characters.

Table 3-1 shows the permissible mixtures of type in an assignment statement.

Table 3-1 Permissible types in mixed assignments

Variable Type	Expression Type					
	integer	real	double precision	complex	logical	string
integer	D	T	T	---	---	D
real	F	D	P	---	---	D
double precision	F	P	D	---	---	D
complex	F,R	R	P,R	D	---	D
logical	---	---	--	---	D	D

Abbreviations:

D Direct assignment, no conversion.

F The integer is converted to floating point of the appropriate precision.

T The floating point value is truncated to integer. Any fractional part is thrown away, which always results in a truncation towards zero. In other words, 33.6 is truncated to 33, and -98.999 is truncated to -98, not to -99. If the floating point value is too large to be expressed in integer, then it is truncated at the left end as well, with meaningless results. As in other cases of Integer overflow, no error diagnostic is generated.

P Increase or decrease the precision. Conversion from double precision to real is not rounded, but truncated.

R The value of the expression becomes the real part; the imaginary part is zero.

CHAPTER 4

CONTROL STATEMENTS

FORTRAN statements are normally executed in the order written, one after another. Control statements are used to change this order by transferring control to some point other than the following statement.

STATEMENT LABELS

Statement labels (also called statement numbers) are used to identify statements so that control can be transferred to them from elsewhere. A label is a decimal integer of up to five digits (i.e. from 1 to 99999). As shown in chapter 1, the label appears in the first five columns of the source line, which is called the label field, As with integer constants, blanks and leading zeros are ignored.

Although a statement label is a number, its value has no significance and implies no ordering. It is simply an identifying label. Two statements may not have the same label.

Most of the control statements reference labels to identify a transfer point. READ and WRITE statements also reference the labels of FORMAT statements, although this does not involve any actual transfer.

GO TO STATEMENTS

Unconditional GO TO Statement

The GO TO statement transfers control to another statement. It has the form:

```
GO TO k
```

where: k is a statement label

For example:

```
   GO TO 51
17 N = -N
51 OUTPUT N
```

The statement labeled 17 would be skipped.

Computed GO TO Statement

The computed GO TO transfers control to one of several places depending on the value of a variable. It is written:

GO TO (k_1 , k_2 , ..., k_n), v

where: k_i is a statement label
v is a simple (unsubscripted) integer variable whose value is between 1 and n.

The comma before v is optional and may be omitted.

If the value of v is j, then the GO TO transfers to label k_j . If j is less than 1 or greater than n, this is diagnosed as an error at run time.

Example:

```
GO TO (14,3,999,80), KEY
```

If KEY=1, the transfer is to statement number 14, if KEY=2 to statement number 3, and so on.

Assigned GO TO Statement

The assigned GO TO also enables transfer to various labels, but without having to know what those labels may be. Instead of specifying any statement numbers, this statement specifies a variable, which is expected to contain the location of some statement label. Elsewhere in the program, an ASSIGN statement (see below) is used to assign the desired label to the variable. The assigned GO TO has the form:

```
GO TO v
```

where: v is a simple integer variable that has previously been assigned a label using the ASSIGN statement.

This feature can be used to make subroutines out of sections of the program, rather than making each section a separate program and using CALL and RETURN (which are described below). A section could end with the statement

```
GO TO JUMP BACK
```

Before transferring to this section, then, the desired return point would be assigned to the variable JUMP BACK.

Other FORTRANs, including the ANSI standard, require that all of the possible destination labels be listed in the assigned GO TO statement, as shown below. Computer Automation FORTRAN IV accepts this form, but does not require it. Example:

```
GO TO M, (23,9,2)
```

Here 23, 9, and 2 are the only labels that may legally have been assigned to M. The comma following the variable is optional. While a diagnostic will be generated at compile time if an illegal label is specified, no testing will be performed on the value assigned to M at run-time.

ASSIGN STATEMENT

The ASSIGN statement is used to assign a statement label to a variable, and has the form:

```
ASSIGN k TO v
```

where:
- k is a statement label
- v is a simple integer variable

For example, the "subroutine" described in the previous section ended with an assigned GO TO via the variable JUMP BACK. Before transferring there, you would use an ASSIGN, such as:

```
ASSIGN 47 TO JUMP BACK
```

The assigned GO TO would then transfer to statement label 47.

The label that is assigned must lie in the same program as the assigned GO TO. It is not permissible, for example, to assign a variable in one program, allocate the variable in COMMON storage, and then transfer to it from another program.

Also keep in mind that assigning a label is quite different from assigning a value with an assignment statement. The statement

```
NUEVE = 9
```

is not equivalent to

```
ASSIGN 9 TO NUEVE
```

since the 9 in the former case is not a label but a value. Attempting an assigned GO TO on such a variable would be meaningless and disastrous.

Conversely, it is also not meaningful to do arithmetic on a variable that has been assigned by an ASSIGN statement. For example, the statements:

```
ASSIGN 8691 TO NMR
NMR = NMR + 4
```

would cause the value of NMR to be unpredictable.

IF STATEMENTS

Logical IF Statement

The logical IF statement tests the truth of a logical expression to determine whether or not to execute another statement. If that other statement is a GO TO, this acts as a conditional transfer. The logical IF is written:

```
IF (e) s
```

where: e is a logical expression.
s is any executable statement other than a DO or another logical IF.

If e is true, the statement s is executed; otherwise it is skipped. In either case, the next statement executed is the one following the IF, unless statement s causes a transfer elsewhere. Often the expression e is a relational expression or several relational expressions combined by .AND. and .OR.. Logical variables, constants, and function references may appear too.

Examples:

```
IF (A>B)  OUTPUT 'A TOO LARGE:',A
```

If A is greater than B (i.e. the relational expression A > B is true), the program outputs a message and the value of A; otherwise it does not. In the following example, ERROR is a logical variable:

```
IF (ERROR  .OR.  N  .EQ.  10) GO TO 31
```

If ERROR was previously set true or if N equals 10, the GO TO statement is executed and control does not fall through to the succeeding statement.

A logical IF cannot control more than one statement. To achieve this effect, you have to reverse the test and jump around the several statements, as shown here:

```
      IF (TEMP .LE. 99) GO TO 7
      FEVER = .TRUE.
      OUTPUT TEMP, NAME
    7 CG = .55555*(TEMP-32)
```

Arithmetic IF Statement

An arithmetic IF statement always transfers control to one of three labels, depending on whether the value of an arithmetic expression is negative, zero, or positive. It has the form:

IF (e) k_{neg} , k_{zero}, k_{pos}

where:

e	is an integer, real, or double precision expression.
k_{neg}	is the statement label transferred to if e is negative.
k_{zero}	is the label transferred to if e is zero.
k_{pos}	is the label transferred to if e is positive.

For example:

```
      IF (N) 99,2,7
      IF (SIN(THETA)*VEL) 1000,2000,3000
      IF (ALPHA) 6,10,6
```

In the last case, the negative and positive labels are the same, so this statement is equivalent to:

```
      IF (ALPHA .EQ. 0) GO TO 10
      GO TO 6
```

There are several points to keep in mind when deciding whether to use an arithmetic IF or a logical IF. The arithmetic IF provides a three-way test. However, if only a two-way test is needed, the logical IF is probably more readable. If you need to compare two integer quantities, there is another consideration. You could write:

```
      IF (J<K) GO TO 5
```

or:

```
      IF (J-K) 5,6,6
    6 next statement
```

In the latter case, however, if J is very large positive and K is very large negative, the difference (J-K) may be too large to represent and will overflow and cause an incorrect test. The relational operator (<) always gives the correct answer, but generates slightly more object code.

The section on relational expressions in the previous chapter cautioned against testing for equality of floating point values. The same thing applies to the zero test in an arithmetic IF statement, especially if the expression involves a subtraction.

DO STATEMENT

The DO statement is used to control repetitive execution of a group of statements. For example, if you wanted to set to zero all elements of an array of size 50, you could write:

```
      J = 1
    4 A(J) = 0
      J = J + 1
      IF (J  .LE.  50) GO TO 4
```

The same thing can be done in two statements using DO:

```
      DO 2 J = 1 , 50
    2 A(J) = 0
```

This says, "Do the following statements, up to and including statement number 2, first with J equal to 1, then with J equal to 2, then 3, and so on up to J=50".

The general form of the DO statement is either of the following:

DO k v=m_1, m_2
DO k v=m_1,m_2,m_3

where:
- k is the label of the statement that is to end the loop.
- v is a simple integer variable.
- m_1, m_2, and m_3 are the DO parameters and must each be either a simple integer variable or an integer constant (signed or unsigned).

A comma may optionally be used to separate k and v.

This statement causes the following actions:

1. Set the variable v equal to m_1. v is called the DO control variable or the DO index. m_1 is called the initial value.

2. Execute the statements following the DO, up to and including the statement with label k. These statements constitute the range of a DO loop (see also below).

3. Increment v by m_3. m_3 is called the increment and must be greater than zero. If it is not specified, it automatically has the value 1.

4. Test whether v is now greater than m_2, which is called the limit. If it is, the DO loop is finished. Proceed to the statement following statement k. If v is still less than or equal to m_2, go back to the statement immediately following the DO statement and execute the loop again.

You can see that a DO loop will always be executed at least once, even though the initial value is greater than the limit. For example, the DO loop:

```
DO 44 NR = 10,5
```

will be executed exactly once. It is a good idea to avoid writing DO statements like this because some FORTRANs choose to execute intitially satisfied loops no times instead of one time.

DO loops are not allowed to run downwards instead of upwards; that is, the increment, m_3, may not be negative.

Note that it is not necessary for the control variable to hit the limit exactly. It can jump over the limit and the loop will terminate as soon as it does so. For example, the loop:

```
DO 3, NAMA = -20, 0, 6
```

will execute four times, with NAMA equal to -20, -14, -8, and -2. When NAMA reaches +4, the loop will not be executed again.

In order to produce more efficient object code, there is one restriction on the parameters of a DO. In step 4 above, the control variable is compared with the limit using a subtract operation, which means that they must not differ by more than 32767. To put it another way:

$$m_2 - (m_1 + m_3) \leq 32767$$

DO Loop Ranges

The statements executed as part of a DO loop (up to and including the terminal statement) are called the range of the DO loop. There are some rules regarding what you may and may not do within this range.

The terminal statement (the one with label k) may be any executable statement except:

GO TO (of any form)
Arithmetic IF
DO
RETURN
STOP

If the terminal statement were a transfer, then control could never reach the loop testing code. If it were a DO, then two loops would be incorrectly nested (see below). However, a DO loop may end on a logical IF, even when it contains any of the above (except DO), because then there is a way to reach the loop testing code.

If it works out that the last statement in a DO loop needs to be a transfer that is not allowed, there is a dummy statement called CONTINUE that can be used instead as the actual termination. For example:

Instead of	You can write
DO 5 I = 0,N	DO 5 I = 0,N
:	:
5 IF (VECT(I)) 4,6,6	IF (VECT(I)) 4,5,5
	5 CONTINUE

This provides an avenue for control to get to the loop testing code. The same thing could be accomplished by writing:

```
    DO 5 I = 0,N
    :
5   IF (VECT(I)<0) GO TO 4
```

Within the range of a DO loop, you must not alter the value of the control variable (v) or any of the parameters (m_1, m_2, m_3). The DO statement needs to have complete control over these; otherwise unpredictable actions may occur. On the other hand, it is perfectly acceptable to use these values, as long as they are not changed. The DO control variable is particularly useful, either as a subscript (to step through various elements of an array) or as a counter. For example, you could set each element of a one hundred position array equal to the value of its position using the following loop:

```
    DO 13 I = 1 , 100
13  MM(I) = I
```

It is also not permissible to jump into the range of a DO loop. The DO statement does some special set-up for the loop that cannot be skipped over. On the other hand, it is permissible to jump out of a DO loop before it has completed. For example, if a special situation occurs during a DO loop that makes it unnecessary to do the rest of the loop, you can transfer out instead of falling through the bottom of the loop. In this case, the DO control variable will have the proper value, namely the one it had at the time the transfer was made. This is not true on normal completion of a DO loop. If the loop terminates normally, the value of the control variable becomes undefined and should not be depended on. For example, in the loop:

```
      DO 4 N=1,10
       .
       .
       .
      IF (A(N)< 0) GO TO 20
       .
       .
       .
    4 A(N) =A(N) + MAX/3
   10 OUTPUT N
       .
       .
       .
   20 OUTPUT N
```

the value of N at statement 20 would be somewhere between 1 and 10. The value of N at statement 10 is unpredictable.

There is one exception to the rule that you cannot jump into a DO loop. If you first jump out, and you make no changes to the DO index or parameters, you can jump back in again and continue on with the loop. The part outside the loop is called the extended range of the loop and is allowed by ANSI standard FORTRAN, but we do not particularly recommend it. In most cases, the extended range can as easily be included within the loop instead of outside. This is usually less confusing and may produce more efficient object code.

It is permissible (and useful) for one DO Loop to lie within the range of another. These are called nested loops. Nesting may extend to any level (like a group of smaller and smaller boxes each inside the previous one) as long as each loop lies entirely within the next outer one. That is, the ranges may not overlap. The following loops are illegally nested:

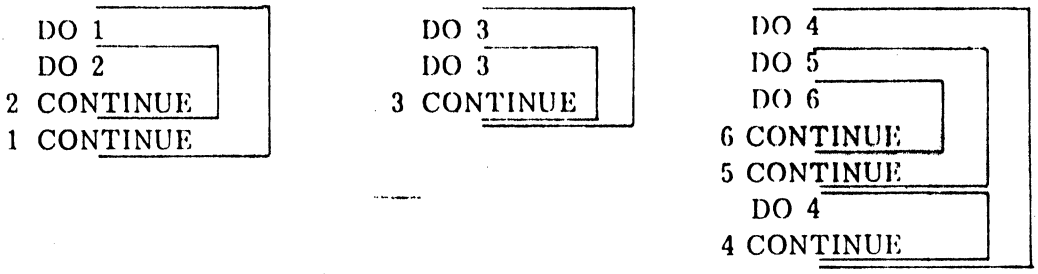
```
    DO 1
    DO 2
  1 CONTINUE
  2 CONTINUE
```

```
    DO 3
  3 DO 4
  4 CONTINUE
```

```
    DO 5
    DO 6
    DO 7
  6 CONTINUE
  7 CONTINUE
  5 CONTINUE
```

In the third example, loops 6 and 7 correctly lie within loop 5, but loop 7 does not lie within loop 6.

The following loops are correctly nested:



The second and third examples illustrate that two or more nested loops may terminate on the same statement. When that happens, the compiler generates the various sets of loop testing code in the proper order (inversely to the order that the DOs appeared).

The rules about jumping into and out of DO loop ranges apply in the same way when the loops are nested. You cannot jump from an outer loop into an inner loop. If more than one loop ends on the same statement, only the inner one can jump to that statement.

The following example shows nested DOs used to multiply a 3x8 matrix (A) by an 8x5 matrix (B), producing a 3x5 matrix (C). (The DIMENSION statement will be described in chapter 6.)

```
      DIMENSION A(3,8) , B(8,5) , C(3,5)
      DO 2 J = 1 , 5
      DO 2 I = 1 , 3
      C(I,J) = 0
      DO 2 K = 1 , 8
2     C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

CONTINUE STATEMENT

This is a "do nothing" statement that only serves as a place to put a statement label for the termination of a DO, when the DO would otherwise end on a transfer. The statement is written:

```
      CONTINUE
```

The discussion of DO ranges in the previous section contains an example of the use of CONTINUE.

CALL STATEMENT

This statement transfers control to a subroutine. The subroutine is a separate program that may either be written in FORTRAN (see the SUBROUTINE statement in chapter 7) or in assembly language. The CALL may or may not pass arguments to the subroutine, depending on the form used:

```
CALL sub
```

or:

```
CALL sub(a1 , a2 , ... , an)
```

where: sub is the name of the subroutine.

a_i is an argument, which may be a constant, variable, expression, or the name of an array or another subprogram. Arguments are discussed in greater detail in chapter 7.

A subroutine differs from a function in two ways. First, it may be called with no arguments, while a function may not. Second, it does not return a value through its name and so may not be used in an expression. In fact, a subroutine has no data type associated with it. It is simply the name of a block of instructions to be executed.

A subroutine can return values in a sense by storing them in its arguments. Arguments to a subroutine may be either input arguments or output arguments (or both), depending on what the subroutine does with them. For example, a subroutine to compute the roots of a quadratic equation might be called with:

```
CALL QUAD(A,B,C,R1,R2)
```

where A, B, and C are set up before the CALL. The subroutine, QUAD, uses these arguments to compute R1 and R2, and the calling program can then use R1 and R2. Arguments should be modified in this way only if they are variables or arrays. An example of an argument used for both input and output might be:

```
CALL ROUND (X,4)
```

which could round X to four digits and store the new value back into X.

Examples of CALL statements:

```
CALL AVERAGE (X-Y,A(I)**2,13.7)
CALL ERROR
CALL OUTPUT(ALPHA , N.GE.0)
CALL HOME(GR4-6633,'I PI MOD E')
```

The last example shows the use of a long alphanumeric string as an argument. This is described further in chapter 7.

RETURN STATEMENT

CALL is used to transfer to a subroutine. RETURN is used to get back. It can "get back" from either a subroutine or a function written in FORTRAN. When it returns from a subroutine to a CALL, it goes to the statement immediately following the CALL. When it returns from a function to a function reference, it goes back to the point of reference in an expression and supplies the function value, so that the rest of the expression can be evaluated. RETURN is written as simply:

```
RETURN
```

In any FORTRAN subprogram (SUBROUTINE or FUNCTION), a RETURN statement must be the last statement executed. It does not have to be physically the last statement in the program. There may be several RETURNs in a program, each having the same effect.

PAUSE STATEMENT

PAUSE is used to temporarily suspend execution, usually to allow the computer operator to perform some specified action (such as mounting a tape or deciding whether to continue). The operator can then signal the program to continue execution, beginning with the statement immediately after the PAUSE.

A PAUSE statement types out "PAUSE" to the computer operator and will also display a number to him:

```
PAUSE          (equivalent to PAUSE 0)
PAUSE n
```

where: n is an unsigned decimal integer.

PAUSE may not be a meaningful operation in a real time environment, especially if there is no computer operator and/or no display device.

STOP STATEMENT

STOP is written in the same two formats as PAUSE, namely:

```
STOP          (equivalent to STOP 0)
STOP n
```

where: n is an unsigned decimal integer.

STOP terminates execution of a program and causes control to be returned to OS or RTX. It is usually the last statement executed in a main program. If it appears in a subprogram control is not returned to the calling program. The integer value will be output (if possible) before termination.

END STATEMENT

END must be the physically last statement in each program. It is not an executable statement (such as STOP), but simply terminates compilation. However, if no STOP or RETURN has been encountered, END will have the same effect as STOP in a main program or RETURN in a subprogram. An END may be labeled.

The END statement introduces one restriction on the use of continuation lines. Ordinarily statements may be broken at any point and continued on the next line. However, once the compiler has found "END", it will not read another line to look for continuation. Thus the statement:

```
END           =           1.0
```

will properly be recognized as an assignment statement, while:

```
    END
2             =           1.0
```

will not.

CHAPTER 5

INPUT/OUTPUT

INPUT/OUTPUT LISTS

There are several forms of input/output statements. All of them make use of an input/output list to specify the items to be processed. In an output statement these items have their values output, while in an input statement these items have new values read into them.

Simple Lists

A simple list is composed of scalar variables, array elements, and array names, separated by commas. Parentheses may also be used to enclose groups of items if you desire; this has no effect. (On DO-implied lists, below, parentheses are mandatory.) The OUTPUT statement, described in the next section, also permits constants to appear in the list.

Examples of input/output lists:

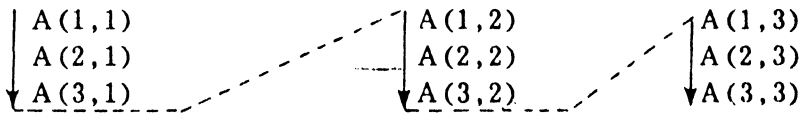
```
X
J, MAX, MATRIX(3,1,2)
ALPHA, B(J), MATRIX, (RATE,TIME)
Z(1), Z(2), Z(3)
```

When an unsubscripted array name appears, it represents all of the elements in the array, one after another. The elements are taken in the same order that they are stored in memory. (This is discussed in chapter 6.) Suppose that MATRIX is a 3x2x2 array. Its elements would be processed in the following order:

```
MATRIX(1,1,1)
MATRIX(2,1,1)
MATRIX(3,1,1)
MATRIX(1,2,1)
MATRIX(2,2,1)
MATRIX(3,2,1)
MATRIX(1,1,2)
MATRIX(2,1,2)
MATRIX(3,1,2)
MATRIX(1,2,2)
MATRIX(2,2,2)
MATRIX(3,2,2)
```

Note that it starts with the lowest value for each subscript and ends with the highest value for each subscript. In between, the first subscript varies most rapidly and the last subscript varies least rapidly. This is sometimes called "columnwise"

ordering, because in a two-dimensional array, the elements are taken by columns rather than by rows. For example:



To process array elements in a different order, or to process only part of the array, you have to specify the various elements, either individually or using DO control, as described below.

DO Controlled Lists

A DO controlled list is a simple list followed by a comma and then by a DO control, with all of this enclosed in parentheses. An I/O list DO control looks exactly like the part of a DO statement that follows "DO k", namely:

$$v = m_1 , m_2 , m_3$$

where: v is a simple integer variable.
m_1 , m_2 , m_3 are the DO parameters, each either a simple integer variable or an integer constant.

The "range" of an I/O DO control consists of the items in the simple list preceding the DO control. The meaning is then essentially the same as in the DO statement: "Process these items over and over, first with v equal to m_1 and then incrementing v by m_3 until it exceeds m_2 ." If m_3 is not present, it is automatically 1.

For example, the first five elements of the array X could be specified by writing:

```
(X(I), I=1,5)
```

This has exactly the same effect as writing:

```
X(1),X(2),X(3),X(4),X(5)
```

A DO controlled list, enclosed in parentheses, becomes a simple list item and can be intermixed with other items as if it were a variable or array name. In particular, this means that one DO controlled list can contain another one (as one of its simple list items), and this nesting can be continued to any level. This makes it possible to step individually through each subscript range of a multi-dimensional array.

For example, in the preceding section we showed how the 3x3 array A was processed columnwise when it appeared unsubscripted. You could do the same thing by writing:

```
((A(J,K), J=1,3), K=1,3)
```

The inner loop is on J, causing the first subscript to vary most rapidly. Conversely, if you wanted to print out the array by rows, you could write:

```
((A(J,K), K=1,3), J=1,3)
```

The DO control variable (as well as the other parameters) are available (materialized) within the list and may be used as list items, but only on output. Inputting into them would change their values within a loop, which is not allowed. For example, the following list could print out the values in two 100-position arrays, with each two values preceded by the position count:

```
(J, X(J), Y(J), J=1,100)
```

The DO index might even appear only as a list item. Suppose you wanted to print the odd numbers from 1 to 25 as headings to a table. You could do this by writing:

```
(N, N=1,25,2)
```

This assumes, of course, the use of a WRITE statement and the proper formatting, which will be described in subsequent sections.

On input, every list item assumes its new value as soon as it has been processed, so it can be used right away such as for a subscript. For example:

```
J, ALPHA(J), K, BETA(J,K), M1, M2, (X(I), I=M1,M2)
```

FREE FORM INPUT/OUTPUT

The standard form of input/output in FORTRAN involves the use of the READ and WRITE statements and the FORMAT statement. The FORMAT statement is very ingenious and can perform a lot of fancy editing, such as numbers in certain columns, decimal points in certain positions, headings that line up, preceding dollar signs, etc. On the other hand, its features, and the way they interface with the I/O list, are rather complicated to learn, even in simple cases.

Many programs need only to get some values in and some answers out, in an understandable way. The free form input/output statements in Computer Automation FORTRAN IV serve this purpose. The OUTPUT statement prints out values of any type in an appropriate format. It also prints character strings for identification. The INPUT statement likewise can read in all types of data, in essentially the same variety of forms as can be used for constants in a source program.

OUTPUT Statement

The OUTPUT statement is written:

OUTPUT list

The list is as described in the previous section. In addition, constants may appear in the list (see below).

Integer values and real values are printed with a decimal point and six significant digits (also an exponent if larger than 999999. or smaller than .1). Double precision is the same except with 16 significant digits, which also means that an exponent is not needed until the value reaches 10^{17}. Complex is output as two reals. Logical produces either a T or an F. Values are separated by commas, not only for readability but also for symmetry with the INPUT statement (below), which requires a comma separator.

Values are printed across a line until there is not room for the next value. Then a new line is begun. Line width is dependant upon the listing device.

For example, the statement:

```
OUTPUT K, X(K), ALPHA, AVOGADROS NUMBER
```

might produce the line:

```
12.  ,    23.7141,      .427000,      .602470E 24
```

Here is another example, assuming the types shown:

```
LOGICAL L
DOUBLE PRECISION D
COMPLEX C
OUTPUT L, D, C
```

This might produce:

```
T,    40000000.00018375,      0.00000,      1.00000
```

As described in the previous sections, an array name appearing without subscripts represents all the elements of the array in storage order. DO control may also be used on OUTPUT lists. For example:

```
OUTPUT (K, A(K), K=3,30,3)
```

In either case, array elements are output simply as a sequence of values (with as many on each line as will fit), regardless of the array's dimensions.

You may also use a constant in the list of an OUTPUT statement. This is provided mainly to allow for alphanumeric string constants, although other kinds of constants are acceptable too (except for a signed complex constant).†

The string constant enables you to print messages indicating what is going on or identifying other numeric values. For example:

```
OUTPUT 'ANALYSIS DONE', 'AVERAGE =',AV,'COEFS =',A,B,C
```

Each alphanumeric constant always begins on a new line, so this might produce the following lines:

```
ANALYSIS DONE
AVERAGE =   4.53700
COEFS =   2.00000,    -140.000,    0.00000
```

Blank lines can be introduced by using strings consisting only of blanks:

```
OUTPUT 'BEGIN TABLE OF RATES', ' ', ' ', RATE1, RATE2
```

Note that the second blank string was needed to begin output on a different line from the first blank string.

All output begins in column 2 of the output line, in order to avoid any vertical carriage control. (See "Carriage Control for Printing", later in this chapter.)

The OUTPUT statement always produces its output on the "standard output" device, which is arbitrarily assigned the unit number 6. (See "Unit Assignments", below.) Ordinarily you need not be concerned about this. Each installation will have the standard input and output units assigned to some particular devices, such as a card reader and printer or both to a typewriter. If you want your output on some other device, however, you have to reassign unit 6 at run time.

Another thing that you need not be concerned about, but which may be of some interest, is the formats used by the OUTPUT statement. These are shown below and will be described in detail in a subsequent section ("FORMAT Statement").

Integer	G16.8,','
Real	G16.8,','
Double Precision	G33.16,','
Complex	2(G16.8,',')
Logical	L16,','

† Parentheses are always assumed to enclose a sub-list of items. A complex constant will thus be correctly processed as two real constants, but a sign before the left parenthesis is not allowed.

The following comma is output on all but the last value on a line. These formats are all multiples of seven columns in width, so that numbers will tend to line up.

INPUT Statement

The INPUT statement has essentially the same form as the OUTPUT statement,

```
INPUT list
```

except that the list may not contain constants. (It does not make sense to read in a new value for a constant.)

The INPUT statement reads as many values as there are items in the list. There may be any number of input values on a line, separated by commas. If there is no value on a line (i.e. it is blank), this is assumed to be a value of zero.

The processing of input values by the INPUT statement is more like that of constants in a source program than it is like that of other formatted input (i.e. the READ statement). For one thing, there is no fixed width for the values. They may be as long as desired, terminating on comma or end-of-line. Also, blanks are not significant; they are ignored. In other formatted input, embedded and trailing blanks are usually treated as zeros. To avoid confusion, however, we recommend that you avoid embedding blanks in input values. The line

```
2    3    4
```

may look like three values, but it is only one. Preferably, this one value should be written:

```
234
```

while three values should be written:

```
2, 3, 4
```

As an example, the statement:

```
INPUT A, B, L, M, X(L,M), R(5)
```

might read just the following line:

```
714.6, -31, 4,6, 0, 3E-7
```

or it might read these four lines with the same effect:

```
.7146E3
      -31.0000,    4,    6.0
[blank]
.0000003
```

Note that the third line is blank, which indicates a value of zero. In general, a new field begins at the start of each new line and at each comma. If no value is found between there and the next comma or end-of-line, a zero value is assumed. Thus the values above could also be represented as:

```
714.6, -31.0, 4
6, , 3E-7
```

or as:

```
714.600,  -31 , 4. , 6 ,
.0000003
```

This latter example illustrates the fact that you should not write a comma after the last value on a line unless you intend a value of zero to follow it.

You can also see from the above examples that numeric values can be expressed in a variety of ways. Regardless of the type of the variable being input into, the input value can be an integer or have a decimal point or an exponent (either E or D). If necessary, the resulting value will be converted to the type of the variable. For example, if a number with a decimal point is read into an integer variable, the fractional part will be thrown away.

Complex values must be read in as two real values. Logical values may be any string containing a T or F. The string is terminated by a comma or end-of-line. If neither a T or F has been found, F (false) is assumed. Thus the first three values below are true, the remaining four false:

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

Unsubscripted arrays may be used. There must then be enough values read in to fill the array. Similarly DO controlled lists are also acceptable. For example, if V and W are both ten-element arrays, the statement:

```
INPUT V, (W(J), J=1,5)
```

would expect to find ten values for V and five for W(1) through W(5).

If both the standard input and output are assigned to a typewriter console, then the following statements could be used in a conversational manner to input values and output results:

```
3 OUTPUT 'ENTER BASE AND EXPONENT'
  INPUT X, N
  Y = X ** N
  OUTPUT 'X ** N = ' , Y , ' '
  GO TO 3
```

This might result in the following:

```
(typed out)  ENTER BASE AND EXPONENT
(typed in)   4.7, 2
(typed out)  X ** N =    22.0900
(typed out)
(typed out)  ENTER BASE AND EXPONENT
(typed in)   62, 8
(typed out)  X ** N =    .218340E 15
       etc.
```

The INPUT statement always reads from the standard input device, which is unit number 5. Like the standard output unit (6), this is associated at each installation with a particular device, but can be reassigned at run time.

UNIT ASSIGNMENTS

When you want to perform an input/output operation, it is necessary to specify what device the operation is to be performed on. With the free form I/O statements just described, this is handled automatically. INPUT always reads from unit 5, OUTPUT always writes on unit 6. For all other I/O statements, you must specify a unit number, which is an integer value from 1 to 99. Then, when your program is loaded, the unit numbers you have used must be assigned to particular devices. Of course, you can use units 5 and 6 (on READ and WRITE statements, for example) and these are automatically assigned to the standard input and output devices respectively. Each installation can determine which devices are to be designated as the standard (or default) input and output units.

In any input/output statement (READ,WRITE,REWIND,BACKSPACE, or END FILE) the unit number is specified by either an integer constant or a simple integer variable.

FORMATTED (ASCII) READ AND WRITE STATEMENTS

The formatted READ and WRITE statements deal with ASCII records (as opposed to binary records). They always operate in conjunction with a FORMAT statement, which controls the editing applied to the input or output. This editing may include decimal or hexadecimal conversion, selecting certain columns for the data to appear in, positioning of decimal points, processing of alphanumeric strings, and determining exactly how many records will be read or written. This allows, but also requires, a degree of control over external formats that the INPUT and OUTPUT statements do not have.

The READ and WRITE statements have the following form:†

READ(u,f) list and WRITE(u,f) list

where: u is a unit number, represented by either an integer constant or a simple integer variable.

f is a FORMAT reference. Usually it is the label of a FORMAT statement. It may also be the name of an array in which a FORMAT is stored (see "FORMATs stored in Arrays", later in this chapter).

list is an input/output list, as described in the previous section, "Input/Output Lists".

A READ always causes at least one record to be read from the specified unit. The data read is converted into values which are stored in the items in the list, in order. The conversion is controlled by the FORMAT statement, which is described in subsequent sections. Here is a simple example:

```
    READ(1,7) Y, K
7 FORMAT( F12.3 / I6 )
```

Y is the first variable to be read and F12.3 is the first format specification. This specification says that the value to be read lies in the first 12 columns, with a default decimal point 3 columns from the right end (i.e. between columns 9 and 10). The / means read a new record, ignore the rest of what is on the current record. K is then the next list item and I6 is the next format. This says that the value occupies six columns. If the two records had the following data on them:

```
bbbb-6789012
bb34567
```

Y would be set to -6789.012 and K would become 3456 (since only the first six columns are considered).

---

† See also "END= and ERR= Options", later in this chapter.

A WRITE always writes at least one record on the specified unit. It takes the values in the list and converts them into character strings to be written out, under control of the FORMAT. For example:

```
      WRITE(6,23) Y, K
   23 FORMAT (4X , F8.1 , ' VOLTS ON TEST NR', I5)
```

The first format specification is 4X, which says to skip the first four columns. The next is F8.1, which is used with the value Y. It says that the value must lie in the next 8 columns and have 1 digit after the decimal point. The next format is an alphanumeric string, which operates without any list item, as the 4X did. It causes those characters to be printed in the succeeding columns. Then the I5 causes K to be output in a 5 column field, right-justified. Suppose that Y and K had the values read into them above (-6789.012 and 3456). This WRITE and FORMAT combination would produce the following record:

```
bbbbb-6789.0bVOLTSbONbTESTbNRb3456
```

The sections on the FORMAT statement describe the various things it can do and how it interfaces with the I/O list in more complicated examples.

The FORMAT statement determines the number of records processed, except that it cannot suppress the processing of at least one record. In particular, you cannot read the same record twice or use two WRITE statements to produce information on one record. However, the same effect can be obtained using the DECODE and ENCODE statements, described later in this chapter.

An ASCII record has a maximum size of 132 characters. On some media (cards for example) the size is smaller. Keep this in mind because the READ and WRITE statements do not automatically begin a new record when the old one is full. They only begin a new record when the FORMAT tells them to. If you try to write too many characters on a record, the excess ones will be lost. If you try to read too many characters from a record, the extra ones will be assumed blank.

On some devices a zero-character record is meaningful. For example, an input line from a typewriter might consist only of a carriage return (which is treated as an end-of-line, not as part of the record). This would be equivalent to a whole line of blanks.

Since some format specifications operate without list items, it is possible to have a READ or WRITE statement without a list of variables. For example, the following statements would print four blank lines and then one saying "END":

```
      WRITE(3,9)
    9 FORMAT(////' END')
```

When records are output to a print device, column 1 is reserved for carriage control and will not be printed. See "Carriage Control for Printing", later in this chapter.

UNFORMATTED (BINARY) READ AND WRITE STATEMENTS

The unformatted READ and WRITE statements are not used by a program for communication with the outside world. They are used only to provide intermediate storage on external devices, particularly magnetic tapes. They have the form:

READ(u) list and WRITE(u) list

where: u is a unit number.
list is an input/output list.

These statements process the list items in binary, using as many bits as the type of each variable requires (16 bits for integer, 32 for real, etc.). Each READ or WRITE statement processes exactly one "logical" record. That is, the entire string of bits is considered a discrete entity, called a logical record, even though, in fact, it may have to be broken up into a number of physical records on the external medium. Each logical record includes a count indicating its size. The size is determined by the WRITE statement that produces it. A READ statement may subsequently read less data from a record than it contains, but not more; this is an error. If less than the full record is read, there is no way to get at the remainder. Thus there is a one-to-one relationship between binary READs and WRITEs. This is particularly true because the control words and record format are unique to a particular FORTRAN system. These statements are not intended to create information for, or deal with information from, other computer systems.

Normally there should be a list specified on a binary READ/WRITE. A READ with no list would just skip a record. A WRITE with no list is not very meaningful. A null record would be produced, which could only be re-read by a READ statement without a list. Examples of unformatted READ and WRITE statements:

```
READ(7) (X(J), J=1,200)
WRITE(NU) MATRIX
WRITE(3) AA, BB, (CC(J,3), J=200,500)
READ(K) GRID, COEFFICIENTS
```

END= AND ERR= OPTIONS

These options are available on both the formatted and binary READ/WRITE statements to allow you to process multiple files (on READ) and to deal with I/O transmission errors (on both READ and WRITE). They have the following forms:

END=k_1 and ERR=k_2

where: k is the label of a statement to transfer control to if an end-of-file or error is encountered, respectively.

Either or both of these options can appear in a READ/WRITE statement, in either order, in the position shown below:

```
READ(unit,format,options) list
READ(unit,options) list
```

Examples:

```
READ(5,77,END=3) X, Y, Z
```

If this READ statement encounters an end-of-file, control is transferred immediately to statement number 3, without processing the rest of the input list.

```
WRITE(6, ERR=99) MATRIX
```

If an unrecoverable hardware error occurs while trying to write out the contents of MATRIX, processing of the list stops and control is transferred to statement number 99. There is no way of telling how far through the list the statement got before the error.

```
READ(1,100,END=20,ERR=30) L, M, N
```

An end-of-file transfers control to statement 20, an error to statement 30. It is not possible for both to occur at the same time, because an error will be noticed before an end-of-file can be recognized.

If no END= is specified, and an end-of-file is nonetheless read, an error message will be printed and the program will terminate.

INTERNAL DATA CONVERSION

Sometimes it is useful to be able to perform the data conversions that the FORMAT statement does, without actually reading or writing any records. For example, suppose you want to have input cards on which the first value determines how the rest of the card should be processed. It might specify whether the remaining fields should be read as alphanumeric or numeric, such as in the following:

```
1 ABCD
2  462   17
1 WXYZ
```

Here a card beginning "1" has two 2-character alphanumeric fields, while a card beginning "2" has two integer fields, each four columns in width. It is not possible

to read and distinguish both kinds of records using the normal READ statement, since the FORMAT statement has to be specified in advance and cannot be changed partway through the record. Nor is it possible to read the same record twice (unless it is on something like magnetic tape and you backspace and read again).

The DECODE statement handles this kind of operation. It does the FORMAT conversion without the READ. In fact, a formatted READ can be thought of as a two part operation, the input of a record into a buffer and a DECODE on the buffer. Likewise, a formatted WRITE is basically an ENCODE into a buffer and the writing out of the buffer. (This writing out is not the same as an unformatted WRITE.) With a DECODE or ENCODE statement, the buffer is specified by the user. It is usually an array or part of an array. Conversions then take place into and out of that buffer area. These statements have the following forms:

DECODE(c,f,s,n) list	ENCODE(c,f,s,n) list
or	or
DECODE(c,f,s) list	ENCODE(c,f,s) list

where:

c defines the number of characters per internal record (in the buffer area). It is either an integer constant or a simple integer variable.

f specifies a FORMAT statement. It is either a statement number or is the name of an array in which a FORMAT has been stored.

s indicates the start of the internal buffer. It may be an array name, an array element, or a simple variable. If it is a simple variable, it is usually equivalenced to part of an array to provide room for the buffer (see EQUIVALENCE statement in chapter 6).

n is a simple integer variable into which will be stored, on completion of the operation, the number of characters actually processed (scanned or generated).

list is an input/output list..

In a READ/WRITE operation, the size of external records is predetermined; for example, cards are eighty characters long. In a DECODE/ENCODE operation, there are no physical considerations to determine this, so you can specify records of whatever length you like, though we recommend a multiple of two characters. The "records" are simply consecutive areas of memory within the buffer area. Each one begins right after the preceding one ends. For example, if you specify 10-character records, the first five words of the

buffer constitute the first record, the next five words the second record, and so on. As with READ/WRITE operations, the FORMAT statement determines when to start a new record; over-flow from the previous record does not.

The characters in the buffer area are processed at two per word, without regard to the type of the variable or array used to define the start of the buffer.

DECODE Statement

The DECODE statement causes the character string beginning at s to be decoded, according to the FORMAT specified by f, and stored into the items in the I/O list. When the FORMAT specifies a new record, the rest of the current record (of length c) is skipped. If you try to read more than c characters from a record, the extra ones will be blanks.

As an example, consider the case described above of the two kinds of records indicated by a 1 or a 2 in the first column. These could be processed by the following statements:

```
      DIMENSION KARD(39)
      READ(5,9) KEY, KARD
    9 FORMAT(I1, 1X, 39A2)
      GO TO (1,2) KEY
    1 DECODE(78,10,KARD) NAME1, NAME2
   10 FORMAT(2A2)
        .
        .
        .
    2 DECODE(78,20,KARD) NUM1, NUM2
   20 FORMAT(2I4)
        .
        .
        .
```

The READ statement converts the value of KEY from column 1, skips column 2, and stores the next 78 columns in KARD(1) through KARD(39) at two characters per word. (The 39A2 format does this.) Then if KEY is 1, the first DECODE is performed. It processes two alphanumeric strings, each of length 2 characters (as specified by the 2A2) and stores them in NAME1 and NAME2. Otherwise, if KEY equals 2, the second DECODE is done. It scans two 4-character integer fields (2I4), does the required decimal to binary conversion, and stores them in NUM1 and NUM2.

DECODE essentially provides the capability of "rereading" an input record.

ENCODE Statement

An ENCODE statement converts list items into ASCII character strings, according to the format f, and places them in the buffer beginning at location s. If it tries to create more than c characters in a record, the extra ones are lost. They do not flow over

into the next record. When it writes fewer than c characters, the remainder are blanks. In fact, like the formatted WRITE, the first thing ENCODE does with each record is to set it to all blanks. This fact means that you cannot "rewrite" a record with two or more ENCODE statements in quite the way that you can "reread" one with several DECODE statements, since each ENCODE operation will blank out the prevous information. (However, the same effect can often be obtained by using small record lengths and only encoding certain sections with each statement.)

For example, the statements:

```
      DIMENSION JBUF(12)
      X = 4.67
      N = -33
      ENCODE(18,3,JBUF(3)) X, N
    3 FORMAT('VALUES:' , F5.1 , I4)
```

would produce an 18-character string occupying JBUF(3) through JBUF(11), and this string would consist of:

```
VALUES:bb4.7b-33bb
```

Since the FORMAT statement never specified a new record, JBUF(12) would not be affected.

AUXILIARY INPUT/OUTPUT STATEMENTS

These three statements are used for manipulating magnetic tapes and equivalent sequential files on disk.

REWIND Statement

```
REWIND u
```

where: u is the unit number, an integer constant or simple variable.

This rewinds tape unit u to its starting point. If end-of-files have been written, it rewinds past all of them.

BACKSPACE Statement

```
BACKSPACE u
```

where: u is the unit number, as described above.

Tape unit u is backspaced over one logical record. Usually this means one physical record. However, if the data was written by an unformatted (binary) WRITE statement, then one logical record may consist of a number of physical records. In other words, in binary, the BACKSPACE statement always backspaces over all of the information written out by a single binary WRITE statement. This is made possible by the special control words that the binary WRITE statement attaches to its records.

If a tape is positioned at its starting point, a BACKSPACE or REWIND has no effect.

END FILE Statement

END FILE u

where: u is the unit number.

This writes an end-of-file mark on tape unit u. If a tape is being simulated by a sequential disk file, the END FILE statement writes a special indicator that can be recognized as an end-of-file by the END= option, discussed above.

FORMAT STATEMENT

The FORMAT statement operates in conjunction with a formatted READ or WRITE, DECODE, or ENCODE statement. It controls how the characters in each input record are to be interpreted in assigning values to the list items, and how output list items are to be converted to character strings and where these strings are placed in output records. Generally the conversion performed on output by any specification is the reverse of that performed on input.

The FORMAT statement has this basic structure:

k FORMAT (specifications)

The label, k, is shown here because this is one statement that should always have a label. Otherwise it cannot be used.

There are a large number of different kinds of specifications, which are individually described below. Usually they are separated by commas. Instead of a comma, one or more slashes (/) may act as a separator. The slash is itself a specification (for new record), but syntactically it acts as a separator rather than as one of the items to be separated. In certain cases, the separator may be omitted entirely. This is permitted following any H,', or X specification.

Most format specifications operate on one of the I/O list items. (In the case of a complex item, two specifications are required, one for the real part and one for the imaginary part.) Other specifications operate by themselves and do not involve a list item. FORMAT and list interfacing is described in detail in a later section, but basically it works as follows. The FORMAT is processed from left to right. If a specification is one that operates by itself, then its operation is performed and the next specification is examined. If the specification is one that operates on a list item, then the next list item is obtained and the appropriate conversion is performed. If, however, there were no more list items at that point, then the I/O operation is finished, and processing of the FORMAT is terminated, even if it has not all been used. If the end of the FORMAT is reached, and there are still more list items, then the FORMAT is rescanned -- if no more list items, processing is finished. Note that the I/O list is always used completely and only once, while the FORMAT may not be finished or may be processed more than once. Groups of specifications may also be enclosed in parentheses (up to eight levels of nesting). This affects how the FORMAT is rescanned when it reaches the end, and will be explained later.

Computer Automation FORTRAN IV includes sixteen format specifications, which fall into five categories:

Decimal Conversion	Modifier	Non-decimal Conversion	Alphanumeric String	Record Position
rIw	nP	rZw	nHs	nX
rFw.d	$	rLw	's'	Tw
rEw.d	*	rAw		/
rDw.d				
rGw.d				

The capital letters and $, *, ', and / are specifications. The small letters (except for s) represent integer constants, which are counts with the following meanings:

r is a repeat count that causes the specification to be repeated r times (r > 0). If r is not present, it is 1. Thus 4I5 means I5,I5,I5,I5.

w specifies the total width of a field (w > 0).

d usually specifies the number of digits to the right of the decimal point (except on G format output) ($d \geq 0$).

n is a count of characters or of decimal scaling.

s is a string of alphanumeric characters.

These parameters are all discussed further in the sections on the appropriate specifications, below.

I Format (Integer)

Form: rIw
where: r is an optional repeat count.
w is the total field width that will be created or scanned.

I format is intended primarily for integers, but it can also handle variables as well as input fields that are floating point. In all cases, however, fractional parts will be lost.

Output. The integer value of the list item is converted to decimal and right-justified in a field of width w characters. If negative, it is preceded by a minus sign. All of this is preceded by blanks to fill out the field. If w is not specified large enough to hold all of the digits or the minus sign, this is an error -- no value is output. Instead, the whole field is filled with question marks to signal the overflow. A width of 6 is always large enough to avoid overflow.

Here are some examples of output using an I4 format:

Value	Output Field
7	bbb7
-12	b-12
0	bbb0
+9999.73	9999
-1000	????
32767	????

Input. A field of w characters is scanned for a decimal value, with or without a plus or minus sign. Leading blanks are ignored. Embedded or trailing blanks are treated as zeros. For this reason, you should be careful to right-justify input values in their field. Otherwise each trailing blank will increase the value by a power of ten.

If there is a decimal point and/or an exponent present in the field (i.e. a floating point number), the fractional part of the resulting value will be lost. See "Numeric Input Fields", later in this chapter, for more information on acceptable ways to write numeric values for input.

For example, suppose the following input field were read into five variables using a 5I4 format specification:

```
bbb1b-23486937.9b3bb
```

the resulting values would be 1, -23, 4869, 37, and 300. Note that, unlike free-form input with the INPUT statement, no separators are required between fields. The format determines where one value ends and the next begins. Therefore, be careful in preparing formatted input. If the values are off by even one column, the results will usually be different.

For added readability and safety, I format fields (and all other numeric input fields) may also be terminated with commas, as described in a subsequent section, "Comma Field Termination".

F Format (Fixed Decimal Point)

Form: rFw.d

where: r is the repeat count.

w is the field width.

d is the number of digits to the right of the decimal point (default value if no decimal point is input).

In standard FORTRAN, F Format is used only with real type data (or the parts of complex data). In Computer Automation FORTRAN IV, it and the other numeric formats (I,E,D,G) can be used with integer and double precision data as well. Integers will simply be converted to floating point, and will always have a fractional part of zero.

On input, F, E, D, and G formats operate exactly the same. On output, F produces no exponent (e.g. 375.4), E uses an E exponent (e.g. .3754E 03), D uses a D exponent (.3754D 03), and G uses either the F or E form, depending on the size of the number.

Output. The floating point value of the list item is converted to decimal, with d digits after the decimal point. It is rounded at the last digit and then right-justified in a field

of width w. As with integers, it is preceded by a minus sign if necessary, and then by blanks to fill out the field to the left. If w is not large enough to accomodate all the digits or the minus sign, an error is signaled by filling the entire field with question marks.

The following examples are for output with an F8.3 format:

Value	Output Field
2.75	bbb2.750
-31.4886	b-31.489
.000477	bbb0.000
8127	8127.000
-900.0007	-900.001
-999.9998	????????
22650.0	????????

To be sure that w is large enough, you have to have some idea how big the numbers will get, since they require more space as they get bigger. If n is the number of digits to the left of the decimal point and d is the number of digits to the right of the decimal point, then to allow for these digits and the decimal point and minus sign, w must be this large:

$$w \geq d + 2 + n$$

Input. The next w characters in the input field are scanned for a decimal value, which may or may not have a leading plus or minus sign, a decimal point, or a trailing exponent. Since there are a large variety of forms in which the number may appear (it is even possible to omit the E or D in the exponent), please refer to "Numeric Input Fields", later in this chapter for complete details.

As with I format, leading blanks are ignored, while embedded and trailing blanks are treated as zeros. This will not be so harmful if a decimal point has appeared, since the trailing zeros will have no effect, but keep it in mind.

If there is no decimal point in the input field, then by default one is assumed d positions from the right. This usually means d positions from the end of the field, but if there is an exponent it means d positions from the beginning of the exponent. Also, d positions mean actual character positions, regardless of whether they have blanks or digits in them.

For example, an F8.3 format would produce the following conversions:

Input Field	Resulting Value
bbbb1234	1.234
bbb1234b	12.34
bb1.234b	1.234
-.756bE4	-7560.
bb3.Eb1b	3.E10
-b3bEbb1	-.3
bbbbbbbb	0.

E Format (Floating Point with E Exponent)

Form: rEw.d

where: r, w, and d are the same as for F format.

E format is similar to F format, except that on output it always attaches an exponent to the value. This means that it can represent numbers of any size without needing extra width.

Output . The floating point value is converted to decimal in the form of a fractional part less than 1 followed by an exponent. The fractional part consists of a decimal point and exactly d digits. It is round at the d'th digit. The exponent consists of E followed by a space or a minus sign followed by a two digit decimal exponent. If the value is negative, it is preceded by a minus sign. Then it is right-justified in a field of width w and preceded by blanks. If w is not large enough, this is an error and the whole field will be filled with question marks. To accommodate d digits, the exponent, the decimal point, and a possible minus sign, this relationship should be observed:

$$w \geq d + 6$$

These are some examples of output using an E10.4 format:

Value	Output Field
.76	0.7600Eb00
12.537	0.1254Eb02
-0.000632	-.6320E-03
-99999.	-.1000Eb06
0.	0.0000Eb00

The P scale factor (described later) can be used to make the fractional part larger or smaller than its normal range of from .1 to less than 1.

Input. Originally E format may have been intended to read numbers with exponents, while F format was for numbers without exponents. Now, however, they operate identically on input, so the examples shown for F input also apply to E input. See also the section on "Numeric Input Fields", later in this chapter.

D Format (Floating Point with D Exponent)

Form: rDw.d

where r, w, and d are the same as for E and F format.

D format is exactly the same as E format, except that the exponent on output values contains a D instead of an E, to signal double precision. In ANSI standard FORTRAN, D format may only be used with double precision list items, while E and F formats may only be used with real ones, but in Computer Automation FORTRAN IV they may all be used interchangeably. This means that D format is typically not used very much.

As an example of D output, D10.4 would convert:

12.537 to b.1254Db02

Input under D format is exactly the same as for E and F formats. See also the section on "Numeric Input Fields", later in this chapter.

G Format (General)

Form: rGw.d

where:

r is the repeat count (optional).

w is the total field width.

d on input, is the default position of the decimal point (as with E and F). On output, however, it is the total number of significant digits to be produced.

G format is a combination of F and E formats. On output, it acts like either F or E, depending on which makes more sense for the size of number involved. It can be used with integer, real, double precision, or either part of complex data. Integers are converted to floating point first.

Output. G format attempts to express numbers in the most natural way, which is in F format unless they are too large or too small, in which case in E format. The d (above) specifies the number of significant digits to be output, and this is exactly the number of digits that will be produced. If the magnitude of the number is such that it can be expressed by placing the decimal point anywhere within or at either end of those d digits, then that will be done and no exponent will be needed. However, if preceding or trailing zeros would be required to express the value correctly (i.e. more than d digits total), then E format will be used instead; the number will be normalized and output with an exponent.

To express this algebraically, let M be the magnitude of the value to be output (rounded to d significant digits). Then select an integer p such that:

$$10^{p-1} \leq M < 10^{p} \qquad \text{(if M=0, then p=0)}$$

If the format is Gw.d, let j=w-4 and k=d-p. Then if $0 < p \leq d$, the format used is:

Fj.k,4X

On the other hand, if p is less than 0 or greater than d, the format is:

Ew.d

This had best be illustrated by some examples. The first column contains the values. The next two columns are the output fields produced by the formats shown.

Value	G8.3	G8.2
.07283	.728E-01	0.73E-01
.7283	.728bbbb	0.73bbbb
7.283	7.28bbbb	b7.3bbbb
72.83	72.8bbbb	b73.bbbb
728.3	728.bbbb	0.73Eb03
7283.	.728Eb04	0.73Eb04

When the F form is used, and there is no exponent, those four positions are blank. This causes the numbers to line up underneath each other better.

The size of w does not affect the choice of format; this is determined only by the size of d and the size of the value. If w is not large enough, the field is filled with question marks. To avoid this, the same rule applies as for E format:

$$w \geq d + 6$$

The P scale factor, described below, has a peculiarity in its effect on G format. It applies only when the E form is used, not the F form. This has two implications. First, all numbers output in G format appear as their actual value, never off by a power of ten. Second, values output in F form with a non-zero P scale factor cannot subsequently be input using the same format and obtain the same value. The scale factor will take effect during input but not during output. This is one of the few exceptions to the rule that what is output by a particular format can be input by the same format.

NOTE

The "d" field of a G format must always be included, as a positive non-zero value. For example, the value "123.456" output in a format of "G10.0" yields " .E 03".

Input. G format input is exactly the same as F, E, and D input. See also "Numeric Input Fields".

P Specification (Scale Factor or Power of 10)

Form: nP

where: n is a positive or negative integer (or zero) that specifies the power of ten to be used as a scale factor.

The P scale factor is a modifier that can be applied to any F, E, D, or G format to change the position of the decimal point, i.e. to multiply or divide the value by a power of 10. It is not separated from the format specification by a comma; it precedes it immediately. If the format has a repeat count (or a $ or * modifier, described below), the scale factor precedes that too. For example:

```
3PF10.2
-1P3E14.6
0PG9.3
-2P5*$F12.2
```

At the beginning of a FORMAT statement, no scale factor is in effect. This is equivalent to a scale factor of zero. Whenever a non-zero P is used, it continues to apply to all floating point formats thereafter until changed again. It is not reset to zero if the FORMAT is rescanned due to additional list items. To reset it to zero, you must specify a 0P. Thus the following statements:

```
      WRITE(6,1) A, B, C, D, E
    1 FORMAT( F10.2 , E12.4 , 2PF7.3)
```

Process the list items with the following effective formats:

Variable	Format
A	F10.2
B	E12.4
C	2PF7.3
D	2PF10.2
E	2PE12.4

Output. The internal value is multiplied by 10^n before output. In other words, the decimal point is moved right n places. (Of course, if n is negative, the decimal point is moved left.) On F format, this causes the number to appear larger or smaller than it really is. However, on E and D formats, the exponent is decreased or increased to compensate for the change in mantissa, so the actual value remains the same. The only effect is to change the form of the number by introducing digits to the left of the decimal point or zeros to the right of the decimal point. This is illustrated in the following examples:

P	Value =7.3629		Value=9.9	
Scale	F6.2	E9.2	F7.3	E10.3
2	736.29	73.63E-01	990.000	99.000E-01
1	73.63	7.36E 00	99.000	9.900E 00
0	7.36	.74E 01	9.900	.990E 01
-1	.74	.07E 02	.990	.099E 02
-2	.07	.00E 04	.099	.001E 04
-3	.01	.00E 05	.010	.000E 05
-4	.00	.00E 06	.001	.000E 06

Scaling on D format is exactly the same as for E. G format is a little strange. When it chooses the E form, the scale factor works in the usual way, increasing the mantissa and decreasing the exponent. This leaves the actual value the same. In order to be consistent and say that G format always outputs the correct actual value, when it chooses F form the scale factor (if any) does not take effect; it is ignored. Note, however, that this introduces the inconsistency that if you output a number in G format with a P scale factor, you may not be able to read it in again with the same format and get the same value, since the P will apply during input (see below).

When a scale factor is in effect, numbers are rounded after the scaling has been performed This can have an interesting affect on E format. In order to get the proper number of digits left of the decimal point (or zeros right of it), an extra shift may be required. This explains the discontinuous way that the exponents change in the above examples.

The value zero is not affected by a scale factor.

Input. In general, the effect of a P scale factor on input is to reverse what it would have done on output. (The only exception is the one above concerning G format.) This means the external value is divided by 10^n, or the decimal point is moved left n places. However, remember that in E form output, the exponent was changed to compensate for the moved decimal point, leaving the actual value unchanged. Therefore, on input, if a number has an exponent specified, it is assumed correct and no shifting is done. In other words, a P scale factor affects an input field only if it does not have an exponent. For example:

External Field	Resulting value as function of P scale		
	0P	1P	-2P
0.68	.68	.068	68.
0.68E0	.68	.68	.68

This is true for all of the floating point formats, F, E, D, and G, since they all work identically on input.

Although there seem to be a lot of exceptions about the effect of P scale factors, these two rules are always true, on both input and output:

1. If the number in the external field has an exponent, then it is equal to the internal value, regardless of any P scale factor.

2. If it does not have an exponent, then:

   external value = internal value x 10^n

$ Specification (Preceding Dollar Sign)

Form: $

A special Computer Automation feature, the $ modifier enables you to print amounts of money with a dollar sign immediately preceding, even with values of various sizes. It applies to either F or I format and should be written immediately ahead of the F or I (i.e. after a repeat count or any other modifier). For example:

```
$I5          3$F10.2          -2P2*$F20.2
```

Output. After the value is right-justified in its field, a dollar sign will be placed immediately ahead of the first character, which is usually a digit but may be a decimal point or minus sign. The actual position of the dollar sign will depend on the size of the number. If the field width (w) is not large enough to allow for the dollar sign ahead of the number, this is an error and the overflow will be signalled by filling the entire field with question marks. It is a good idea, therefore, to allow plenty of width on the format specification.

For example, here is how the value 46.35 would be printed using various formats with the $ modifier:

Format	Output Field
$F8.2	bb$46.35
$I4	b$46
$F4.0	$46.
$F5.2	?????

The value -0.98 printed with a $F9.2 format would yield:

bbbb$-.98

Input. $ is intended primarily for output. However, to be consistent, what it does on input is to allow and ignore a dollar sign preceding a number. Thus these two fields would be treated as the same:

bbb$4000.00 and bbbb4000.00

The dollar sign must precede a minus sign if there is one. If the $ modifier does not appear on the format specification, then a dollar sign may not appear in the field. If one does, it will be detected as an error.

*Specification (Asterisk Fill)

Form: *

This is another special Computer Automation feature. It is often used in conjunction with $ for printing checks. It causes the left part of the field to be filled with asterisks instead of blanks. Like $, the * is a modifier that appears ahead of an I or F, but after any repeat count. If both * and $ are used, the * should come first. (This is easy to remember, because that is the way they will appear in the output field, with the asterisks first.) For example:

*F10.4 3*I20 *$F12.2

Output. All of the positions that would normally contain preceding blanks, to fill out the field to the left, are changed to asterisks. The same is true if both * and $ are used at the same time; the dollar sign is inserted first and then the remaining positions are filled with asterisks. It is not a error if there are no preceding positions to put asterisks into. The asterisk is simply a substitute for blank, used if necessary to fill out the field. Everything else must still fit in the field, including a dollar sign if specified.

Here are some examples of the use of *, some in combination with $:

Value	Format	Output Field
91.27	*F9.2	****91.27
-4062.948	*F9.2	*-4062.95
3000	-2P*$F6.2	$30.00
27	*$I10	*******$27
0	*I6	*****0

Input. Like $, the * feature is intended mainly for output, but does something consistent on input. It allows and ignores any number of preceding asterisks, up until it finds something that is not an asterisk. For example, all of the above output fields could be read as input fields using the same formats. If the * modifier does not appear in the format specification, then the asterisk may not appear in the field. If one does, it will be detected as an error.

Numeric Input Fields

There are a variety of ways that you can express a numeric value for input, and they are all equally permissible under any of the numeric formats, namely I, F, E, D, and G. Any field that can be read using one of these formats can be read using any of the others. The resulting value will be the same, too, except for the truncation performed by I format. This means that numbers input by I format need not be integers (though they usually are), numbers input by F format can have exponents, etc.. Free form input (by the INPUT statement) is the same too, except for two things: blanks are ignored and and there is no fixed field width (see below).

A numeric input field can be thought of as having two parts, a mantissa and an exponent. If either part is missing, it is assumed zero. (Of course it is sort of pointless to have an exponent on a zero mantissa, but it is legal.) The mantissa may take any of these forms:

nnn nnn. .nnn nnn.nnn

(where nnn is a string of decimal digits). It may be preceded by a plus or minus sign.

The exponent is normally written in one of these ways:

Eee E+ee E-ee

where ee is a one or two digit power of 10 to multiply the mantissa by. If the plus or minus sign is present, it is also permissible to leave out the E. We do not particularly recommend this form, since it is less readable and less like the form of source program constants, but it is a traditional feature in FORTRAN and is thus allowed. In this case, the field has the form:

mantissa+ee or mantissa-ee

A D may be used instead of E in the exponent, with no change in meaning. It is not necessary to signal that an input value is double precision, either by using a D exponent or by using D format. If the variable in an input list is double precision, then its input field will be processed in double precision regardless of the format or exponent used. And if the variable is single precision, a D exponent on the data will not make it double.

The rest of this discussion applies only to formatted input, not to free form input by the INPUT statement.

When the mantissa contains no decimal point, the decimal point is assumed to be d positions from the right end. (On I format d is automatically zero.) This means the right end of the whole field unless there is an exponent. If there is an exponent, it means d positions from the beginning of the exponent (which may begin with E, D, +, or -). In other words, from wherever the mantissa ends, you count back d positions (including blanks) to place the decimal point.

When using formatted input, remember that blanks will usually be treated as zeros and can change the value of either the mantissa or the exponent. Leading blanks (on either the mantissa or exponent) do not have any effect, except that they are counted as part of the field width. However, once a digit or decimal point has been found, any embedded or trailing blanks that follow are interpreted as zeros. Following are some examples of permissible numeric input fields and how they are interpreted. Notice that F, E, D, and G are interchangeable on input, and that when a decimal point appears in the field, it makes no difference what the value of d is in the format.

Input Field	Format	Resulting Value
b23.b	F5.2	23.
b2b.b	E5.1	20.
b2bbb	D5.1	200.
-b.b7	G5.0	-.07
b5bEbb3	E7.2	.5E 3
b5bEb3b	E7.2	.5E30
4.60E+1	F7.0	46.
4.600+1	F7.0	46.
b-b1b-b1b	G9.0	-10.E-10
b+-b	D4.2	0.

Some of the above are pretty strange and misleading representations, and we do not particularly recommend them, but they illustrate how formatted processing works. To keep things simple and avoid mistakes, we recommend these conventions:

1. Do not embed blanks in strings of digits.

2. Make sure numbers are right-justified in their fields, so there will not be trailing blanks. Or use a comma terminator, described below.

3. Use the E to introduce an exponent.

4. Use a decimal point when needed, rather than relying on the default value (d) in the format.

5. Use a value of zero for d (on input), so that the default decimal point will be at the end of the number, where it naturally belongs.

Comma Field Termination

One of the problems with using formatted input, particularly, say, if you are typing in numbers at a typewriter, is that you have to know the exact field width specified in the format and then you have to count carefully to make sure you right-justify the number in the field. This kind of input was really designed for cards, where columns are clearly marked, and even there it is not always convenient.

Computer Automation FORTRAN IV provides a way of avoiding this problem. Any field being processed by an I, F, E, D, G, Z, or L format may be terminated early by a comma. When the comma is encountered, the field is treated as ending on the previous character, even though the field width (w) has not been used up. Thus you can avoid trailing blanks, even when you do not know what the field width is. For example, to read two values using a 216 format, instead of having to use:

```
bbbb13bbbbb4
```

you can use:

```
13,4,
```

After a field is terminated by a comma, the next field begins immediately after the comma, rather than where it would have begun if the full width had been used. The above example illustrates one difference between this and free form input (by the INPUT statement). There had to be a comma after the 4, even though it was the last value on the line, because otherwise the field would have had its declared width of 6, including five trailing blanks that would be interpreted as zeros.

However, end-of-line is also treated as a terminator, in exactly the same way as comma. This is significant mainly on typewriter input, where a carriage return indicates end-of-line. Thus when typing in values, it is not necessary to follow the last one on a line by a comma to terminate it. The carriage return will have the same effect. The above example could therefore be typed in as:

```
13,4(CR)
```

A comma terminator has another significant effect. It not only overrides the value of w specified in the format, it also overrides the value of d. A default decimal point makes some sense when fields end in a specific column, but when they can end anywhere before that, it becomes more difficult to remember where the decimal point is going to end up. So we have made the rule that, when numeric fields end on a comma they should look like what they really are. If they have no decimal point, they are an integer value. (In other words, the decimal point is assumed at the comma or, if there is an exponent, at the beginning of the exponent.) For example, the four values produced by a 4F20.3 format in reading the line:

```
bb12345,62bbb,b504E-3,bbb2.,
```

would be:

```
12345. , 62000. , .504 , 2.
```

Notice that we used a very large field width on the F format, yet the input fields need be only as long as required to express the numbers. It is a good idea to use large widths in this case, especially in situations where the input values are being prepared by someone who did not write the program. This will make sure that there is plenty of room for the number and the comma. If the field is too short and the comma falls beyond it, then it is too late to terminate the field. For example, if you used an I3 format to read the line:

```
483,
```

the first field was already terminated at the third digit, so the comma falls in the following field and terminates it.

A blank or empty field always means zero, so you can use consecutive commas to represent a group of zero values. For example;

```
1,,2, , , 3,
```

represents:

```
1, 0, 2, 0, 0, 3
```

Z Format (Hexadecimal)

Form: rZw
where: r is the repeat count.
w is the field width.

Z format operates on internal values at the rate of four bits per hexadecimal digit (four digits per word), regardless of the type of the value. Thus an integer contains four hexadecimal digits, a real value eight, etc. The hexadecimal digits are:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Output. If w is exactly the right size for the data type (e.g. 4 for integer), then the entire value is output in hexadecimal, including leading zeros. If w is larger than this, the hexadecimal number is right-justified in the field and preceded by blanks (as with the other numeric formats).

If w is smaller than needed, only the w rightmost digits will be output -- the ones on the left will be skipped. This is not considered an overflow error, so no question marks will be printed. Z format is specifically designed to be able to print just part of a number.

The modifiers P, *, and $ do not apply to Z format.

Examples:

List Item Type	Value	Format	Output Field
Integer	:C102	Z4	C102
Logical	:0000	Z6	bb0000
Real	:47D00A80	Z5	00A80
Real	:FF1234EB	Z9	bFF1234EB

Input. When w is exactly the right size, the list item is completely filled with the hexadecimal digits in the field. If w is smaller, then the number is treated as a hexadecimal integer; i.e., it is right-justified with preceding zeros in the list item.

If w is too large, the left-hand characters are skipped, and the required digits are taken from the right end of the field. In all these cases, the operation on input is the reverse of that on output.

Hexadecimal input fields may also be terminated early with a comma. In that case the same rules apply, but w is counted as being the number of characters that appeared before the comma. As in other numeric formats, blanks are treated as zeros.

Examples:

Input Field	Format	List Item Type	Resulting Value
3A22	Z4	Integer	:3A22
ABCD	Z2	"	:00AB
C1D2E3	Z5	"	:1D2E
bb5CCbbb7b	Z9	Real	:05CC0007
A800,	Z8	"	:0000A800
123,	Z2	Integer	:0012

L Format (Logical)

Form: rLw

where: r is the repeat count.
w is the field width.

L format operates only on list items of logical type and the values true and false.

Output. The logical value is converted to either T or F and right-justified in the field with preceding blanks. For example, a value that is true, when output with an L5 format, becomes:

bbbbT

Input. Within the field of width w, the first T or F determines the value. If neither a T nor F is found, the value is false. Characters appearing between the T or F and the end of the field are ignored, except for comma, which terminates the field. The following examples are input with an L6 format:

Input Field	Resulting Value
bbbbbT	True
AFTERb	False
X=Y+37	False
.TRUE.	True
X,TRUE	False

A Format (Alphanumeric)

Form: rAw

where: r is the repeat count.
w is the field width.

A format converts internal values to ASCII character strings (and vice versa) at the rate of eight bits per character (two characters per word), regardless of the type of the value. Thus an integer contains two characters, a double precision value eight, etc. The ASCII characters and their hexadecimal equivalents are shown in appendix C.

Output. If w is exactly the right size for the data type (e.g. 4 for real), then the entire value is output as a character string. If w is larger, the character string is right-justified in the field and preceded by blanks (as with other formats).

If w is smaller than needed, only the first w characters will be output -- the ones to the right of this will not. This is the opposite of Z format (where information is lost at the left). In both cases, this is not an overflow error. Both A and Z format are designed to be able to process part of an item -- A format the front part, and Z format the back part.

Examples:

List Item Type	Value	Format	Output Field
Integer	:DAB3	A2	Z3
"	:DAB3	A5	bbbZ3
Real	:C3C1C9BF	A5	bCAI?
"	:C3C1C9BF	A3	CAI

Not all of the 256 combinations of eight bits correspond to printable ASCII characters. (In fact, most of them do not.) Therefore you should not use A format to output miscellaneous numeric values and expect to read what comes out. The variables should already contain alphanumeric information, either previously read in by another A format, or set up by a Hollerith or string constant, or even numerically constructed.

Although integer variables only hold two characters, in many applications it is a good idea to use them for working with alphanumeric information. They can be operated on (e.g. masking, shifting) more easily. You can use integer arrays to hold as many characters as you need.

For printing headings and other messages, H format (below) is generally more convenient than A format, since it does not require any variables to have been set up.

Input. When w is exactly the right size, the list item is completely filled with the ASCII characters in the field. If w is larger, the characters are taken from the right end of the field. The left-hand characters are skipped.

If w is smaller than needed, then the characters in the field are left-justified in the list item and followed by blanks to fill out the rest of it (as with Hollerith constants). This is the opposite of Z format (and numeric formats in general), which locate short input fields at the right end of the value. A general rule for alphanumeric data is that internal values are left-justified, while external fields are right-justified.

Blanks have no significance in an A format input field as they are just another character. The same is true of comma, which will not terminate an alphanumeric field.

Examples:

Input Field	Format	List Item Type	Resulting Value	
UP	A2	Integer	:D5D0	(2HUP)
DOWN	A4	"	:D7CE	(2HWN)
TRUE	A1	Real	:D4A0A0A0	(4HTbbb)
b3,b	A4	"	:A0B3ACA0	(4Hb3,b)

H Format (Hollerith)

Form: nHs

where: s is a string of alphanumeric characters of any length.
n is the positive integer count of the number of characters in the string, including blanks.

H format is one of the formats that operates without a list item. It transfers character strings directly from the FORMAT statement into the external field or (less often) vice versa. Note that the form of a Hollerith format is just like that of a Hollerith or string constant.

Output. The n characters following the H are transmitted to the next n positions in the output field. For example, if R equals 12.75, the statements:

```
      WRITE(6,10) R
   10 FORMAT(11HbINTERESTb=,F6.2,8H%/MONTH!)
```

would print the following line:

```
INTEREST = 12.75%/MONTH!
```

Note that the blank after the equal sign came not from the H format, but from the F format (a leading blank).

Count the characters very carefully in an H string. If your count is too high, it may extend over into some other format specifications; if it is too low, part of the alphanumeric string will be interpreted as formats. Better yet, if the string is very long, use the ' format (below), which does not require a count.

Input. H format is primarily designed for output, but there are occasions when it can be useful on input. What it does is to take the next n characters from the input field and insert them into the FORMAT statement, replacing the characters that were there before. The original characters are then lost. This might be used to change a title printed to identify each set of output values. For example, if the statements:

```
      READ(5,7)
    7 FORMAT (35H                                    )
```

read an input line containing:

```
KOHOUTEK ORBITAL COEFFICIENTS
```

the FORMAT statement would be changed to:

```
    7 FORMAT(35H KOHOUTEK ORBITAL COEFFICIENTS     )
```

which could then be printed with a WRITE statement.

' Format (Hollerith)

Form: 's'

where: s is an alphanumeric string of any length.

This is an alternate form of the H format, with the character string enclosed in quotes rather than being counted. For that reason, it is probably easier to use, especially on long strings.

Output. The characters between the quote marks are transmitted to the output field, which will have the same length. To include the single quote character itself in the string, it should be written as two single quotes. For example:

```
'JOE''S PLACE'
```

is equivalent to:

```
11HJOE'S PLACE
```

In this case, the quotes must be truly consecutive. If there is even one blank between them, they will be interpreted as the end of one string and the beginning of another (since no comma separator is required). This is the only situation in FORTRAN where a blank is significant in a statement without being contained in an alphanumeric string.

Input. As many characters are taken from the input field as are needed to fill the positions between the quote marks. As with H format, this feature is less often used. There should not be any quote characters in the input field. If there are, they will be changed to blanks. Otherwise, they could have disastrous effects on the FORMAT statement.

X Specification (Skip)

Form: nX

where: n is the positive integer count of how many positions to skip over.

NOTE

An X with no "n" value preceding it will be ignored; it is not equivalent to 1X.

X format skips over the next n characters in the external field. It transmits no data.

Output. Normally you can think of X as creating n blank spaces in the output field. For example:

```
FORMAT(I5, 5X, 'SAMPLE')
```

might produce this line:

```
bbb32bbbbbSAMPLE
```

However, the T format (below) can be used to back up in a line. Using an X then would not blank out what had previously been written. The only reason it seems to do so normally is that all output lines are set to blanks initially.

Input. The next n character positions in the input field are ignored. The next format will pick up processing at the n+1st position.

T Specification (Tab)

Form: Tw

where: w is the character position to tab to.

T is much like X, in that it transmits no data but merely changes the character position. Instead of skipping forward a fixed amount, it skips to a particular column. It works like the tab key on a typewriter, except that it can tab backwards as well as forwards. This means, on output, that previously written characters can be written over, and on input, fields can be read more than once. (This is only occasionally useful.)

One useful thing that T does that X cannot is to get you to a particular input column following a field that has been terminated early with a comma. Since you do not know where the comma is going to be, you cannot skip forward with an X to a fixed place. For example, you might have agents preparing cards with a number somewhere in the first 20 columns and a name beginning in column 21. If they use a comma to terminate their values (which is a good idea), the following FORMAT would handle this:

```
FORMAT(F20.0, T21, 10A2)
```

If a card did not have a comma terminator, then the T would have no affect.

The first position on a line is 1. You cannot tab to the left of that.

/ Specification (New Record)

Form: /

Syntactically the slash acts as a separator (i.e. like a comma) in a FORMAT statement. Any number of slashes may appear between two specifications or at the beginning or end of the list of specifications. A comma should not be used before or after a slash. For example:

```
FORMAT(/F10.2,5X,4A2/2014//8Z10,'FINAL'///)
```

Whenever a slash is encountered, the current record is terminated and a new record is begun. On output this means that the old record is written out and processing starts at column 1 of the new record. On input it means that any data remaining on the old record is not processed. The next record is read and scanning starts at column 1.

If a slash is followed immediately by another slash (or the end of the FORMAT), then the record just begun is terminated without any processing. On output this means a blank record is written. On input it means that one record is skipped.

In some FORTRANs, a slash preceding the final right parenthesis of a FORMAT does not take effect on output (no blank record is produced). In Computer Automation FORTRAN IV it does, so input and output are consistent.

As an example, if the FORMAT shown above were used with an I/O list with the proper number of items (namely 33), it would write (or read) eight records, with the 1st, 4th, 6th, 7th, and 8th being blank (or ignored).

Parenthesized Format Groups

A group of format specifications may be repeated by enclosing them in parentheses and putting a repeat count in front.

For example:

```
FORMAT(I3 , 2(3A4,2X) , F10.2 , 3(5HRAH! ))
```

is equivalent to:

```
FORMAT(I3,3A4,2X,3A4,2X,F10.2,15HRAH! RAH! RAH! )
```

except for the way that rescan takes place (see below). Note that a Hollerith format cannot have a repeat count, so the only way to repeat it is within parentheses.

Parenthesized groups may be nested within each other, to a depth of eight.

If the end of the FORMAT comes and there are still more list items, the FORMAT is rescanned. However, if there are any parenthesized groups, the rescan begins at the last such group, rather than at the beginning of the FORMAT. Therefore, in the first example above, it would rescan only the 3(5HRAH!). This would not do much good, since that part contains no formats that can transmit any data. In this case, to get the whole FORMAT rescanned, you could write:

```
FORMAT((I3,2(3A4,2X),F10.2,3(5HRAH! )))
```

so that the last parenthesized group (determined by where it ends) is the large one surrounding all of the specifications. The following section gives a more complete description of how this works.

FORMAT and List Interfacing

Formatted input/output operations are controlled more by the FORMAT statement than by the I/O list. When a READ or WRITE (or DECODE or ENCODE) statement is executed, the FORMAT processor takes control. It proceeds by the following steps:

1. Each time one of these statements is begun, a new input record is read, or construction of a new output record commences. Thus each statement must process at least one record.

2. A record is terminated (i.e. no longer scanned, on input, or written out, on output) when any one of these three things happens:

   a. A slash is found in the FORMAT.

   b. The final right parenthesis is reached and there are still more list items, so the FORMAT has to be rescanned. A rescan never processes the same record.

   c. There are no more list items. This can happen either at the final right parenthesis or at a format specification that would require another list item (e.g. F format).

3. A new record is begun on either condition a or b above. Condition c is the end of the statement, so no new record is begun.

4. Any specification that does not require a list item (i.e. H, ', X, T, or /) is always processed when it is encountered, regardless of whether there are any more list items.

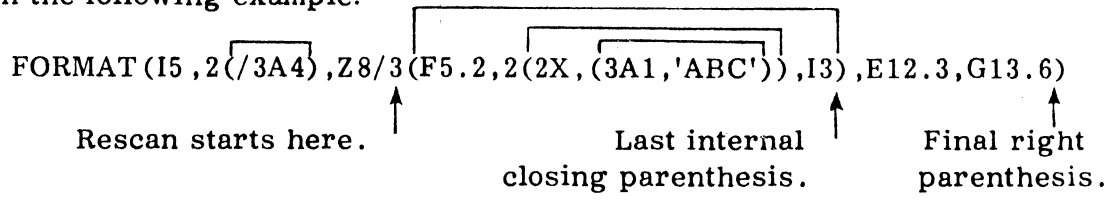
5. A specification that does require a list item (i.e. I, F, E, D, G, Z, L, or A) causes the FORMAT processor to look and see if there are any remaining. If there is one, it performs the appropriate conversion and proceeds (unless there is a type conflict between the format and the variable, which is detected as an error). On the other hand, if there are no more list items, the current record is terminated (written out if output), the input/output statement is finished, and the next statement is executed.

6. When the final right parenthesis is reached, the FORMAT processor again looks to see if there are any more list items. If not, the operation is terminated,

as described above. However, if there are more list items, the current record is terminated, a new record is begun, and the FORMAT is rescanned. The rescan takes place as follows:

- If there are no parenthesized groups of specifications in the FORMAT statement, the whole FORMAT is rescanned.

- If there are any parenthesized groups, the rescan starts at the group whose right parenthesis was the last one found before the final right parenthesis of the FORMAT statement. To put it another way, the outer parentheses are called level zero. Nested groups within are then referred to as level one, level two, etc., according to their depth of nesting. The rescan goes back to the most recent level one group, if there is one, otherwise level zero. If the level one group has a repeat count, it is included in the rescan. This is illustrated in the following example:



- If the part to be rescanned contains no format specifications capable of transmitting data (i.e. no I, F, E, D, G, Z, L, or A), but there are still list items that need to be transmitted, this is an error and will be diagnosed. The same is true if the whole FORMAT statement contains no data formats and there are any list items.

7. Each list item requires one format specification (or one repetition of a repeated format), except for complex variables, which require two, one for the real part and one for the imaginary part.

8. Whenever a new record is read, or an old record written out, and an error occurs or an end-of-file is found, and the ERR= or END= option has been specified on the READ/WRITE statement, processing terminates immediately and control is transferred to the specified statement.

FORMATs Stored In Arrays

This feature makes it possible to use a variety of FORMATs without having to include all of them in the program as FORMAT statements, each one of which is referenced by a different, but identical, READ or WRITE statement. Instead of referencing the label

of a FORMAT statement, any READ, WRITE, DECODE, or ENCODE statement can reference the name of an array. The FORMAT can then be stored in the array, as an ASCII character string.

The first character in the array should be the opening left parenthesis. The rest of the format specifications follow, and then the closing right parenthesis. The letters "FORMAT" do not appear.

The FORMAT can be constructed in the array, using Hollerith constants, DATA statements, etc. However, more often it is read in at run time, using "A" format. (In fact, this feature is sometimes called "FORMATs at run time".) For example, these statements could appear first:

```
      DIMENSION MM(10)
      READ(5,1) MM
    1 FORMAT(10A2)
```

and read in the line:

```
(2F10.3,I7)
```

The array MM would then contain the following values:

MM(1)	MM(2)	MM(3)	MM(4)	MM(5)	MM(6)	MM(7)	MM(8)	MM(9)	MM(10)
'(2'	'F1'	'0.'	'3,'	'I7'	')b'	'bb'	'bb'	'bb'	'bb'

Now MM can be referenced as a FORMAT; for example:

```
      READ(5,MM) X, Y, K
```

The FORMAT processor will go to the first element of the array to begin, instead of to a FORMAT statement.

You want to be careful to fill up all character positions of each of the array elements that will be scanned; that is, two character per element if integer, eight if double precision, etc. Otherwise there will be gaps between the FORMAT characters. This can be disastrous, especially on the Hollerith formats, which will include these gaps as part of character strings. This problem can also occur when using the ANSI allocation option (see chapter 9).

CARRIAGE CONTROL FOR PRINTING

Normally printed output is single spaced; each record appears on the next line. There are provisions for double spacing, and ejecting to the top of a new page, and you should be aware of them so that you will not activate them accidentally or lose information.

The first character position in any line that is being output to a print device is reserved for a vertical carriage control character. There are two such control characters, and they cause the following actions to be taken:

Character	Action
0	Upspace two lines before printing (double space).
1	Skip to top of page before printing (page eject).

Any other character causes a normal single upspace before printing. (Overprinting (+ in column 1) is not supported.) In any case, column 1 is never printed. It serves only to control carriage action. The actual line is considered to begin in column 2, so column 2 will be printed in column 1 on the paper (i.e. the whole line is shifted left one position).

Carriage control is usually specified with a 1Hx format at the beginning of the FORMAT statement (1Hb provides normal spacing). However, information in column 1 could result from almost any format specification (e.g. F or A), in which case it would be lost and might also produce an unexpected printer action. Therefore, if you are not looking for carriage control, be careful that your formats will not produce anything in column 1. The free form OUTPUT statement always begins its output in column 2 so that no carriage control action will occur.

Note that if a record is output to some other device, such as a magnetic tape, column 1 will be included. If the tape is later listed, the same carriage control action will take place as if it had been printed directly.

CHAPTER 6

DECLARATION STATEMENTS

CLASSIFICATION OF NAMES

Every name in a FORTRAN Program is classified as one of the following:

1. Scalar (simple variable)
2. Array
3. Subprogram
4. COMMON block

If it is a scalar or array, it must have a type. Subprograms have a type if they are functions, but not if they are subroutines. Some of these classifications require explicit declaration, using a declaration statement. Others result from implicit declaration; that is, the contexts in which the name is used.

Explicit Declarations

Explicit declarations include the following:

1. Arrays. In order to be used as an array, a name must first have been dimensioned. This can be done with a DIMENSION or type statement, or in a COMMON statement.

2. Type. The IJKLMN rule (see below) determines the type of a name, unless it is explicitly declared first, using an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL statement.

3. Subprograms. Subprograms can be defined or referenced within a program (sometimes both). You define a subprogram with a FUNCTION or SUBROUTINE statement, or by a statement function definition. These are described in the next chapter. Most subprograms that are only referenced are classified implicitly (see below). However, the EXTERNAL statement is used in certain cases.

4. Storage Allocation. Normally the compiler chooses where to allocate scalars and arrays. If you need to have them in a certain order, or overlapping, or you want to share storage with other programs, you can use the COMMON and EQUIVALENCE statements.

5. Data Initialization. FORTRAN does not guarantee the initial contents of variables upon loading, so you should not use a variable until it has been assigned a value. The DATA statement assigns initial values upon loading, so that you do not have to take the time and space to do it with assignment statements at run time.

In general, declaration statements must appear at the beginning of the program. EXTERNAL and DATA are exceptions. See appendix A.

Implicit Declarations

When you have not explicitly declared a name, it will be classified implicitly, usually at its first appearance in the program. This takes place according to the following rules:

1. A name that begins with I, J, K, L, M, or N is integer type. Any other name is real. Certain library functions are exceptions. See paragraph 6, below.

2. A name that is called with a CALL statement is a subprogram.

3. A name that appears in an expression, followed by an argument list enclosed in parentheses, is a function, i.e. a subprogram. Of course, if the name has previously been dimensioned, neither this nor the following rule would apply to it.

4. A name that appears to the left of an equal sign, followed by a dummy list enclosed in parentheses, is a statement function, i.e. a subprogram. These are described in the next chapter.

5. If a name first appears in any other context (than the above or a declaration statement), it is automatically classified as a scalar (simple) variable.

6. The complex and double precision functions in the library automatically have a known type, as long as they are used in the proper way as functions. Their type does not have to be declared.

Conflicting and Redundant Declarations

Conflicting and redundant declarations (either explicit or implicit) are not allowed. For example, once a name has appeared in a type statement, it should not appear in another one. A name may not be placed in COMMON twice, nor dimensioned twice. Once it has been dimensioned, it may not be used without subscripts (except where specifically allowed, such as in an I/O list or argument list). If a name has been

implicitly classified as a scalar, it may not be declared EXTERNAL. Errors such as these will be diagnosed by the compiler.

DIMENSION STATEMENT

The DIMENSION statement declares the dimensions of an array. It is written:

DIMENSION A_1, A_2, A_3, ...

where A is an array declaration. Array declarations (which may also appear in type and COMMON statements) have the form:

$v(r_1, r_2, \ldots, r_n)$

where: v is the name of the array.

n is the number of dimensions for the array. In Computer Automation FORTRAN IV, arrays may have any number of dimensions.

r defines the subscript range of each dimensions.

Usually the subscript range is specified by a single, unsigned integer representing the upper bound of that subscript. For example, a 3x10 array would be declared:

DIMENSION ALPHA(3,10)

This means that the first subscript runs from 1 to 3, the second from 1 to 10.

In Computer Automation FORTRAN IV, subscripts may have a lower bound other than 1. In this case, both the lower bound and the upper bound must be shown, separated by a colon. Thus the subscript range (r) can have either of the forms:

s_U or $s_L : s_U$

The lower bound is assumed 1 in the first case. When both bounds are specified, they may be positive, negative, or zero, as long as the upper bound is greater than the lower bound. For example:

DIMENSION STEP(0:10)

gives STEP a size of eleven elements, but the first is STEP(0) instead of STEP(1).

DIMENSION TIME(-60:+60)

declares TIME to have 121 elements, the first being TIME(-60), and the last being TIME(60).

Whenever an array element is referenced in the program it must have the same number of subscripts as dimensions, and each subscript must lie in the range declared for it.

In a subprogram, when v is a dummy array, the subscript limits, s_l and s_u may be unsigned dummy scalars instead of integers. This is discussed in the next chapter, under "Adjustable Dimensions".

Additional examples of DIMENSION statements:

```
DIMENSION PRICE(1900:1980,12), ND(0:100)
DIMENSION MGO(24), LTO(22), BB(36,22,34)
DIMENSION KLBOT(6,6,10,20), NCENT(-273:-100)
DIMENSION MATRIX(10,10)
```

Array Storage

An array cannot actually be represented in memory as a multiple dimensioned entity. It can only be strung out in order as a one-dimensional entity. Sometimes it is important to know the order in which multi-dimensional arrays will be stored. Two examples are : (1) when an array appears without subscripts in an input/output list, it is transmitted in storage order and (2) when an array is used to hold alpha-numeric strings (e.g. read in A format or set up by the DATA statement), these strings will be placed into consecutive array elements.

Arrays are stored starting at a lower memory address and moving to a higher memory address. The array elements are in order such that the first subscript varies most rapidly, the last subscript least rapidly. On a two-dimensional array, this is called "column-wise", since the columns are stored consecutively, but the rows are not. This rule applies whether the upper and lower bounds are positive or negative. For example, here are two arrays listed in storage order, showing the element count for each subscript combination:

DIMENSION X(2,3,2)		DIMENSION Y(-2:1,3)	
1	X(1,1,1)	1	Y(-2,1)
2	X(2,1,1)	2	Y(-1,1)
3	X(1,2,1)	3	Y(0,1)
4	X(2,2,1)	4	Y(1,1)
5	X(1,3,1)	5	Y(-2,2)
6	X(2,3,1)	6	Y(-1,2)
7	X(1,1,2)	7	Y(0,2)
8	X(2,1,2)	8	Y(1,2)
9	X(1,2,2)	9	Y(-2,3)
10	X(2,2,2)	10	Y(-1,3)
11	X(1,3,2)	11	Y(0,3)
12	X(2,3,2)	12	Y(1,3)

TYPE STATEMENTS

There are five type statements, used to explicitly declare the type of a scalar, array, or function. Since the IJKLMN rule implicitly classifies all names as either integer or real, you will need a type statement for all double precision, complex, or logical names (except certain library functions), plus whenever you want to override the IJKLMN rule. The type statements have the form:

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL

$N_1, N_2, N_3, \ldots$

where N is either the name of a scalar, array, or function, or it is an array declaration, i.e. the name of an array followed by dimensions enclosed in parentheses (as described in the previous section). Whenever an array declaration appears, the statement is acting as both a type statement and a DIMENSION statement, so no DIMENSION statement is needed. For example, the statements:

```
COMPLEX C1, Z
REAL ALPHA(8,10), MM, R
```

declare C1 and Z to be complex (it may not be known yet whether they are scalars, arrays, or functions); declare ALPHA to be a real 8x10 array; and declare MM and R to be real. R would be real anyway, by the IJKLMN rule, but can be declared if desired. Declaring the type of a name does not affect unrelated attributes, such as whether it is a scalar, array, or function. For example, the name MM in the above example could also appear, either before or after the REAL statement, in a DIMENSION statement or an EXTERNAL statement.

There are a number of library functions that have a special type that is known to the compiler (e.g. ABS is real). If you should declare a type for one of these names, it will no longer be recognized as a special name.

Other examples of type statements:

```
INTEGER COUNT, P, DAY OF MONTH
REAL GEORGE(19,65), THING(12), ESTATE(50,135), MC COY
DOUBLE PRECISION X, DRATE, DTIME
LOGICAL L1, L2, TRUTH(0:10)
```

If you need to convert a whole program from single precision to double precision, you may not need a whole string of DOUBLE PRECISION statements. The ADP (Automatic Double Precision) option, described in chapter 9, is designed to do that for you.

ALLOCATION OF VARIABLES

Normally the compiler chooses where to allocate variables. It allocates the arrays first, then the scalars. These come at the beginning of the program, ahead of the object program instructions (see appendix B). Two methods of controlling the allocation of variables are available to you. You can move some of the variables out of the local area into a COMMON area that is shared with other programs, using the COMMON statement. Or, within either the local or COMMON area, you can overlap some variables on top of others or cause them to be in a certain order, using the EQUIVALENCE statement.

To take advantage of these features, you may have to know the amount of storage occupied by each type of variable. In the case of arrays, this is the size of each element of the array. If the ANSI allocation option is specified (see chapter 9), the size of integer and logical variables is different, as shown.

Type	Size in Words
Integer	1 (2 if ANSI)
Real	2
Double Precision	4
Complex	4
Logical	1 (2 if ANSI)

COMMON STATEMENT

The COMMON statement assigns variables to a special storage area that can be shared by more than one program. In earlier FORTRANs, there was only one COMMON area. Later the capability was added of defining additional COMMON areas and giving them names. These are called labeled COMMON areas, so the original COMMON is called blank COMMON, since it has no name. Blank COMMON remains the more often used, but both have some advantages.

Blank COMMON

Variables are usually declared in blank COMMON with a statement of the form:

COMMON v_1 , v_2 , v_3 , ...

where v is the name of a variable (scalar or array) or is an array declaration (array name followed by dimensions). When an array declaration appears, it need not appear in a DIMENSION statement.

This causes the variables named to be allocated in blank COMMON, in the order listed, i.e. v_1 first, then v_2, etc. If there is more than one blank COMMON statement, the variable lists are strung together as if they had all been declared in one statement. In other words, each COMMON statement picks up where the previous one left off.

Blank COMMON begins at the same place for all programs that are loaded together, so if two or more programs want to use the same variables, they should declare them in COMMON in the same order. For example, if both programs have the statement:

```
COMMON CAUSE, LAW, GHIA(70)
```

then they can pass information back and forth in the variables CAUSE and LAW and the array GHIA. The variables must be in the same order, however, since it is the location within COMMON that is important, not the names of the variables. In fact, it is not necessary for the names to be the same, except that it makes it easier to remember what corresponds to what. For example, another program could have the statement:

```
COMMON SENSE, MARKET, THIEF(50)
```

causing SENSE to correspond to CAUSE, MARKET to LAW, and THIEF to the first 50 elements of GHIA. This points out two things. One is that the sizes of blank COMMON do not have to be the same. Whatever corresponds, corresponds; whatever is left over, does not. For example, the last 20 elements of GHIA in the upper program do not correspond to anything in the lower program. The other point is that you have to be very careful about the sizes of various types of variables, so that they really do match up. If you make a real variable correspond to an integer one, two things will happen. They will not be able to pass information back and forth in any straightforward way, because the values are expressed in quite different formats. And the variables that follow in COMMON will not line up, because the real variable occupies two words, while the integer occupies only one word. The only cross-type correspondence that is really recommended is complex to two reals.

There is an exception to this rule about making types agree. Sometimes COMMON is used, not to pass information back and forth, but simply to conserve memory by using the same locations for two sets of variables. If the variables are used only temporarily by each program, so that it does not matter if other programs destroy them, then several programs can use the same COMMON area for their variables, without regard to whether they match up or not. This is a less frequent use of COMMON.

As an example to show how COMMON is arranged in memory, the following shows how two COMMON statements (in two different programs) would arrange the variables, beginning at relative location zero in blank COMMON:

COMMON X, Y, Z, N		COMMON A(3), J	
0000	X	0000	A(1)
0001		0001	
0002	Y	0002	A(2)
0003		0003	
0004	Z	0004	A(3)
0005		0005	
0006	N	0006	J

There is one restriction on variables in blank COMMON. They cannot be initialized with the DATA statement (described later in this chapter).

Blank COMMON can also be declared using a special form of the labeled COMMON declaration, with the name blank, as shown below.

Labeled COMMON

Labeled COMMON makes it possible to have more than one COMMON area. For example, program A might have some data that it shares with program B but not with program C, and some other data that it shares with C but not with B. Programs D and E, then, might share some data with each other but not with A, B, or C. The usual technique, when using only blank COMMON, is to put all the data in blank COMMON, and then each program has to keep track of where the data it needs is. Generally this is done by "gang punching" the same set of COMMON statements and putting them at the head of each program. Using labeled COMMON can cut down on the amount of superfluous data that has to be declared in each program. Also, labeled COMMON variables can be initialized with the DATA statement, whereas blank COMMON variables cannot.

NOTE

> Labeled COMMON block names may also be used within the same program as names of variables, without conflict. Any usage of the label other than in a COMMON declaration will be assumed by the compiler to refer to a variable, and not the COMMON block.

Labeled COMMON is declared in much the same way as blank COMMON, except that each group of variables is preceded by the name of the labeled COMMON block, enclosed in slashes. That is:

COMMON /block name/v_1,v_2,.../block name/v_1,v_2,... etc.

For example, the situation described above, with the five programs A, B, C, D, and E, might be handled with three labeled COMMON blocks, as shown here:

```
Program A:      COMMON /AB/DICK,HIVE /AC/DC,LU,GULL
Program B:      COMMON /AB/DICK,HIVE
Program C:      COMMON /AC/DC,LU,GULL
Program D:      COMMON /DE/W9OML,K
Program E:      COMMON /DE/W9OML,K
```

Each program needs to define only the data that it wants to share with any other programs. A block of COMMON may be used by any number of programs. As with blank COMMON, if the same block is declared more than once in the same program, the variables are strung out into a single list, in the order they appeared. In other words, each reference to the same block picks up where the previous one left off (in a single program). Thus the statements:

```
COMMON /BLOCK1/P,Q,R /BLOCK2/S,T
COMMON /BLOCK2/U,V,W /BLOCK1/X,Y,Z
```

are equivalent to the single statement:

```
COMMON /BLOCK1/P,Q,R,X,Y,Z /BLOCK2/S,T,U,V,W
```

The size of a labeled COMMON block must be the same in all programs that use it. (This is different from blank COMMON.) It is a particularly good idea, therefore, to use exactly the same COMMON statements, with all the variables having the same names. This is not necessary, but it makes it easier to assure the same size.

Labeled COMMON blocks are named with the same kind of names as variables, functions, etc., i.e. beginning with a letter, containing letters and digits, and the first six characters significant. A COMMON block must not have the same name as a subprogram or any other COMMON block, in order to avoid conflicts during loading.

Blank COMMON can be specified using the same form as for labeled COMMON, but with the name (between the slashes) blank. This means that blank and labeled COMMON may be intermixed in the same statement. If the blank COMMON declaration comes first, the slashes may be omitted too, so that it looks just like the form shown above for blank COMMON. For example, the statements:

```
COMMON /ALPHA/ A, B   //  C, D
COMMON // E,F  /ALPHA/  G
```

are equivalent to:

```
COMMON C,D,E,F  /ALPHA/A,B,G
```

Each labeled COMMON block is arranged in the same way as blank COMMON, with the variables following one another in the order listed, starting at a low memory address and moving to a higher memory address.

In some FORTRANs, variables in labeled COMMON may only be initialized (with the DATA statement) in a special program, called a BLOCK DATA subprogram. This is not necessary in Computer Automation FORTRAN IV. Any program may initialize labeled COMMON. The BLOCK DATA subprogram is accepted for compatibility, however (see chapter 7).

EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to make two or more variables occupy the same location or set of locations. It is written:

EQUIVALENCE set_1 , set_2 , set_3 , ...

where set is an equivalence set of the form:

$(v_1, v_2, \ldots, v_n)$

This says that the variables v_1 through v_n are to occupy (or begin at) the same location. Each variable (v) may be one of the following:

1. The name of a scalar variable or an array. When an array name appears, it means that the first element of the array will occupy that location. The other elements will follow. For example:

```
DIMENSION MATRIX(11,11)
EQUIVALENCE(X,Y), (M1,MATRIX)
```

   determines that X and Y will lie in the same location, and that M1 will coincide with the first element of MATRIX, i.e. MATRIX(1,1). An array must be dimensioned before appearing in an EQUIVALENCE statement.

2. An array element, where the subscripts are signed or unsigned integers. For example, the statements:

```
DIMENSION MCOL7(11), MATRIX(11,11)
EQUIVALENCE (MID,MATRIX(5,5)) , (MCOL7,MATRIX(1,7))
```

   would allocate the scalar MID in the middle of the array MATRIX,

coinciding with MATRIX (5,5), and would cause the array MCOL7 to overlay the seventh column of MATRIX, by defining its starting location to be the same as MATRIX(1,7).

3. A scalar or array name followed by a position count enclosed in parentheses. This has the same meaning as if the variable were a one-dimensional array with a normal lower bound of 1. In other words, X(1) means the same as X, X(2) is the element position immediately after X, X(3) is the next, and so on. Thus it is not a count of how many positions away from the variable; it is one less than that. X(3) means 2 positions after X. By element positions, we do not necessarily mean words. We mean steps of the number of words occupied by the variable, depending on its type. In other words, for integer variables the position count is in one-word increments. For real, the increment is two words, for double precision four, etc. This is consistent with the statement above that the variable is treated as if it were a one-dimensional array. For example:

```
EQUIVALENCE (Z,Y(2),X(3))
```

allocates X, Y, and Z one after the other in that order, even though they each require two words.

For arrays, there is a potential conflict between a position count and a subscript. If the array ALPHA has more than one dimension, then ALPHA(5) is clearly a position count. But what if ALPHA has only one dimension? Is the 5 a subscript or a position count? The answer is that it is a subscript, if it makes any difference. Usually it does not. If ALPHA has a normal lower bound of 1, then ALPHA(5) means the same thing either way. (That is why it was defined that way.) However, if ALPHA has a different lower bound, for example:

```
DIMENSION ALPHA(-3:12)
```

then ALPHA(5) means the same as it would in an expression, namely the ninth element of ALPHA, not the fifth (which would be ALPHA(1)).

You should be very careful in equivalencing variables of different types to each other. For one thing, the sizes may be different. More importantly, if you intend to pass information back and forth, you have to know what you are doing. EQUIVALENCE is not the same as an assignment statement -- types will not be converted. If you were to write:

```
EQUIVALENCE (K,X)
X = 4.38
OUTPUT K
```

the value of K would be whatever was in the first word of the two-word floating point value, and this depends on the particular computer's format for floating point numbers. This is not, in general, a very safe kind of thing to do.

INTERACTIONS OF COMMON AND EQUIVALENCE

An allocation statement must not cause conflicts with any previous allocation statements. This means, for instance, that you can not put the same variable into COMMON twice, nor equivalence two variables that are both already in COMMON.

You may, however, equivalence an unallocated variable to something in COMMON, thus causing that variable to be allocated in COMMON too. For example:

```
COMMON A, B, C, D
EQUIVALENCE (C,Y,X(2))
```

puts Y into blank COMMON coinciding with C, and X(2) i.e., X(1) coincides with B. This could be written more clearly as:

```
EQUIVALENCE (C,Y) , (B,X)
```

An EQUIVALENCE never changes the order of variables already in COMMON. Those are fixed by the COMMON statement. EQUIVALENCEs may simply overlay these variables with others.

Equivalencing an array into COMMON (or using a position count) may increase the size of that COMMON area. This is permissible if it extends COMMON at the end, i.e. beyond the last position currently included. It is not permissible to extend COMMON backwards, i.e. ahead of the first position in COMMON. For example, given the statements:

```
COMMON /BLK/ I, J, K
DIMENSION L(4)
```

this EQUIVALENCE causes a legitimate extension of the COMMON block, as shown:

```
EQUIVALENCE (I,L(1))

0000  I     L(1)
0001  J     L(2)
0002  K     L(3)
0003        L(4)   Legal extension.
```

However, the following EQUIVALENCE tries to extend the block in the other direction:

```
EQUIVALENCE (K,L(4))

            L(1)   ---Illegal Extension
0000  I     L(2)
0001  J     L(3)
0002  K     L(4)
```

This same rule applies to both blank and labeled COMMON. Note that if a labeled COMMON block is extended by EQUIVALENCE, the resulting size must be the same as the size declared in all other programs.

EXTERNAL STATEMENT

Form: EXTERNAL $s_1, s_2, s_3, \ldots$

where: s is the name of an external subprogram.

The EXTERNAL statement declares that the names listed are closed, external subprograms. It is not a statement that is needed very often, because most subprograms can be recognized as such by their usage in the program. For example, in:

```
AB = F(X)
CALL FROG(Y)
```

the names F and FROG are automatically classified as subprograms. The EXTERNAL statement has two special uses:

1. The name of one subprogram can be passed as an argument to another. For example:

```
CALL TEST(F,FROG)
```

If F and FROG had already appeared in statements such as the two shown above, and were known to be subprogram names, there would be no problem. However, if this was the first appearance of F or FROG, there would be no way to know that they were supposed to be subprogram names: -- the compiler would implicitly classify them as scalar variables. So the EXTERNAL statement would be needed here to declare those two names. For safety, it is not a bad idea to always use an EXTERNAL declaration in such cases. Some FORTRANs require this.

NOTE

The EXTERNAL statement is not required for references to known subprograms (basic external functions) such as SQRT (e.g., X = SQRT (Y)); however, it is required when a known subprogram is used as an argument, if not previously referenced as a subprogram.

2. There are library functions whose names are specially recognized by the compiler. For example, it knows that ABS is real and has one argument, and that CMPLX is complex and has two arguments. Some of these functions the compiler generates "in-line"; it does not call an external routine. If you want to use one of these "intrinsic" names for an external routine of your own choosing, you have to first declare it in an EXTERNAL statement. This makes the compiler forget what it knows about the name and treat it like any other external subprogram. For example:

```
EXTERNAL FLOAT
RATE = FLOAT(BOND)/100
```

This is not, in general, something that we recommend. It may make your program confusing to understand.

DATA STATEMENT

A DATA statement gives initial values to variables. Normally FORTRAN does not guarantee the contents of variables upon loading, so you should not use a variable until it has been assigned a value. If the variable is not going to change, then instead of assigning it with an assignment statement (which takes time and space at run time), you can assign it with a DATA statement, so it will be loaded with a particular value. The DATA statement has the form:

DATA $V_1/C_1/$, $V_2/C_2/$, $V_3/C_3/$, ...

where:

V is a list of variables, separated by commas. This may include scalars, arrays, and array elements.

C is a list of constants, separated by commas. A constant may be repeated several times by preceding it with a count and an asterisk:

n*c

where n is a positive integer and c is a constant.

Missing commas between V/C/ groups will cause a warning diagnostic to be output.

There must be the same number of constants as variables in each group, so that they can be assigned on a one-to-one basis. For example:

```
DATA A,J,B(3)/4.6,-12,0.0/ , TITLE/'ABCD'/
```

has the same effect as:

```
A = 4.6
J = -12
B(3) = 0.0
TITLE = 'ABCD'
```

except that the assignment is done during loading, not during execution.

We do not recommend that you use the DATA statement to initialize variables that are later going to change value, because this makes initialization dependent on loading, and therefore you can not restart the program without reloading it. It is better to use assignment statements for values that are going to change, and the DATA statement for values that are not. This means that the values become, essentially, constants with names. This is useful in several places.

For example, instead of writing out the speed of light as 2.997925E10 at every reference, you can use the statement:

```
DATA C/2.997925E10/
```

and then refer to the value as C. This also simplifies updating the program, in case the speed of light should change.

Of course, you could get the same effect using an assignment statement, with only a small loss in time and space. There are other situations where the loss is more significant. Suppose you want to write an ARCTAN function. You will want to have a table of constants. But how do you build a table of constants? You cannot use subscripts on constants, so you would have to execute a group of assignment statements at the beginning of each calculation, such as:

```
V(0) = 0.0
V(1) =  .1243550
V(2) =  .2449787
etc.
```

which would slow the program down quite a lot. This is an ideal application for the DATA statement, since it takes no space or time during execution, and the values are not going to change.

DATA Variable List

The variable list consists of scalars, arrays, and array elements, separated by commas. When an unsubscripted array name appears, it represents all the elements of the array in storage order (the same as in an input/output list). There must be enough constants to fill up the whole array. It is not possible to initialize part of an array. To do that, you have to write out the individual array elements. (Or you can EQUIVALENCE a smaller array to the part you want to initialize, and use the smaller array in the DATA statement.)

The subscripts used in an array element may only be integer constants. (A variable subscript would not have a value at compile time.)

With one exception, each variable must be initialized by a constant of the same type. The type conversions performed by the assignment statement are not done by the DATA statement. For example, you must write:

DATA X/3./ and not DATA X/3/

The one exception is that any type of variable may be initialized by a hexadecimal or alphanumeric string constant, using as many digits or characters as required. This is described below.

In the DATA statement, a complex variable is treated as a singled entity. This differs from the input/output list, where a complex variable is treated as two real parts. Thus a complex variable should be initialized by a complex constant (i.e. two reals enclosed in parentheses) or by a single hexadecimal or alphanumeric constant. You cannot, for example, initialize the real part in hexadecimal and the imaginary part in floating point.

Dummy variables may not be initialized (since they have no real existence at compile time), nor may variables in blank COMMON (since that area is preempted by the loader). However, you may initialize labeled COMMON, and you may do so in any program. It is not necessary to use a BLOCK DATA subprogram, but you may if you prefer.

If a variable appears in more than one DATA statement in a program, the latest one overrides the previous ones. Similarly, if more than one program initializes a variable in labeled COMMON, the last one loaded will take precedence. We do not recommend this.

DATA Constant List

The constant list may contain constants of any type, including integer, real, double precision, complex, logical, hexadecimal, or alphanumeric string. Numeric constants may be signed or unsigned. Any constant may be repeated n times, using the form n*c, where n is greater than zero. For example:

```
DATA A,B,C,D,E /2*-3E7,3*'CDE'/
```

The total number of constants (including repetitions) must be the same as the total number of variables (including all the elements of unsubscripted arrays), except in the case of alphanumeric strings. One string constant can act as several constants, as described below. However, two string constants cannot act as one.

These are the rules for using the various types of constants:

1. An integer, real, double precision, complex, or logical constant must correspond to a variable of the same type. Note that you must write the "D" exponent on a constant that initializes a double precision variable.

2. A hexadecimal constant may initialize a variable of any type. It does so in a manner similar to Z format input. The constant may have as many digits as are required by the variable type (i.e. four for integer, eight for real, etc.). It may not have more. If it has fewer, they are right-justified in the variable. Since a complex variable is handled as one value, it may accept up to sixteen hexadecimal digits. If it finds fewer than nine digits, the real part will be zero. Here is an example using hexadecimal constants:

```
COMPLEX CPX
DATA J,CPX /3ZA80,:4E832FB0CE805EE7/
```

3. Alphanumeric strings may also initialize any type of variable, but they differ in several ways from the other constants. For one thing, blanks are significant within the strings. Also, when there are fewer characters than needed to fill a variable, they are left-justified and followed by blanks to fill out the whole variable. The most important difference, however, is that a string constant can initialize more than one variable. If it contains more characters than needed by the first variable, it goes on to the next, and keeps going until it runs out of characters. If there are not enough characters to completely fill the last variable, it is filled out with blanks. Usually this feature is used to initialize arrays, as in:

```
INTEGER LC(20)
DATA LC/'THE WEED OF CRIME BEARS BITTER FRUIT.    '/
```

(Note that some extra spaces were needed at the end of the string to provide for all the elements of the array.) It is not necessary to use an array, however. In fact, the string could fill up a variety of variables of different types, if such a thing were needed. For example:

```
COMPLEX C1
DATA X,M,C1/9HABCDEFGHI/
```

is equivalent to:

```
DATA X/4HABCD/, M/2HEF/, C1/8HGHIbbbbb/
```

On the other hand, one variable may not be initialized by more than one string constant. If the first constant is not long enough, the rest of the variable is filled out with blanks. Thus:

```
DATA X /4HABAB/
```

is not the same as either:

```
DATA X /2HAB,2HAB/        -or-    DATA X /2*2HAB/
```

In the first case, X is assigned the four characters 'ABAB'. The latter two cases both assign 'AB ' to X and have 'AB' left over (which is an error).

CHAPTER 7

PROGRAMS AND SUBPROGRAMS

When a FORTRAN program is loaded and executed, it may consist of several different kinds of units. There must be one, and only one, main program. There may be subroutines, functions, and tasks written in FORTRAN. There will be system routines and functions provided from the library. There may also be other subprograms or tasks that you write in assembly language.

MAIN PROGRAMS

A main program is any program that does not begin (except for comment lines) with one of the following statements:

FUNCTION
SUBROUTINE
TASK
BLOCK DATA

Since those statements always have to come first, a main program may not contain any of them, nor may it contain a RETURN statement.

The starting location of a main program is defined as F:MAIN, and execution always begins there. Thus if there were two main programs, there would be a double definition of F:MAIN. You can write a main program in assembly language, by defining the first location as F:MAIN and using that as the transfer address (operand of the END line).

TASKS

A task is a program that you connect to a real time interrupt. The first statement must be a TASK statement, which is written:

TASK name

where name is a standard FORTRAN name, just like a subprogram.

A task is not the same as a subprogram, because it is not called in the usual way, and because it exits with a STOP statement, rather than a RETURN. It also has no arguments. On the other hand, it differs from a main program in that there may be several tasks with various names, and they do not specify a transfer address to begin execution.

A task may have its own local storage and may also use variables in COMMON to communicate with other programs. Since a task is not a subprogram, however, the local storage is not protected if the task is re-entered. Since tasks are usually executed under RTX, they should be compiled with the RTX option. For further information, see chapter 9.

SUBPROGRAMS

Subprograms are programs that may be called by other programs. A subprogram is either a function or a subroutine. Functions are referenced as elements of an expression, and return a value. Subroutines are referenced with the CALL statement, and do not return a value (except possible indirectly). These two classes can be broken down further, as follows:

Functions

1. FUNCTION subprograms
2. Statement functions
3. Intrinsic functions
4. Basic external functions
5. Assembly language functions

Subroutines

1. SUBROUTINE subprograms
2. Assembly language subroutines

FUNCTIONs and SUBROUTINEs are complete programs, written in FORTRAN. Statement functions are defined in a single statement, and may be included within any FORTRAN program. A basic external function is an assembly language function (usually), in the library, whose name and attributes are known to the compiler. An intrinsic function is also a library function known to the compiler, but it is not a closed external routine. It is generated in-line by the compiler. (I.e. it is like an assembly language "macro".) The library functions are listed and described later in this chapter. Except for intrinsic functions, all of these are called with a standard calling sequence. In many cases, it is not necessary for the compiler to know what kind of subprogram is actually going to satisfy a reference. For example, in:

```
A = F(X)
```

the function F might be a FUNCTION subprogram, a statement function, or an assembly language function; it does not affect the way the statement is generated.

FUNCTION Subprograms

A FUNCTION is a subprogram whose primary purpose is to compute a value and return it to the calling program. It must begin with a FUNCTION statement, which can be written in either of the following ways:

FUNCTION f(d_1,d_2,...,d_n)
type FUNCTION f(d_1,d_2,...,d_n)

where:

- f is the name of the function
- d is the name of a dummy, which corresponds to one of the arguments in the calling reference. See "Arguments and Dummies", below.
- type is one of the type specifications, namely INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

The type of the function name determines the type of the result that is returned. If no type is specified, the IJKLMN rule will apply.

A dummy is named with a regular FORTRAN name. Within the subprogram, it is classified as a scalar, array, or subprogram name, and should correspond to a similar entity in the calling program. Most dummies are simple scalar variables.

Other programs reference the name f as a function. Within the function itself, however, the name f is treated as a scalar variable. This is the variable whose value is returned as the result of the function. Therefore you should always assign it a value before executing the RETURN statement. For example:

```
FUNCTION SQ(X)
SQ = X ** 2
IF (X<0) SQ = -SQ
RETURN
END
```

A FUNCTION must always have at least one dummy. Normally, function dummies are "input" values, and are not changed within the program. However, if the corresponding argument is a variable (and not an expression, constant, or subprogram name), it is permissible for a function to store values back into it by assigning values to the dummy.

If dummies are to have other than implicit type (IJKLMN rule), they must be declared in a type statement. The type modifier attached to the FUNCTION statement does not apply to the dummies, only to the FUNCTION name. For example:

```
DOUBLE PRECISION FUNCTION POLY(RAD,N)
DOUBLE PRECISION RAD, PI
DATA PI/3.14159265358979324/
IF (N<3 .OR. RAD .LE. 0) STOP 100
POLY = 2 * N * RAD * DSIN(PI/N)
RETURN
END
```

SUBROUTINE Subprograms

A SUBROUTINE is a subprogram whose primary purpose is not to compute and return a single value. Usually it performs more complicated operations, such as input/output, matrix manipulation, or other blocks of computation. When a large program is broken into modular units, the units are mostly SUBROUTINEs. A subroutine is referenced by the CALL statement, rather than in an expression. It must begin with a SUBROUTINE statement, of the form:

SUBROUTINE sub(d_1,d_2,...,d_n)

or:

SUBROUTINE sub

where: sub is the name of the subroutine.
d is the name of a dummy.

Note that a SUBROUTINE is permitted to have no dummies, while a FUNCTION must have at least one. The rules for dummies of SUBROUTINEs and FUNCTIONs are exactly the same, and are described in the section "Arguments and Dummies", later in this chapter.

The last statement executed in a SUBROUTINE should be a RETURN statement. This simply returns control to the statement following the CALL -- it does not return a value. However, SUBROUTINEs often return values indirectly by storing them either in COMMON or in the dummies. When a dummy is assigned a value, the corresponding argument must be a variable (scalar, array, or array element).

Example of a SUBROUTINE:

```
SUBROUTINE PRINT(VOLTS,NR)
OUTPUT 'VOLTAGE = ' , VOLTS , 'TEST NUMBER:' , NR
NR = NR + 1
RETURN
END
```

Statement Functions

A statement function is similar to a FUNCTION subprogram, in that it computes and returns a value. However, instead of being a separate program, it is defined in a single statement, called a statement function definition, and can be included within any other program, whether main program or subprogram. It is written:

$f(d_1,d_2,\ldots,d_n) = exp$

where:
- f is the name of the statement function.
- d is the name of a dummy scalar variable.
- exp is the expression that defines the value of the function.

The type of the function is determined by the IJKLMN rule unless f has appeared in a type statement. The expression must have a type that can legally be assigned to the type of the function. The rules for this are the same as for assignment statements (see table 3-1). The expression should contain at least one reference to each of the dummies. It may also reference other variables, arrays, and functions in the program, including other statement functions that have been defined previously. A statement function may not reference itself.

A statement function must have at least one dummy. FORTRAN allows these dummies to have the same name as any other quantity in the program, except for the other dummies of that statement function. However, less confusion results from using distinct names for the dummies.

Statement functions must precede all the executable statements in a program, and must follow most of the declaration statements. See appendix A.

Examples of statement functions:

```
RSQ(A,B) = SQRT(A**2+B**2)
F(X) = 1/X - 3/X**3 + BASE
INC(K) = MATRIX(K+1) - MATRIX(K)
```

BLOCK DATA Subprograms

A BLOCK DATA subprogram is a special program unit that may be used to initialize variables in labeled COMMON. It has no name and generates no object code. It begins with the statement:

```
BLOCK DATA
```

and may contain only declaration statements and an END statement. In particular it should contain COMMON and DATA statements to perform the initialization.

Some other FORTRANs require a BLOCK DATA subprogram to initialize labeled COMMON. Computer Automation's FORTRAN does not -- labeled COMMON may be initialized in any program. However, BLOCK DATA is provided for compatibility.

Be careful to declare each COMMON block completely (listing all the variables, not just those that are going to be initialized), so that the variables will be in the right position to correspond with the declarations in other programs, and so that the size of the block will also correspond.

For example:

```
BLOCK DATA
COMPLEX C1, C2
REAL MAT (35)
COMMON/BETA/C1,C2,VAL,KNUM,ARRAY (10,10),MAT
DATA VAL/.57721/(0.,1.),(0.,-1.)/
END
```

Since a BLOCK DATA subprogram has no name, it must either be compiled along with the main program in batch mode (so that it is automatically included during linking), or the module which includes it must be linked unconditionally; otherwise it will be left out of the linking process, since no other module will have referenced it as an external module.

ARGUMENTS AND DUMMIES

Correspondence

The quantities passed to a function or subroutine when it is referenced are called arguments. The subprogram must provide the same number of names by which to identify the arguments. These are called dummies. They are formal parameters and have no real existence of their own. A reference to a dummy is actually a reference to the corresponding argument. The dummy list in a subprogram indicates the number, order, and type of the arguments.

An argument may be any of the following:

1. A scalar variable
2. An array element
3. An array name (unsubscripted)
4. An expression
5. An alphanumeric string constant
6. A subprogram name (with no arguments)

Note that a single element, such as a constant or a function reference, is considered an expression. On the other hand, although a scalar or array element is also a simple expression, these must be considered separately. This is because a subprogram can store values back into a scalar or array element, but it may not store into a constant or function reference or other expression. An unsubscripted array name is the same as the first element of the array.

The address passed for an alphanumeric string is that of the first word (i.e. first two characters). The word preceding this always contains the character count, identifying how many characters are in the string.

A dummy is always specified as a name. It may be classified, within the subprogram, as any of the following:

1. A scalar variable
2. An array
3. A subprogram

This classification takes place using the same rules for implicit and explicit declarations as apply to other names (see chapter 6). In general, the type of a dummy must be the same as the type of the corresponding argument. For example, the following is incorrect, because the types do not match:

```
COMPLEX Z                SUBROUTINE SUB(M,IMP)
CALL SUB(Z,J)            COMPLEX IMP
```

If either of the arguments or the dummies were reversed, the types would match properly

There is one case where the types do not have to match. An alphanumeric string argument has no type and may correspond to a dummy of any type (though integer is recommended). A SUBROUTINE name also has no type, but should correspond to another SUBROUTINE name.

Table 7-1 below shows the permissible kinds of correspondence between an argument and a dummy:

Table 7-1. Permissible Argument/Dummy Correspondence

Argument	Dummy			
	scalar	array	stored into	subprogram
scalar or array element	yes	(yes)	yes	no
array name	(yes)	yes	yes	no
alphanumeric string	(yes)	yes	no	no
expression	yes	no	no	no
subprogram name	no	no	no	yes

The correspondences marked "(yes)" are permitted, but may or may not be particularly useful. This will be discussed further below, under "Dummy Arrays".

When a dummy corresponds to a variable (scalar or array) in the argument list, every reference to the dummy is actually a reference to the argument variable. Thus not only will the dummy initially have the value of the argument variable, but if the dummy is changed, the argument variable is changed too. This is a way for both functions and subroutines (mostly subroutines) to return results through the argument list.

For example:

```
CALL TRIG(A,SINA,COSA,SINHA)
    •
    •
    •
```

```
SUBROUTINE TRIG(X,SX,CX,SHX)
SX = SIN(X)
CX = COS(X)
SHX = TANH(X)
SHX = SHX / SQRT(1-SHX**2)
RETURN
END
```

On the other hand, when a dummy corresponds to an expression (or constant), the latter acts only as an "input" value for the dummy. The dummy must not be changed. For example, if X is a scalar variable and F is a function:

```
CALL GRUNCH(X,2.5,F(X),F)
```

```
SUBROUTINE GRUNCH(A,B,C,D)
```

then A may be stored into, the others may not. A, B, and C should be dummy scalars, while D should be a dummy subprogram.

CAUTION

Storing into improper dummies is not detected as an error, due to the large overhead it would require at run time. Therefore, be aware of this possibility, since it can cause strange things to happen to your program (like changing the value of constants that need to be used subsequently).

Since a dummy has no real existence on its own, it may not be allocated or initialized. That is it may not appear in a COMMON, EQUIVALENCE, or DATA statement.

Dummy Arrays

A dummy is an array if it is dimensioned in the subprogram. Normally the calling argument is also an array, or else an alphanumeric string. As with all dummies, a dummy array does not actually occupy any memory -- it just identifies an area in the calling program. The subprogram assumes that the argument passed to it is the address of the first element of an array, and it calculates subscripts from there. Of course it has no way of knowing what the dimensions of the argument array are, so you have to be sure to give the dummy array appropriate dimensions. Usually this means the same dimensions as the argument array, but occasionally it can be useful to use different dimensions. For example:

```
DIMENSION EDGAR(10,10)          SUBROUTINE SUB(SNERD)
CALL SUB(EDGAR)                 DIMENSION SNERD(5,4)
```

Here the dummy, SNERD, is much smaller than the argument, EDGAR. This will cause the subroutine to treat the first two columns of EDGAR as if they were a 5x4 array. If the CALL had said:

```
CALL SUB(EDGAR(1,8))
```

then SNERD would represent the eighth and ninth columns of EDGAR, instead of the first and second.

It is also possible for the calling program to tell the subprogram what dimensions to use on a dummy array. This is described in the following section.

When an alphanumeric string is passed as an argument, it is usually received by a dummy array. For example, in this situation:

```
CALL FOR('PHILIP MORRIS')       SUBROUTINE FOR(JY)
                                INTEGER JY(8)
```

The first seven elements of JY correspond as follows:

```
JY(1) = 'PH'        JY(2) = 'IL'
JY(3) = 'IP'        JY(4) = 'bM'
JY(5) = 'OR'        JY(6) = 'RI'
JY(7) = 'Sb'
```

Note that the character string has an extra blank, if necessary, to fill up the last word. The positions beyond this, however, are undefined, so JY(8) should not be used. Also, since an alphanumeric string (when used as an argument) is filled out only to the nearest word boundary, if the dummy array is any type but integer, there may be elements that are only partly defined. For example, if JY were double precision, the first element would contain a full eight characters, but the second element would contain only six. The last two characters would be unpredictable. This makes it a good idea to use integer arrays for alphanumeric strings.

An alphanumeric string is stored in memory as a string of characters, preceded by an integer count of those characters. The address passed as the argument, however, is that of the first two characters. The count is primarily intended to be used by assembly language subprograms, but it can be accessed in a FORTRAN program, if you use an out-of-range subscript (i.e. one less than the lower bound of the array). In the above example, JY(0) would contain the character count. The compiler will let you do this. The dummy array must be integer to access the character count.

Table 7-1 showed some argument/dummy correspondences marked "(yes)", which need some clarification. If a dummy array corresponds to a scalar, that means the first element of the array corresponds to the scalar. The other elements will correspond to whatever follows the scalar. This will be unpredictable, unless you use EQUIVALENCE on the scalar to make sure something meaningful follows it.

On the other hand, if the argument is an array (or an alphanumeric string), and the dummy is a scalar, then you will only be able to access the first element of the array (or the first few characters in the string), since dummies cannot be equivalenced. In this case it would be better to specify the first element of the array (or a shorter string), to make it clearer what you are doing.

Adjustable Dimensions

A dummy array occupies no actual storage. Its dimensions are used only to locate its elements, not to allocate storage for them. Therefore, it is not necessary for the subprogram to know what the dimensions are at compile time. The dimensions may also be passed along as arguments. This means that any of the dimensions of a dummy array may be specified by other dummies that are integer scalars. Thus the calling programs can change the dimensions for each call. For example, you might call a matrix multiplication subroutine with the following arguments:

```
DIMENSION A(5,8) , B(8,10) , C(5,10)
CALL MATMPY(A,B,5,8,10,C)
```

and the subroutine could be written like this:

```
SUBROUTINE MATMPY(A,B,J1,J2K1,K2,C)
DIMENSION A(J1,J2K1), , B(J2K1,K2) , C(J1,K2)
DO 2 K = 1,K2
DO 2 J = 1,J1
C(J,K) = 0
Do 2 JK = 1,J2K1
2       C(J,K) = C(J,K) + A(J,JK)*B(JK,K)
RETURN
END
```

Compare this with the example shown at the end of "DO Loop Ranges", in chapter 4.

Of course, when we say that a calling program can change the dimensions for each call, we mean only that the subroutine can be made to handle separate arrays of differing dimensions. This does not mean that the same array should be described with different dimensions in subsequent calls. If this is done, then the row/column relationship of the dummy array won't match that of the actual array.

If a dummy array has both lower and upper bounds specified, either or both may be adjustable. For example:

```
FUNCTION GAMMA (MM,J,N)
DIMENSION MM (0:N,J:N)
```

The dummies used as adjustable dimensions may be referenced elsewhere in the subprogram, but they may not be changed. The dimensions must be determined once and for all at the beginning of the subprogram. However, each call can supply different dimensions.

Dummy Subprograms

A dummy subprogram may only correspond to an argument that is a subprogram name, and it is the only kind of dummy that may do so. A call on the dummy subprogram is actually a call on the argument subprogram.

For example, the function COMPARE, below, could be used to compare the single and double precision versions of other functions and return the difference:

```
EXTERNAL ALOG,DLOG,EXP,DEXP
DOUBLE PRECISION DLOG,DEXP
A = COMPARE(ALOG,DLOG,X)
B = COMPARE(EXP,DEXP,Y)
```

```
FUNCTION COMPARE(F,DF,RV)
DOUBLE PRECISION DF,DV
DV = RV
COMPARE = DABS(DF(DV)-F(RV))
```

Note that the library routines had to be declared EXTERNAL in order to pass them as arguments. This caused them to lose their special type, so the double precision ones had to have their type declared too. The real ones did not have to, because the IJKLMN rule gave them the correct type.

LIBRARY FUNCTIONS

FORTRAN includes a number of library functions, which perform calculations such as square root, arc tangent, absolute value, maximum value, inclusive OR, type conversion, etc. These are listed in a table 7-2. When you use one of these in your program, it will automatically be provided, either as a closed routine at load time or as in-line object code at compile time. The names of all of these functions are recognized by the compiler as either basic external functions or intrinsic functions.

Intrinsic and Basic External Functions

Intrinsic and basic external functions are distinguished by the fact that their names are known and recognized by the compiler. There are three reasons for doing this:

1. All library functions return a certain type of result, and this may not be the type that the name would acquire by the IJKLMN rule (e.g. all the double precision and complex functions). Instead of requiring you to declare these if you want to use them, the compiler knows what type each should be.

2. All library functions also accept a certain type and number of arguments. By knowing this information, the compiler can produce an error message for any usage with too many, too few, or wrong type arguments.

3. Some of the functions' operations are so short that it is more efficient to generate the necessary instructions to do them than to call an outside routine.

The thing that differentiates intrinsic and basic external functions is that intrinsic functions are generated in-line, while basic external functions are called in from the library. In other words, steps 1 and 2 above are performed on both kinds of functions, step 3 only on intrinsic functions.

Most of the time, you need not be concerned about any of this. It is all handled automatically. There are only two rare situations where it becomes important: when you want to pass the name of a library function as an argument to another subprogram; or when you want to write your own function with a name that is the same as a library function.

Suppose you want to write your own square root routine and use it instead of the standard SQRT. You can do this, since SQRT is a basic external function and will be called. However, if you tried to write your own IABS function (integer absolute value), it would never be called, because IABS is intrinsic and generated in-line. Also, if you wanted to write some completely unrelated function called SQRT (e.g. Sam's Quick Roster Tabulation, an integer function with three arguments), you would conflict with the compiler's knowledge that SQRT is real with one argument. Both of these problems can be solved by declaring IABS or SQRT in an EXTERNAL statement. When that is done, the compiler forgets everything it knows about the function.

There are other ways besides being declared EXTERNAL that an intrinsic or basic external name can lose its special recognition. If it is used in some other context than as a function reference, it may become a scalar, an array, a dummy, etc. Appearing in a type statement (e.g. INTEGER) also cancels special knowledge of a library function.

A FORTRAN library could consist of any combination of intrinsic, basic external, and ordinary functions. Ordinary functions would work properly (some small FORTRANs only have these), but they just would not be as efficient or give diagnostics on improper arguments. In Computer Automation FORTRAN IV, all of the standard library functions are either basic external or intrinsic, as shown in the table below. You could add other functions to your system library, and those would be ordinary functions.

Table of Library Functions

Table 7-2 lists all of the standard library functions. The first column gives the function name. The X in one of the next two columns indicates whether the function is intrinsic or basic external. The next two columns contain the type of the function (i.e. the type of the result) and the type of the arguments. The following abbreviations are used here:

I	Integer
R	Real
D	Double precision
C	Complex

(There are no library functions with logical type arguments or results.)

The sixth column indicates how many arguments the function expects. The indication N 2 means any number of arguments, but at least two.

The last column explains what the function does. When a formula is shown, it is not necessarily used in evaluating the function but merely serves to help define the operation.

Boolean Operations

Computer Automation FORTRAN IV provides the capability to do some Boolean operations (e.g. masking, merging) on all the bits in a word. You cannot do this with the logical operators (e.g. .AND.,.OR.), because they deal only with the values true and false. There are four library functions that perform the corresponding operations on all sixteen bits. The functions IAND, IOR, and IEOR accept any number of integer arguments and perform AND, inclusive OR, or exclusive OR on them, respectively. The function INOT takes one integer argument and returns the 1's complement of it.

There is no "Boolean" type, so these operations are done in integer. Bit patterns can be established using hexadecimal constants, which are also generally integer.

For example:

```
MASK = IOR(KEY,:F)
IF (IAND(NAME,4ZFF00) .EQ. 4ZC100) GO TO 73
M = IAND( IOR(L,:3A00) , IEOR(M1,M2) , LAST )
```

These functions are intrinsic, i.e. generated in-line, so the object code for them is as good as if they were special operators in the language. However, remember that not only do the Boolean operations depend on a particular computer's word format, but there is very little consistency among FORTRANs about how (or whether) such operations may be specified.

Table 7-2. Library Functions.

Name	Intrinsic	Basic External	Function Type	Argument Type	Number of Arguments	Definition of Function
						Absolute Value
IABS	X		I	I	1	Integer.
ABS	X		R	R	1	Real.
DABS	X		D	D	1	Double precision.
CABS		X	R	C	1	Complex (modulus). This is a real value, namely: $CABS(x+iy) = \sqrt{x^2+y^2}$
						Maximum/Minimum Value
MAX0		X	I	I	N≥2	Integer maximum value of integer arguments.
MAX1		X	I	R	N≥2	Integer maximum value of real arguments.
MIN0		X	I	I	N≥2	Integer minimum value of integer arguments.
MIN1		X	I	R	N≥2	Integer minimum value of real arguments.
AMAX1		X	R	R	N≥2	Real maximum value of real arguments.
AMAX0		X	R	I	N≥2	Real maximum value of integer arguments.
AMIN1		X	R	R	N≥2	Real minimum value of real arguments.
AMIN0		X	R	I	N≥2	Real minimum value of integer arguments.
DMAX1		X	D	D	N≥2	Double precision maximum value of double precision arguments.
DMAX0		X	D	I	N≥2	Double precision maximum value of integer arguments.
DMIN1		X	D	D	N≥2	Double precision minimum value of double precision arguments.
DMIN0		X	D	I	N≥2	Double precision minimum value of integer arguments.
						Modulus (remaindering)
						Arg_1 (mod arg_2), with the sign same as arg_1. Undefined if arg_2 is zero.
MOD		X	I	I	2	Integer. MOD(j,k) = j - k*[j/k] where the brackets indicate integer part.
AMOD		X	R	R	2	Real. AMOD(x,y) = x - y*AINT(x/y).
DMOD		X	D	D	2	Double precision. Same as AMOD.

Table 7-2. Library Functions. (Cont'd.)

Name	Intrinsic	Basic External	Function Type	Argument Type	Number of Arguments	Definition of Function
						Boolean
IAND	X		I	I	N≥2	AND (i.e. extract).
IOR	X		I	I	N≥2	Inclusive OR (i.e. merge).
IEOR	X		I	I	N≥2	Exclusive OR.
INOT	X		I	I	1	NOT (i.e. 1's complement).
						Type Conversion
FLOAT		X	R	I	1	Convert integer to real.
INT		X	I	R	1	Convert real to integer.
IFIX		X	I	R	1	Same as INT.
DFLOAT		X	D	I	1	Convert integer to double precision.
IDINT		X	I	D	1	Convert double precision to integer.
DBLE		X	D	R	1	Convert real to double precision.
SNGL		X	R	D	1	Convert double precision to real.
CMPLX		X	C	R	2	Convert two real values to complex. CMPLX(x,y) = x + iy
REAL		X	R	C	1	Real part of complex value.
AIMAG		X	R	C	1	Imaginary part of complex value.
						Truncation (integer part)
AINT		X	R	R	1	Truncate to integer and back to real.
DINT		X	D	D	1	Truncate to integer and back to double.
						Sign transfer Magnitude of arg_1 with sign of arg_2 . Positive if arg_2 is zero.
ISIGN	X		I	I	2	Integer.
SIGN	X		R	R	2	Real.
DSIGN	X		D	D	2	Double precision.

Table 7-2. Library Functions. (Cont'd)

Name	Intrinsic	Basic External	Function Type	Argument Type	Number of Arguments	Definition of Function
						Positive difference dim(x,y) = x - min(x,y)
IDIM	X		I	I	2	Integer.
DIM	X		R	R	2	Real.
DDIM	X		D	D	2	Double precision.
						Complex conjugate
CONJG		X	C	C	1	CONJG(x+iy) = x - iy
						Square Root
SQRT		X	R	R	1	Real.
DSQRT		X	D	D	1	Double precision.
CSQRT		X	C	C	1	Complex. CSQRT(Z) = u+iv = $e^{(\log Z)/2}$ allocated so that u ≥ 0.
						Logarithm
ALOG		X	R	R	1	Real natural logarithm (base e).
ALOG10		X	R	R	1	Real common logarithm (base 10).
DLOG		X	D	D	1	Double precision natural logarithm.
DLOG10		X	D	D	1	Double precision common logarithm.
CLOG		X	C	C	1	Complex natural logarithm. CLOG(Z) = CLOG(x+iy) = u + iv = log|Z| + i ATAN2(y,x) allocated so that $-\pi < v \leq \pi$.
						Exponential (e^x)
EXP		X	R	R	1	Real.
DEXP		X	D	D	1	Double Precision.
CEXP		X	C	C	1	Complex. CEXP(x+iy) = EXP(x)*(COS(y)+i SIN(y))

Table 7-2. Library Functions. (Cont'd)

Name	Intrinsic	Basic External	Function Type	Argument Type	Number of Arguments	Definition of Function
						Sine (of angle in radians)
SIN		X	R	R	1	Real.
DSIN		X	D	D	1	Double precision.
CSIN		X	C	C	1	Complex. $CSIN(Z) = (e^{iZ} - e^{-iZ})/(2i)$
						Cosine (of angle in radians)
COS		X	R	R	1	Real.
DCOS		X	D	D	1	Double precision.
CCOS		X	C	C	1	Complex. $CCOS(Z) = (e^{iZ} + e^{-iZ})/2$
						Tangent (of angle in radians)
TAN		X	R	R	1	Real.
DTAN		X	D	D	1	Double precision.
						Arctangent (in radians)
						When two arguments, arg_1 = ordinate (y), arg_2 = abscissa (x). Result (R) is the arctangent of y/x, quadrant allocated in the range $-\pi < R \leq \pi$. If both arguments are zero, the result is zero.
ATAN		X	R	R	1	Real, one argument.
ATAN2		X	R	R	2	Real, two arguments (coordinates).
DATAN		X	D	D	1	Double precision, one argument.
DATAN2		X	D	D	2	Double precision, two arguments.
						Hyperbolic Functions
						$\sinh(x) = (e^{x}-e^{-x})/2$
SINH		X	R	R	1	Real.
						$\cosh(x) = (e^{x}+e^{-x})/2$
COSH		X	R	R	1	Real.
						$\tanh(x) = -i \tan(ix) = (e^{x}-e^{-x})/(e^{x}+e^{-x})$
TANH		X	R	R	1	Real.
DTANH		X	D	D	1	Double Precision.

CHAPTER 8

IN-LINE ASSEMBLY LANGUAGE

Computer Automation FORTRAN IV allows assembly language instructions to be used in a FORTRAN program. This does not include all of the features of a full-fledged assembler (not all of the machine instructions or directives are accepted) and you should not think of it as taking the place of the assembler. In general, it is better to write FORTRAN programs in FORTRAN and remove machine language sections to separate subprograms that can be assembled by the assembler and called by the FORTRAN program. This is especially true, of course, if you want to maintain compatibility of the FORTRAN programs from one machine to another.

There are two situations that inline assembly language is primarily intended for:

1. When timing is critical and you want to perform some special short operation that the FORTRAN language does not include. For example, a rotate shift.

2. When memory space or timing is critical, and you want to shorten a program by hand coding some of the statements. For example, knowing exactly what subscripts are used in a DO loop, you might rewrite the loop control and the subscripting more efficiently. This requires considerable familiarity with the object code and addressing techniques, and is kind of a desperation move.

LINE FORMAT

A section of inline assembly language begins following the appearance of the special FORTRAN statement:

```
ASSEMBLER
```

It ends when the assembly directive:

```
FORTRAN
```

is encountered.

Within an assembly language section, the instructions may be written in free-form; column 7 is no longer significant. However, it is probably a good idea to line up your opcodes and operands for better readability. Note that the statement ASSEMBLER is processed in FORTRAN and may not begin before column 7, while the directive FORTRAN is processed in assembly language and so may begin anywhere from column 2 on. However, the FORTRAN directive is a special case; it does not have either an operand field or a comment field. Additionally, the first in-line assembly language statement must not contain a character in column 6; if it does, it will be considered a continuation of the ASSEMBLER statement.

If an assembly line has a label, it must begin in column 1, unless there is an X in column 1 (conditional compilation), in which case the label must begin in column 2. At least one blank must separate the label from the op-code. If there is no label, the op-code may begin in column 2 or later (column 3 or later if there is an X in column 1).

Similarly, there must be at least one blank between the op-code and the operand, and between the operand and a comment, if any. Since blank is a separator, there may not be any blanks embedded in the label, op-code, or operand fields (unless they are part of an alphanumeric operand). Some op-codes do not use an operand, so the field following the op-code is automatically treated as a comment. There are no op-codes that can be used either with or without an operand. If an op-code requires an operand, it does not matter how many blanks must be skipped over to find it.

Here is a sample section of assembly language:

```
                    .
                    .
                    .
                    KEY = J + MASK

                    ASSEMBLER

#12                 LDA     KEY

                    RRA     7

                STA     KEY     THIS IS A COMMENT.

                            FORTRAN

                    CALL FIND(KEY)
                    .
                    .
                    .
```

LABEL FIELD

The labels used on assembly lines are ordinary FORTRAN statement numbers, except that they are written with a preceding number sign, e.g. #45. The reason for the # is that when such a label is used as an operand, it must be possible to distinguish it from a decimal value. This problem does not occur in FORTRAN statements, because labels and values can never appear in the same place. The # should be in column 1 (or column 2 if there is an X in column 1).

You can use the full range of statement numbers, from #1 to #99999, and these are tabulated right along with the labels on FORTRAN statements. A FORTRAN statement may reference an assembly label and vice versa. There may not be any duplicate labels. Note that a FORTRAN statement would reference an assembly label without the preceding #.

There is one other kind of label that is used only on the SET directive for conditional assembly. It is a number preceded by #X and is described in the section on "Conditional Assembly", later in this chapter.

OP-CODE FIELD

The op-code must be separated by at least one blank from the previous field, which may be any of the mnemonics shown in the section "Op-code Classes", below. This includes most of the standard machine instructions, except for input/output, which is not generally safe to do in the middle of a FORTRAN program. It also includes the assembler directives DATA, BAC, TEXT, RES, ENT, STOP, SET, IFT, IFF, ENDC, and LPOOL, as well as the special FORTRAN mnemonic "FORTRAN," and the floating point interpretive op-codes which are described in the section, "Floating Point Interpreter", below.

KINDS OF OPERANDS

A variety of different kinds of operands may be used, depending on the op-code. No one op-code accepts all of them. Before listing these operands, let us set down some of the ground rules that were established about what kinds of things may and may not be referenced.

1. You can reference FORTRAN variables and subprograms. (However, see paragraph 6 below, concerning addressing.) You cannot define them. That is, you cannot use a FORTRAN name in the label field to simulate a SUBROUTINE or FUNCTION statement, or to allocate a variable.

2. You can reference FORTRAN statement labels, by preceding them with #.

3. You can reference external system routines (e.g. floating arithmetic, input/output). All such routines have names that contain a colon (:), so any name with a colon is assumed to be an external system routine. (The name cannot begin with a colon, because that indicates a hexadecimal constant.) These are special names that may be used only on assembly lines, and only via base page pointers (e.g., JST *BP (DELAY:) is correct, but JST DELAY: isn't.)

4. You cannot reference the temps (#T) generated by the compiler, since there is no way of knowing how many the compiler will create. If you want a temp,

you should either use a variable instead, or else define a temp with RES and reference it via a statement label (#n).

5. You also cannot reference FORTRAN constants, including compiler generated indirect addresses. To use a constant, you should define it with DATA and reference it with a regular statement label.

6. A general rule is that in-line assembly operands always reflect what is actually going on in the addressing. The compiler, unlike the assembler, will not generate something different from, or in addition to, what you write. This gives you complete control over the output, but requires you to do some extra work to get it. For example, if the standard assembler encounters:

```
LDA    M
```

and M is not within range, it will change the instruction to an indirect reference through base page and create a word in base page pointing to M. This is not necessarily the way the compiler would address M, so it leaves the decision to you. If something is out of range, you can create an indirect address pointer (using DATA) and reference it with a statement label, or you may insert an LPOOL directive. LPOOL usage is described in the Operating System Assembler Language Reference Manual (96552).

A third alternative is to use a base page pointer (using BP, described below). However, this method has one important limitation: you may not use BP to create a pointer to a forward label (that is, one which occurs further along in the coding sequence), but only to a label which has already been processed by the compiler.

Creating indirect pointers will sometimes be necessary when referencing a FORTRAN variable or statement label, and usually when referencing an external subprogram or system routine. For variables and labels, this will probably require some trial and error, since you may not be able to determine in advance whether a given variable or statement number will be in range -- it depends on what the compiler generates for other statements. If you try to reference something that is out of range, this will be diagnosed as an error, and you can then change it.

7. Normally the compiler decides what things it will allocate base page pointers for, and it tries not to use too many. You can specifically request a base page pointer to be created by using BP(x), where x is the operand. Be careful about creating so many base page words that there is not room for them. Use this mainly on operands that are referenced frequently. Note that what the compiler does with BP is essentially the same as what the assembler does with a preceding =.

The following, then, are the things that may appear in an operand field (given the proper op-code):

1. Indexing, indicated by a preceding @. The operand, in this case, must be an absolute value in the range :0-:FF.

2. Indirect addressing, indicated by a preceding *. If both @ and * are used, they may be in either order.

3. A decimal integer value. In certain cases (e.g. DATA) it may be signed. The permissible size of a value depends on the op-code, as shown in the following section.

4. A hexadecimal value, preceded by : .

5. An alphanumeric value, enclosed in quotes.

6. Blank (no operand field).

7. Current location counter ($), optionally followed by a plus or minus sign and an addend, which is a decimal or hexadecimal value.

8. A statement label (#n), optionally followed by an addend, as above. Do not use an addend on a statement label unless you absolutely have to. It is better programming practice to put another label on the word actually being addressed. An addend on the label of a FORTRAN statement is particularly questionable, since there is no guarantee what the object code will be around that statement.

9. A FORTRAN name, optionally followed by an addend. This may be the name of a scalar, array, external subprogram, or statement function. It may also be the name of a COMMON block if there is no variable in the program with the same name. Note that you cannot reference the entry point of a FUNCTION from within it, because that name is used for the result variable.

10. An external system name, as a base page reference only. Any name with a : in it is automatically assumed to be a system name, and must be provided at load time, either from the library or in a program that you supply.

11. A base page reference, BP(x), where x is a FORTRAN name, a system name, a previously defined statement label, the current location counter, or a value of any type allowed by DATA except an alphanumeric value (see below). Where applicable, this may include an addend (as part of x), except for system names, or other external references. Note that BP of a value puts that value (not a pointer to the value) into the base page.

12. A conditional assembly label (#Xn), described in a later section.

The following section describes which operands may be used with which op-codes.

OP-CODE CLASSES

The op-codes can be divided into thirteen classes, based on the kinds of operands they permit. For each class, we will list the op-codes it includes and the permissible operands. Any operand that is allowed to have an addend may have one. This is not

specifically mentioned in each class.

Note that the classes shown below do not correspond exactly to the instruction classes described in the CAI BETA Assembler or Macro Assembler manuals; in particular, I/O Instructions are not supported in FORTRAN Assembly language.

Those opcodes below which are marked with an asterisk will be executed by emulation on the LSI-3/05 processor (see T3 option, Chapter 9), since they are not valid LSI-3/05 instructions.

Class 1. Memory References

Op-codes:					
	ADD	AND	LDA	JMP	LDR
	ADDB	ANDB	LDAB	JST	LDD
	SUB	IOR	LDX	IMS	LDC
	SUBB	IORB	LDXB	CMS	MPM
		XOR	STA	CMSB	DVM
		XORB	STAB		ADX
			STX		
			STXB		
			EMA		
			EMAB		

The last column contains some of the special mnemonics for use with the floating point interpreter. The rest are in class 6. These are all described in the following section.

Operand: Indexing (@).
Indirect (*).
Decimal or hexadecimal value in the range (0,255).
Current location ($). Relative addressing on this or the next two kinds of operands must be in the range (-255,256).
Statement label (#n).
FORTRAN name, if in relative addressing range.
Base page (BP).

Class 2. Double Word Memory Reference

Op-codes: DVD* MPY* NRM*

These instructions generate a two-word item.

Operand: Same as for DATA (Class 9), except that alphanumeric strings are not allowed.

Operand: Decimal or hexadecimal value in the range (0,255).
Single alphanumeric character ('a').
Base page (BP).

Class 4. Conditional Jump

Op-codes:			
	JAG	JAL	JSS
	JAP	JAM	JSR
	JAZ	JXZ	JOS
	JAN	JXN	JOR

Operand: Current location ($).
Statement label (#n).

Note that only relative addressing is allowed, and it must be in the range (-63,64). The mnemonic JOC is not supported.

Class 5. Shift

Op-codes:				
	ARA	LRA	RRA	LLL
	ARX	LRX	RRX	LLR
	ALA	LLA	RLA	LRL
	ALX	LLX	RLX	LRR

The first three columns are single shifts, the last column contains double shifts.

Operand: Decimal or hexadecimal value in the range (1,8) for single shift, or (1,16) for double shift.

Note that the value is reduced by one when the instruction is generated. In other words, a shift of one looks like a shift of zero in the generated hexadecimal word.

Class 6. Register Change and Control

Op-codes:						
	ZAR	TAX	NAX	ICA	NOP	NEG
	CAR	TXA	NXA	ICX	ENT	ABS
	NAR	EAX	IAX	SBM	ENDC	DIM
	CXR	ANA	IXA	SWM	LPOOL	SGN
	NXR	ANX	IPX	SIA	FORTRAN	ADJ
	SOV	CAX	DAX	SOA		REL
	COV	CXA	DXA	EIN		DBL
	ROV			DIN		CPX
						INT
						XIT
						XNL

ENDC, LPOOL, and FORTRAN are special directives. The last column contains floating point interpretive mnemonics, described in the following section.

Operand: Blank (no operand allowed).

Class 7. SCM and SCMB

Op codes: SCM SCMB

Operand: Must not include indirect or indexing (* or @). Either a base page reference (BP) or a decimal or hexadecimal value in the range (0,255)

Class 8. BAO, BXO, AND SIN

Op-codes: BAO BXO SIN

Operand: Decimal or hexadecimal value in the range (0,15) for BAO and BXO, and (1,7) for SIN.

Class 9. DATA, BAC

Op-codes: DATA BAC

Operand: Indirect (*) on DATA, but not on BAC.
Decimal or hexadecimal value of full word range. Decimal values may be signed.
One or two alphanumeric characters, enclosed in quotes. A single character will be right justified and preceded by binary zeros.

Current location ($). There are no restrictions on relative addressing range.
Statement label (#n).
FORTRAN name.
External system name.
Base page (BP).

Class 10. RES.

Op-codes: RES

There may be either one operand or two separated by a comma. When a second operand is used, RES acts like a multiple word DATA. When the first operand is zero, it acts like EQU $, and there must not be a second operand.

1st operand (word count): Unsigned decimal or hexadecimal value.

2nd operand (fill value): Same as for DATA, in Class 9.

Class 11. TEXT

Op-codes: TEXT

Operand: Any number of alphanumeric characters, enclosed in quotes. A single quote is represented by two quotes. If the number of characters is odd, the last one is left-justified in the word and followed by a blank.

Class 12. SET

Op-codes: SET

This and the following class are conditional assembly mnemonics, described below. A SET directive must have a label in the label field, of the form #Xn.

Operand: The decimal value zero or one.

Class 13. IFT, IFF

Op-codes: IFT IFF

Operand: Conditional assembly label (#Xn).

FLOATING POINT INTERPRETER

Floating point operations at run time are done interpretively, rather than using a separate subroutine call for each. The first thing generated is a call to the interpreter, followed by a sequence of pseudo op-codes. These op-codes have the same instruction format as regular machine instructions, and in fact, some of them are exactly the same mnemonics and generated values as some of the machine instructions. All of these op codes are included in the in-line assembly feature, so that you can make use of them to do floating point operations.

It is not necessary to exit and reenter the interpreter to change mode, e.g. from real to double precision. It is only necessary to do a load of the proper type (e.g. LDR or LDD) or a type conversion command (e.g. REL or DBL). The interpreter then keeps track of what mode it is operating in, and all of the arithmetic operations (e.g. ADD, MPM, STA) automatically operate in that mode.

The floating point equivalent of the A register is the floating point accumulator, which is maintained in base page for efficient operation. During a sequence of floating point operations, the value in the accumulator is kept in an unpacked format that is easier to work with. It is only packed up into the usual floating point format when it has to be stored into a variable or temp. On normal exit from the interpreter (XIT), the contents of the floating accumulator are not guaranteed. If XNL is used, however, the accumulator is preserved (e.g. when returning from a function).

The actual machine A register is always set up when exiting from the interpreter, so that tests can be made on it (e.g. in relation expressions or arithmetic IFs). It is set to a value that is negative, positive, or zero, according to the last value in the floating accumulator. (This is accomplished by merging the sign bit of the floating value with the first 15 bits of the true mantissa, which includes the normalized "1" bit. For complex, both the real and imaginary parts are merged. In this case the sign is meaningless; only zero/non-zero can be tested for.) Thus it is possible to exit from the interpreter and do a JAZ or JAP or JAM etc.

The normal entry into the interpreter is by calling F:RINT. The first op-code should then be one that determines a mode to operate in, i.e. a load or a conversion. (If it is a conversion, it would convert from the integer value in the A register.) For example:

```
JST     *BP(F:RINT)          LDA     K
LDD     DX          -or-     JST     *BP(F:RINT)
                             REL
```

If there is already a value in the floating accumulator (e.g. after returning from a function call), then there are three alternate entries to the interpreter, which automatically set the mode to real, double precision, or complex. These are F:RREL, F:RDBL, and F:RCPX, respectively. For example:

```
JST     *BP(F:RREL)
STA     X
XIT
```

Table 8-2 lists all of the op-codes recognized by the floating point interpreter, and what they do. This includes the mnemonics that are the same as for machine instructions; these are marked with an asterisk.

Table 8-2. Floating-Point Interpreter Op-codes

Op-code	Description
LDR	Load real. Load the two-word quantity addressed, and unpack it into the floating accumulator. Set mode to real.
LDD	Load double. Load four-word quantity, set mode to double precision.
LDC	Load complex. Load four-word quantity, unpack into two real values in the floating accumulator. Set mode to complex.
REL	Convert to real, from whatever mode is currently set. If none has been set, this means assume integer in the A register. Set mode to real.
DBL	Convert to double precision, and set mode to same.
CPX	Convert to complex. This always involves adding an imaginary part of zero. Set mode to complex.
ADD*	Add by mode (i.e. in whatever mode is currently set).
SUB*	Subtract by mode.
MPM	Multiply by mode.
DVM	Divide by mode.
NEG	Negate by mode.
ABS	Absolute value by mode. Does not apply to complex.
DIM	Positive difference by mode. (See DIM and DDIM in Table 7-2.) Assumes (arg_1-arg_2) in floating accumulator. Does not apply to complex.
SGN	Sign transfer by mode. (See SIGN and DSIGN in Table 7-2.) Assumes arg_1 in floating accumulator, first word (with sign bit) of arg_2 in X register. Does not apply to complex.
STA*	Store accumulator by mode. Pack up floating accumulator into standard format and store as two- or four-word quantity.
LDX*	Load index. Same as machine instruction.

Op-Code	Description
LXP*	Load index immediate positive. Same as machine instruction.
LXM*	Load index immediate negative. Same as machine instruction.
ADX	Add to index. Add contents of addressed location to "X".
AXI*	Add to index immediate. Same as machine instruction.
SXI*	Subtract from index immediate. Same as machine instruction.
ADJ	Adjust index by mode. Multiply "X" by two for real, or by four for double precision and complex, to adjust for the number of words per element.
STX*	Store index. (Needed in case a subscript is to be used later.)
XIT	Exit from interpreter. Floating accumulator not guaranteed. A register reflects negative, positive, or zero value of last floating value.
XNL	Exit with no unlock. Meaningful only under RTX. Same as XIT but guarantees contents of floating accumulator.
INT	Convert to integer and exit. Once the floating accumulator has been converted to integer, you have to exit to make use of it (in the A register).

CONDITIONAL ASSEMBLY

Sections of in-line assembly code can be conditionally assembled, based on the value of special parameters that you set up. These parameters are called conditional assembly labels, and they have the form #Xn, where n is a decimal integer. The value of each parameter is established by a SET directive, whose operand is either one or zero. For example:

```
#X1      SET      0
#X73     SET      1
```

The SET op-code must have a #Xn label and an operand of zero or one. Any other usage is incorrect.

The conditional assembly label should then appear as the operand of an IFT (If True) or IFF (If False) directive. The value zero is considered false. The value one is considered true (unlike FORTRAN logical operations, where negative values are true). The section of assembly code follows the IFT or IFF and is terminated by an ENDC directive, which has no operand or label. The section is processed if the appropriate condition is met; otherwise it is ignored. For example:

```
IFT      #X3
JMP      #475
ENDC
```

The JMP would be assembled if #X3 is one, but not if it is zero.

The lines following an unsuccessful IFT or IFF are not processed at all, except to see if they begin "ENDCb". If not, they are completely ignored. This has the interesting effect that conditional assembly can be used to process or skip over FORTRAN statements as well as assembly lines. Suppose that, in the section following an IFT, there is a FORTRAN directive, some FORTRAN statements, an ASSEMBLER statement, and finally an ENDC. When the IFT is true, all of these will be processed as written. When it is false, everything will be skipped until the ENDC appears. The compiler will not know that some of the lines are in FORTRAN instead of assembly language, but it does not matter, as long as it eventually finds an ENDC line. The only thing to be careful about is that a FORTRAN statement such as:

```
ENDC = 0
```

would be interpreted as an ENDC line (if there is a blank after the C).

Note that the existence of the conditional assembly feature does not invalidate the use of X in column 1 -- it extends it. Either or both features may be used on assembly lines.

MISCELLANEOUS

Here are some additional pieces of information about the use of inline assembly language:

1. Compiler optimization and tracing features are suspended during sections of assembly language. Furthermore, the compiler will try not to dump out literal pools in the middle of assembly language, since it does not know where it would be safe to do so. If a section of assembly language is long enough, or comes at such a place that the compiler needs to dump out literals, it will produce a warning diagnostic, and then dump out the literals preceded by a jump around them. Most of the time this will work properly, but not, for example, in the middle of a CMS test or a group of floating point mnemonics or a table of DATA values. To get around this problem (or to get rid of the diagnostic), you should insert an LPOOL directive somewhere no later than the point at which the literals were dumped. The jump around is not generated by the LPOOL directive, since it sometimes is not needed. If you need one, you should write it. Note that assembly language itself does not generate anything that requires literals. They can only arise from preceding FORTRAN statements.

2. If you reference a FORTRAN name that has not previously been classified, it will be implicitly classified as a scalar.

3. You may not reference the name of an intrinsic function, since it has no location. If you declare it EXTERNAL, however, you can reference the corresponding external library routine.

4. Continuation lines are not allowed, since column six has no special significance.

5. Note that a decimal or hexadecimal value operand is not the same as a constant. That is:

```
LDA     5
```

does not load the value 5 but the contents of location 5. To provide for constants would require the use of literal pools or extra base page words, and this is not done. To reference a constant, you must define it with DATA. If you need a floating point constant, you must express it in hexadecimal, using two or more DATA lines.

6. The "FORTRAN" directive, which causes you to exit from in-line assembly language back into FORTRAN, does not have either an operand field nor a comment field.

CHAPTER 9

COMPILER OPTIONS

SUMMARY

Certain aspects of the compiler's operation can be controlled by a number of options. These options are specified on the control command that calls forth the compiler. There are default conditions for all of them, so that the compiler does something reasonable when no options are specified. The options are listed below. Three of them are described in more detail in the following sections.

ELIST Option

Error listing only. Normally the compiler produces a listing of all the source lines. When ELIST is specified, only source lines with errors are listed, along with their diagnostics.

LOBJ Option

List object code. An object listing can be rather long, and is often not needed, so the default is to not produce one. The object listing is printed separately from the source listing, but the source lines are interspersed at the appropriate places. Thus an object listing includes a source listing.

NBINARY Option

No binary output. Default is to produce a binary module.

XON Option

Compile conditional lines (with an X in column 1). Without this option, they are treated as comments. See "Conditional Compilation", in chapter 1.

ADP Option

Automatic double precision. All single precision quantities are converted to double precision. This is described below.

RSP Option

Reduce scratchpad. If your program overflows the scratchpad when linked (which requires enormous usage, unless there are other, assembly language routines using large amounts), there are two stages of reduction you can request in the compiler's use

of base page references. Normally the compiler uses base page to reference subprograms (including library), arrays, and variables in COMMON. The RSP option causes the compiler to call subprograms without using base page (i.e. by using a literal pool address pointer instead). This may reduce scratchpad usage by 20-50 words, meanwhile increasing the size of the program by somewhat more than that (depending on how many references there are to each subprogram and how spread out they are).

NSP Option

No scratchpad. If RSP doesn't do the job, you may have to resort to the NSP option, which eliminates all base page usage from the generated code (except those specifically requested by in-line assembly language), at the expense of significantly increasing the size of the program. This is mostly because subscripting without base page is quite clumsy.

RTX Option

Real time. This option must be specified when the object programs are to be executed under RTX. It causes slightly different calling and receiving sequences to be generated for proper interface in real time. Without this option, execution under OS is assumed. See below for more information.

T3 Option

Type 3/05 execution. This option causes the compiler to generate LSI-3/05 object code rather than LSI-2. It also assumes that the RTX option is wanted, even if you don't specify RTX. Since OS doesn't run on the LSI-3/05, RTX is obviously required. Note that when T3 is requested, the compiler will generate an external reference to the LSI-3/05 instruction emulator and software console routine (F3EMUL), because certain inline assembly instructions (those flagged with an asterisk in Section 8) don't exist on the LSI-3/05 and must be emulated.

TRACE Option

Run time trace. This causes the compiler to generate extra object code for tracing execution at run time. See below.

ANSI Option

ANSI compatible allocation. ANSI standard FORTRAN specifies that integer, real, and logical quantities occupy the same amount of storage. (Double precision and complex occupy twice that amount.) In most cases this does not matter, and it is more efficient on a 16-bit computer to allocate one word for integers and logicals, and two words for reals. If your program requires ANSI allocation (because of COMMON or EQUIVALENCE alignment), the ANSI option will allocate two words for integer and logical variables. Only the first word will be used in computation; the other will be ignored. Its only purpose is to separate the values so that the required amount of storage is taken.

There is one exception to the statement that the second word is never used. In any operation that simply steps through memory word by word, without regard to the type of variable, all words will be processed, including those that may be only separators between integer values in ANSI mode. This will almost always cause such operations to work incorrectly. Therefore you should not request ANSI allocation on any program that uses ENCODE or DECODE on an integer or logical type buffer, or that uses a FORMAT stored in an integer or logical array.

In general, if a program is compiled in ANSI mode, any programs with which it interfaces should also be compiled in ANSI mode. If there are not integer or logical variables in COMMON, or arrays being passed as arguments, this may not be necessary.

AUTOMATIC DOUBLE PRECISION

If you have programs doing computation in floating point, and you find that the single precision accuracy of about seven digits is not sufficient, you can use the ADP option to convert the program to double precision. Without the ADP option, this conversion would not be as simple as it may sound. It could involve:

1. Declaring every real variable, array, and non-library subprogram in a DOUBLE PRECISION statement.

2. Changing each appearance of a real constant to have a D exponent. (Actually, in Computer Automation FORTRAN IV, those constants that appeared in expressions would become double precision anyway, but not those that stand alone.

3. Changing each appearance of a real library function reference to the corresponding double precision version, if one exists.

4. Changing F, E, and G format specifications to D. (This would be necessary in ANSI standard FORTRAN, which does not permit those formats to be used with double precision data. Computer Automation FORTRAN IV does permit this.)

Therefore, when the ADP option is requested, the compiler proceeds essentially as if there were no such thing as single precision floating point. This means that it takes the following actions:

1. Any name that would ordinarily be typed real (either explicitly or implicitly) is typed double precision.

2. All floating point constants are automatically double precision.

3. Every reference to an intrinsic or basic external library function of real type is changed to reference the double precision equivalent, as shown below. Note that in some cases, this requires a double call, while in other cases it means removing the function call entirely.

Change	To	Change	To
ABS	DABS	COS	DCOS
AIMAG	DFLOAT(AIMAG)	DBLE	Removed
AINT	DINT	DIM	DDIM
ALOG	DLOG	EXP	DEXP
ALOG10	DLOG10	FLOAT	DFLOAT
AMAX0	DMAX0	REAL	DFLOAT(REAL)
AMAX1	DMAX1	SIGN	DSIGN
AMIN0	DMIN0	SIN	DSIN
AMIN1	DMIN1	SNGL	Removed
AMOD	DMOD	SQRT	DSQRT
ATAN	DATAN	TAN	DTAN
ATAN2	DATAN2	TANH	DTANH

As with the ANSI option, when one program is compiled in automatic double precision, the other programs with which it interfaces should also be compiled in this mode, so that arguments will be of the same type and COMMON will be correctly aligned.

Caution

If you know in advance that a program needs to be in double precision, it is better to write it that way in the first place, rather than using the ADP option, because the option is not entirely foolprcof. There are several areas where you must exercise caution in its use. These are:

1. Since there is no double precision complex type, ADP does not work on complex operations.

2. If you declare a library function EXTERNAL, the compiler will no longer recognize it and change it. What will happen is that the name (e.g. ALOG) will be classified as double precision (like any other ordinary name) and then called. However, the routine by that name in the library cannot know that it is supposed to be double precision. It will neither accept a double precision argument nor return a double precision result. You would have to provide a version that did.

3. If you use a real library function (e.g. ABS), but also use the name of the double precision version (e.g. DABS) to identify something unrelated (like a scalar or statement function), you may get diagnostics or strange results when the compiler tries to substitute that name. For example, if this program were compiled in ADP mode:

```
COMMON DCOS
DABS(X) = X/3
A = COS(B)
C = ABS(D)
```

COS would get an error diagnostic, while ABS would call the statement function DABS. Using the names of library routines for other purposes is not a very good idea in any case.

4. ADP does not affect inline assembly code. The operands will change to double precision, but the opcodes will not work properly.

REAL TIME

Any FORTRAN programs that are to be executed under RTX must be compiled with the RTX option. (The T3 option includes the RTX option within it). This changes the calling and receiving sequences somewhat, in order to comply with the RTX conventions that are used to handle real time usage of subprograms. If you want to run the same program under both RTX and OS, you should normally compile it twice. Note that if you have a single task (beginning with a TASK statement), it may be compiled without the RTX option and executed under OS, for debugging purposes. The execution address will then be the name of the task, rather than F:MAIN. (When the same task is compiled with the RTX option, no execution address is generated; it is assumed that F:MAIN, the RTX Mainline sequence, has been assembled separately, and will be linked with the task prior to execution; thus the execution address is F:MAIN.)

In SUBROUTINEs and FUNCTIONs, the local storage (variables and temps) is protected in real time. COMMON storage is not, nor is the local storage of main programs or TASKs. This means that it is difficult to connect a OFRTRAN TASK to more than one interrupt. If this is done, the TASK must have no local storage, which means it cannot do much. About all it can do is to call a SUBROUTINE which does the useful work.

Note that a TASK is essentially a main program with a name, but there must be exactly one true main program. There may be any number of TASKs.

RUN TIME TRACE

When the TRACE option is specified, the compiler generates extra run time calls in the compiled program that cause it to print out trace information (on unit 6) in three places:

1. Whenever a labeled statement is reached, the message:

   xxxxxx LINE dddddd

   is printed before the statement is executed, where:

   xxxxxx — is the name of the program (F:MAIN if main program). If the name is the same as that on the previous trace line, it is not printed. In other words, the name will be printed once when the program is entered, and not again until a new program is entered (or returned to).

   dddddd — is the source line number of the statement about to be executed.

2. When a SUBROUTINE or FUNCTION is entered, the message:

   xxxxxx ENTRY

   is printed immediately after entry. Again xxxxxx is the subprogram name, which will always be printed. Note that the tracing is done upon entry, not upon call. Therefore only subprograms that are compiled in TRACE mode will be traced.

3. When a RETURN statement is reached (whether or not labeled), the message:

   xxxxxx RETURN LINE ddddd

   is printed before executing the RETURN.

This information is sufficient to follow the flow of the program, since it will trace all jumps (the transfer point will be labeled) and all calls, except to library routines (which are assumed to operate correctly) and to subprograms not compiled in TRACE mode (which are also assumed to operate correctly). It is not necessary that all of the programs loaded be compiled in TRACE mode. As soon as certain parts are checked out, they can be compiled normally, so only the remaining parts are traced. Note that assembly language subprograms are not traced, nor are sections of in-line assembly language.

Appendix A

STATEMENT-ORDERING AND SIZE RESTRICTIONS

STATEMENT ORDERING

There are a few rules about the order in which statements may appear in a FORTRAN program. Some of these are inherent in the language (e.g. END must come last), while others improve readability and compiler efficiency (e.g. most declarations must come at the beginning). Table A-1 divides the statements into six groups, labeled 1 through 5 and X. Groups 1 through 5 must appear in that order, with no overlapping. For example, all the statements in group 2 must follow group 1 and precede group 3. Any of the groups except group 5 may be empty. Within a group, the statements may appear in any order. Note that there can be at most one statement in group 1.

The statements in group X need not appear together; in fact, they may appear anywhere after group 1 and before group 5. However, a DATA statement must follow any declaration statements that affect the variables to be initialized. In practice, EXTERNAL and DATA statements usually appear in group 1, and FORMAT statements in group 4.

Table A-1 also indicates whether each statement is executable or not. Occasionally it is important to know this. For example, a DO loop must end on an executable statement.

Table A-1. Statements and Ordering

Group	Statement	Executable	Non-executable
1	BLOCK DATA		X
	FUNCTION		X
	SUBROUTINE		X
	TASK		X
2	COMMON		X
	COMPLEX		X
	DIMENSION		X
	DOUBLE PRECISION		X
	EQUIVALENCE		X
	INTEGER		X
	LOGICAL		X
	REAL		X
3	Statement Function		X
4	Assignment	X	
	ASSEMBLER		X
	ASSIGN	X	
	BACKSPACE	X	
	CALL	X	
	CONTINUE	X	
	DECODE	X	
	DO	X	
	ENCODE	X	
	END FILE	X	
	GO TO	X	
	IF	X	
	In-line assembly		
	INPUT	X	
	OUTPUT	X	
	PAUSE	X	
	READ	X	
	RETURN	X	
	REWIND	X	
	STOP	X	
	WRITE	X	
5	END		X
X	DATA		X
	EXTERNAL		X
	FORMAT		X

OBJECT PROGRAM SIZE RESTRICTIONS

Due to object program layout, the total number of subprograms, unique arrays, dummy arrays and unique common scalars referenced must be less than 248.

Due to the structure of the compiler, there are certain other program size restrictions. The number of each of the following items must be less than 1023:

Scalar and Array Variables
Common Variables
Equivalenced Variable Names
Statement Numbers
Names in Explicit Type Statements
Unique REAL DOUBLE PRECISION and COMPLEX Constants
Unique INTEGER Constants
Unique Subprograms called
Arithmetic Statement Function Definitions

The total length of all Hollerith constants must be less than 1023; this includes character strings in OUTPUT statements but not in FORMAT statements. The length of a Hollerith constant is the number of words (that is, half the number of characters) in the string, plus 3.

APPENDIX B

COMPILER LISTINGS AND DIAGNOSTICS

COMPILER LISTINGS

The full listing of a compiled program consists of four parts:

1. Source listing
2. Variable storage allocation
3. Object listing
4. Summary

When no special options are requested, the object listing is not produced, but the other three are. The LO (List Object) option causes the object listing to be produced. If the EL (Error List only) option is specified, the source listing is suppressed, except for the first line and any lines that have errors.

Figure B-1 shows a complete program listing. For further explanation, please refer to the FORTRAN IV Operations Manual (96510-01).

COMPILER DIAGNOSTICS

Figure B-2 is a sample program for illustrating the format of compiler diagnostics. Most errors are detected during the Scan phase and are printed on the source listing immediately following the statement in error. A dollar sign is printed underneath the position at which the error was detected, followed by a brief message. If the message is followed by W's, it is only a warning. If it is followed by E's, it is an error and the statement has not been generated. Instead, a call to a run time error routine is generated. Thus if any statement with an "E" type error is executed, a run time diagnostic will occur.

If there is more than one dollar sign printed, the count at the beginning of each message indicates which dollar sign it refers to, counting from left to right. Note on line 0013 that both messages refer to the same dollar sign.

A few error conditions are not detected until the Allocate phase (or even the Generation phase), so the diagnostics for these would appear in the allocation map or in the object listing. For example, the UNDEFINED LABELS and ALLOCATION ERRORS messages in Figure B-2.

Most of the error messages are self-explanatory, but the FORTRAN IV Operations Manual contains a complete list of them, along with descriptions of possible causes. The Operations Manual also describes the compiler abort messages (usually caused by hardware failure) and the error messages produced at run time (when the program is executed).

Note that the last line of the summary (i.e. the last line printed in any program listing) indicates how many errors have been detected.

```
PAGE 0001   09/20/74   10:51:25   FORT:4 (A0)
 BO FILE:   FOUT     OPTIONS:     LO

0001 C            DEMONSTRATE OBJECT LISING
0002        INTEGER NN(25), LL(10)
0003        DOUBLE PRECISION DX, DY
0004        COMMON MM(100), M /BLK/ Y
0005        EQUIVALENCE (L,LL)
0006        ISI(KD) = KD*8
0007     10 K = (L+300)*M - 74
0008        MM(I) = K
0009        X = ABS(Y+4)
0010        DX = DABS(DY/4.3)
0011        IF (DX .LT. 0) GO TO 70
0012        CALL SUB(L+300,7HABCDE  ,Y+4)
0013     20 WRITE(6,30) Y
0014     30 FORMAT( 5X , I5 , ' VALUES.' )
0015        IF (K .EQ. M) GO TO 10
0016           ASSEMBLER
0017           LAP     :2A
0018           ADD     K                      (LOCAL VARIABLE IN RANGE)
0019           STA     *BP(MY:NAM)            (SPECIAL SYSTEM NAME)
0020           JMP     #50                    (FORWARD REFFRENCE IN RANGE)
0021 #40       RES     32,'  '                (TO FORCE LITERAL POOL)
0022           FORTRAN
0023     50 DO 60 I = 1,10
0024     60 NN(I) = -1
0025        ASSIGN 40 TO K
0026        MM(3) = 0

01) UNDIMENSIONED E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E

0027     70 STOP
0028        END
```

Figure B-1. Sample Compiler Listing

```
PAGE 0002  09/20/74  10:51:25  FORT:4 (A0)
 OC FILE:  FOUT     OPTIONS:     LO

COMMON BLOCK/F:BCMN/ ALLOCATION  :0065 WORDS

LOCN  NAME    TYPE       WORDS    LOCN  NAME    TYPE       WORDS

:0000 MM      INTEGER      100    :0064 M       INTEGER        1

COMMON BLOCK/BLK   / ALLOCATION  :0002 WORDS

LOCN  NAME    TYPE       WORDS    LOCN  NAME    TYPE       WORDS

:0000 Y       REAL           2

ARRAY ALLOCATION

LOCN  NAME    TYPE       WORDS    LOCN  NAME    TYPE       WORDS

:0009 NN      INTEGER       25

EQUIVALENCE ALLOCATION

LOCN  NAME    TYPE       WORDS    LOCN  NAME    TYPE       WORDS

:0022 L       INTEGER        1    :0022 LL      INTEGER       10

SCALAR ALLOCATION

LOCN  NAME    TYPE       WORDS    LOCN  NAME    TYPE       WORDS

:002C K       INTEGER        1    :002D I       INTEGER        1
:002E X       REAL           2    :0030 DX      DOUBLE         4
:0034 DY      DOUBLE         4
```

Figure B-1. Sample Compiler Listing (Cont'd)

```
PAGE 0003  09/20/74  10:51:25  FORT:4 (A0)
 00 FILE:  FOUT     OPTIONS:     LO

0001 C              DEMONSTRATE OBJECT LISTING
0002                INTEGER NN(25), LL(16)
0003                DOUBLE PRECISION DX, DY
0004                COMMON MM(100), M /BLK/ Y
0005                EQUIVALENCE (L,LL)
0006                ISF(KD) = KD*8
                        :0038 :F200 F          JMP    #M7
                        :0039 :0800    #M8     ENT
                        :003A :F900 B          JST    *BP(F:RDMY)
                        :003B :0001            DATA   1
                        :003C :0000    KD      DATA   0
                        :003D :6701            LDA    *KD
                        :003E :1052            ALA    3
                        :003F :F706            JMP    *#M8
0007                K = (L+300)*M - 74
                        :0040          #M7     EQU    :0040
                        :0040 :B200 F #10      LDA    #IC1                    :012C
                        :0041 :8E1F            ADD    L
                        :0042 :9A00 F          STA    #T0
                        :0043 :F900 B          JST    *BP(F:RMPY)
                        :0044 :0064 C          DATA   M
                        :0045 :0D4A            SAI    74
                        :0046 :9E1A            STA    K
0008                MM(I) = K
                        :0047 :E61A            LDX    I
                        :0048 :9D00 B          STA    *#BP(MM     -1)
0009                X = ABS(Y+4)
                        :0049 :F900 B          JST    *BP(F:RREL)
                        :004A :AA00 F          LDR    #RC1                    :4180:0000
                        :004B :6900 B          ADD    *BP(Y       )
                        :004C :9A00 F          STA    #T1
                        :004D :0005            ABS
                        :004E :9E20            STA    X
0010                DX = DABS(DY/4.3)
                        :004F :B61D            LDD    DY
                        :0050 :A200 F          DVM    #RC2                    :4189:9999:99
                        :0051 :0005            ABS
                        :0052 :9E22            STA    DX
0011                IF (DX .LT. 0) GO TO 70
                        :0053 :0000            XIT
                        :0054 :2080 F          JAM    #M9
0012                CALL SUB(L+300,7HABCDE  ,Y+4)
                        :0055 :F900 B          JST    *BP(SUB     )
                        :0056 :0003            DATA   3
                        :0057 :0000 F          DATA   #I0
                        :0058 :0000 F          DATA   #HC0
                        :0059 :0000 F          DATA   #T1
0013                WRITE(6,30) Y
                        :005A :F900 B #20      JST    *BP(F:RWF )
                        :005B :0000 F          DATA   #IC5                    :0006
                        :005C :0000            DATA   #30
```

Figure B-1. Sample Compiler Listing (Cont'd)

```
                  :005D :F900 B          JST    *BP(F:RROL)
                  :005E :0000 C          DATA   Y
                  :005F :F900 B          JST    *BP(F:RSIO)
0014       30 FORMAT( 5X , I5 , ' VALUES.' )
                  :0000 :A8B5     #30    TEXT   '(5X,I5,' VALUES.')'
0015          IF (K .EQ. M) GO TO 10
                  :0060 :B634            LDA    K
                  :0061 :9100 B          SUB    *BP(M       )
                  :0062 :2100 F          JAZ    #M10
                  :0040           #M10   EQU    :0040
0016              ASSEMBLER
0017              LAP     :2A
                  :0063 :C62A            LAP    :002A
0018              ADD      K                (LOCAL VARIABLE IN RANGE)
                  :0064 :8E38            ADD    K
0019              STA    *BP(MY:NAM)        (SPECIAL SYSTEM NAME)
                  :0065 :9900 B          STA    *BP(MY:NAM)
0020              JMP     #50               (FORWARD REFERENCE IN RANGE)
                  :0066 :F200 F          JMP    #50
0021 #40          RES     32,'  '           (TO FORCE LITERAL POOL)
                  :0067 :A0A0     #40    RES    32,'  '
0022              FORTRAN
0023       50 DO 60 I = 1,10
                  :0087 :C401     #50    LXP    1
                  :0086 :EE5B     #M11   STX    I
0024       60 MM(I) = -1
                  :0089 :C701     #60    LAM    1
                  :008A :9D00 B          STA    **BP(MM      -1)
0025          ASSIGN 40 TO K
                  :008B :C201            AXI    1
                  :008C :0030            TXA
                  :008D :0D0A            SAI    10
                  :008E :21C6            JAL    #M11
                  :008F :B200 F          LDA    #40
                  :0090 :F200 F   #L     JMP    #M12              LITERAL POOL
                  :0091 :F200 F          JMP    #M9
                  :0092 :9E66     #M12   STA    K
0026          MN(3) = 0
               $
01) UNDIMENSIONED E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*F*E*F*E*E*E*E*
                  :0093 :F900 B          JST    *BP(F:RERR)
                  :0094 :001A            DATA   26
                  :0095 :0000 F          DATA   #RC4                      :6306:4167
0027       70 STOP
                  :0096           #70    EQU    :0096
                  :0096 :F900 B   #M9    JST    *BP(F:RSTO)
                  :0097 :0000            DATA   0
0028          END
                  :0098 :012C     #IC1   DATA   300
                  :0099 :4180     #RC1   DATA   16768
                  :009A :0000            DATA   0
```

Figure B-1. Sample Compiler Listing (Cont'd)

```
PAGE 0005  09/20/74  10:51:25  FORT:4 (A0)
 DO FILL:  FOUT    OPTIONS:    LO

                :009B :4189    #RC2    DATA  16777
                :009C :9999            DATA  -26215
                :009D :9999            DATA  -26215
                :009E :999A            DATA  -26214
                :009F :0000    #T0     DATA  0
                :00A0 :0000    #T1     DATA  0
                :00A2 :0067            DATA  #40
                :00A3 :0007            DATA  7
                :00A4 :C1C2    #HC0    DATA  'AB'
                :00A5 :C3C4            DATA  'CD'
                :00A6 :C5A0            DATA  'E '
                :00A7 :A0A0            DATA  '  '
                :00A8 :0006    #IC5    DATA  6
                :00A9 :63C6    #RC4    DATA  25542
                :00AA :4167            DATA  16743

SUBPROGRAMS CALLED

NAME    TYPE      ARGS   NAME     TYPE      ARGS   NAME     TYPE      ARGS

ABS     REAL      1      DABS     DOUBLE    1      SUB      REAL      3
F:PWF   RUNTIME          F:RROL   RUNTIME          F:RSTO   RUNTIME
MY:NAM  RUNTIME          F:RERR   RUNTIME          F:RSTO   RUNTIME
F:PWDO  RUNTIME          F:RREL   RUNTIME          F:RDBL   RUNTIME
F:PF7   RUNTIME          F:RFF    RUNTIME          F:RDMY   RUNTIME
F:EMPY  RUNTIME

STATEMENT LABELS

LOCN  LABEL  USE          LOCN  LABEL  USE          LOCN  LABEL  USE

:0000 #10                 :0096 #70                 :005A #20    UNUSED
:0000 #30      FORMAT     :0087 #50                 :0067 #40
:0089 #50      DO END     :0040 #M7                 :0039 #M8
:0096 #M9                 :0040 #M10                :0088 #M11
:0092 #M12

ENTRY=:0038
PROGRAM SIZE=:00AB WORDS
BASE PAGE USED=:000D WORDS
COMPILATION COMPLETE  1 ERRORS
```

Figure B-1. Sample Compiler Listing (Cont'd)

```
PAGE 0001  09/20/74  11:41:16  FORT:4 (A0)
 DO FILE:  FOUT    OPTIONS:

0001 C          DEMONSTRATE COMPILER DIAGNOSTICS
0002            DIMENSION MM(10,10)
0003            COMMON X, Y, X
                           $
01) DECLARATION CONFLICT E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*


0004            EQUIVALENCE (X,Y)
0005            LOGICAL LGL, N
0006            INTEGER A,       , C
                                $
01) EXTRA COMMA W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W


0007            SF(P,Q) = P+Q/2
0008            X = 1E50 + LGL
                      $    $
01) CONSTANT SIZE E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E

02) TYPE CONFLICT E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E


0009            IF (A) 2,3,2
0010          2 X = SQRT(A) + SF(Y)
                       $       $
01) ARGUMENT CONVERTED W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W*W

02) ARGUMENT COUNT E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E


0011            B(10) = 0
                $
01) UNDIMENSIONED E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E


0012            X = (RX+ABS(SX))/(VAL+3))
                                     $
01) SYNTAX E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E


0013            J = MM(N)
                      $
01) NOT INTEGER E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E

01) NUMBER OF SUBSCRIPTS E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E


0014            END
```

Figure B-2. Sample Diagnostic Listing

```
PAGE 0002  09/20/74  11:41:16  FORT:4 (A0)
 BO FILE:  FOUT     OPTIONS:

UNDEFINED LABELS E*E*L*E*E*L*E*L*E*E*L*E*E*E*E*E*E*E*E*E*E*E*L*E*E*L*E*L*E*

    3 FIRST REF AT LINE      9


COMMON BLOCK/F:BCMN/ ALLOCATION  :0004 WORDS

LOCN  NAME     TYPE       WORDS    LOCN  NAME     TYPE       WORDS

:0000 X        REAL           2    :0002 Y        REAL           2

ARRAY ALLOCATION

LOCN  NAME     TYPE       WORDS    LOCN  NAME     TYPE       WORDS

:0000 M1       INTEGER      100

SCALAR ALLOCATION

LOCN  NAME     TYPE       WORDS    LOCN  NAME     TYPE       WORDS

:0064 A        INTEGER        1    :0065 J        INTEGER        1
:0066 N        LOGICAL        1

ALLOCATION ERRORS E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*E*

Y
```

Figure B-2. Sample Diagnostic Listing (Cont'd)

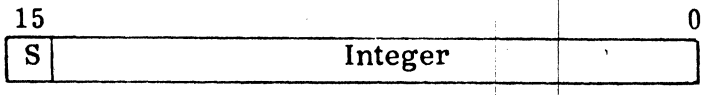
```
PAGE 0003  09/20/74  11:41:16  FORT:4 (A0)
 B0: FILE:  FOUT    OPTIONS:

SUBPROGRAMS CALLED

NAME     TYPE     ARGS    NAME     TYPE     ARGS    NAME     TYPE     ARGS

F:RERR   RUNTIME          SQRT     REAL     1       F:RSTO   RUNTIME
F:RREL   RUNTIME          F:RDMY   RUNTIME

STATEMENT LABELS

LOCN  LABEL  USE          LOCN  LABEL  USE          LOCN  LABEL  USE

:0078 #2                  :FFFF #3                  :0076 #M2
:0068 #M3

ENTRY=:0067
PROGRAM SIZE=:0094 WORDS
BASE PAGE USED=:0004 WORDS
COMPILATION COMPLETE  12 ERRORS
```

Figure B-2. Sample Diagnostic Listing (Cont'd)

APPENDIX C

INTERNAL DATA FORMATS AND ASCII CODES

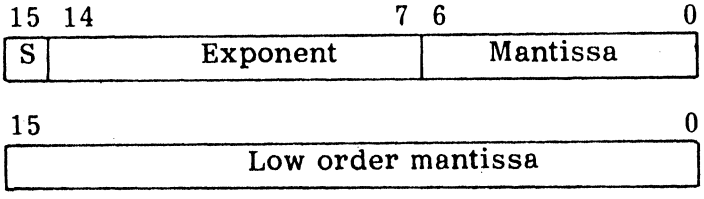
1. Integer. 1 word, unless the ANSI option is requested, in which case 2 words are allocated (for variables) but only the first is used. Bit 15 is the sign bit, and the remaining fifteen bits are the right justified integer value. The negative of a number is its two's complement.



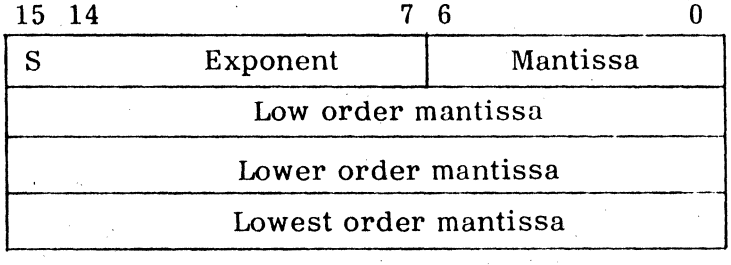
2. Real. 2 words. The first word contains the sign bit, an eight bit exponent (or characteristic) of base two, which is biased by 128, and the high order seven bits of the normalized mantissa. The second word contains the low order sixteen bits of the mantissa. The high order 1-bit in the normalized mantissa is not present, but only implied. This makes room for one more bit of precision. It also means that there is no combination of bits that is not a legitimate normalized floating point value. And it means that even though the first word of a floating number is zero, the value may not be, since there may be bits in the lower order mantissa.

   The exponent range is 2^{-128} to 2^{+127}. The resulting range of values is 1.469368E-39 to 1.701411E38. The 23 bits of mantissa (plus implied high order bit) give an accuracy of somewhat more than seven decimal digits.

   This is a sign-magnitude system. The negative of a number is obtained by merely setting the sign bit; the mantissa and exponent do not change.



3. Double precision. 4 words. Exactly the same as real, except that there are two additional words (32 bits) of mantissa following the first two words. The exponent range is the same. The 55 bits of mantissa give an accuracy of about 17 decimal digits.



4. Complex. 4 words. Consists of two single precision (real) floating point numbers. The first is the real part, the second the imaginary part.

5. Logical. 1 word, unless the ANSI option is requested, in which case 2 words are allocated (for variables) but only the first is used. Only the sign bit (bit 15) is significant in logical operations. Any word that is negative (bit 15 = 1) is true, while any word that is positive or zero (bit 15 = 0) is false. Note that the compiler generates .TRUE. and .FALSE. as all ones and all zeros respectively, but this is not necessary, since only the sign bit is tested in logical operations.

6. Alphanumeric. Hollerith constants are 1 word (two characters). Alphanumeric string constants can be any length, always with two characters per word, and are preceded by a word containing the (right justified) integer count of the number of characters in the string.

   Each alphanumeric character is an 8-bit ASCII code, with the high order bit always set to one. There are thus 128 legitimate ASCII codes, but only 64 of them are graphic (printable) characters. These are shown in Table C-1, along with their hexadecimal equivalents. We do not recommend that you take advantage of knowing these hexadecimal values (i.e.by doing numeric calculations with alphanumeric characters), because the values vary widely on different computer systems. The table also shows the punched card code for each character.

   Note that the three characters [, \, and] do not print on the teletype.

Table C-1. ASCII Character Codes

Character	Hex Value	Card code	Character	Hex Value	Card code
Blank	:A0	Blank	@	:C0	4-8
!	:A1	11-2-8	A	:C1	12-1
"	:A2	7-8	B	:C2	12-2
#	:A3	3-8	C	:C3	12-3
$	:A4	11-3-8	D	:C4	12-4
%	:A5	0-4-8	E	:C5	12-5
&	:A6	12	F	:C6	12-6
'	:A7	5-8	G	:C7	12-7
(	:A8	12-5-8	H	:C8	12-8
)	:A9	11-5-8	I	:C9	12-9
*	:AA	11-4-8	J	:CA	11-1
+	:AB	12-6-8	K	:CB	11-2
,	:AC	0-3-8	L	:CC	11-3
-	:AD	11	M	:CD	11-4
.	:AE	12-3-8	N	:CE	11-5
/	:AF	0-1	O	:CF	11-6
0	:BO	0	P	:D0	11-7
1	:B1	1	Q	:D1	11-8
2	:B2	2	R	:D2	11-9
3	:B3	3	S	:D3	0-2
4	:B4	4	T	:D4	0-3
5	:B5	5	U	:D5	0-4
6	:B6	6	V	:D6	0-5
7	:B7	7	W	:D7	0-6
8	:B8	8	X	:D8	0-7
9	:B9	9	Y	:D9	0-8
:	:BA	2-8	Z	:DA	0-9
;	:BB	11-6-8	[†	:DB	0-2-8
<	:BC	12-4-8	\†	:DC	11-7-8
=	:BD	6-8	]†	:DD	0-5-8
>	:BE	0-6-8	↑	:DE	12-2-8
?	:BF	0-7-8	←	:DF	12-7-8

† Not available on teletype.

APPENDIX D

ANSI COMPATIBILITY

The Introduction stated that ANSI standard FORTRAN is a subset of Computer Automation FORTRAN IV, i.e., that any legal ANSI program will work the same way in Computer Automation FORTRAN IV. There are two minor exceptions. The first was changed to produce smaller object programs, but can be changed back by the ANSI allocation option. The other is quite obscure and rarely occurs, and we have implemented it differently because we felt it made more sense. The two differences are:

1. Integer and logical variables occupy only one word, while real variables occupy two. ANSI says they should be the same. It is hard to do this efficiently on a 16-bit machine, so normally we do not allocate them that way. However, you can request this by using the ANSI option (see chapter 9). This difference is important only in certain cases of mixed mode alignment of COMMON or EQUIVALENCE.

2. According to ANSI, a positive scale factor of value n used with an Ew.d format produces n significant digits to the left of the decimal point and (d-n+1) to the right. That is, as digits are added on at the left, they are taken off at the right, beginning at n=2. The effect for n>d+1 is undefined. In actuality, almost all FORTRAN systems keep constant the number of digits to the right of the decimal point, as shown in chapter 5, under "P Specification".

ADDITIONAL FEATURES

On the other hand, there are a number of significant extensions to ANSI FORTRAN included in Computer Automation FORTRAN IV, as well as some minor extensions. These are listed below.

General Features

1. In-line assembly language.

2. Conditional compilation (X in column 1).

3. Automatic Double Precision (ADP) option.

4. Any number of continuation lines.

5. Extra library functions:

DDIM	DMAX0	DTANH	INOT
DFLOAT	DMIN0	IAND	IOR
DINT	DTAN	IEOR	TAN

Data and Expressions

1. Lower and upper subscript bounds on arrays. If a dummy array, both limits may be adjustable (specified by another dummy).

2. Any number of dimensions on an array.

3. Any integer expression may be used as a subscript. This includes subscripted subscripts.

4. Names of any length (first six characters significant).

5. Hexadecimal constants.

6. Hollerith constants in expressions. If standing alone on the right side of an equal sign, they may be as long as permitted by the type of the variable on the left of the equal sign. Otherwise they are integer (one or two characters).

7. Long alphanumeric strings enclosed in quotes (in DATA statements or argument lists).

8. A real constant in a double precision expression automatically becomes double precision.

9. More cases of mixed mode expressions are allowed, including:

   a. Integer may be mixed with real, double precision, and complex, using the operators +, -, *, and /.

   b. Double precision may be mixed with complex using the same set of operators. (The result is complex.)

   c. An integer may be raised to a real or double precision power.

   d. Integer may be compared with real or double precision, using any of the relational operators.

   e. Complex may be compared with integer, real, double precision, or another complex, using only the operators .EQ. or .NE..

f. An assignment statement may have a complex variable on the left and an integer, real, or double precision expression on the right. (The latter involves a loss of precision.)

10. Boolean operations, using the intrinsic functions IAND, IOR, IEOR, and INOT.

11. > and < may be used as relational operators, in place of .GT. and .LT..

12. ↑ may be used for exponentiation, in place of **.

13. The sequence .NOT. .NOT. is permitted.

Statements

1. Free form I/O statements, OUTPUT and INPUT.

2. Internal data conversion statements, ENCODE and DECODE.

3. TASK statement, for real time programs that are connected to interrupts.

4. END= and ERR= options on READ and WRITE.

5. New FORMAT specifications: T(Tab), Z (Hexadecimal), ' (Alphanumeric string), $ (Preceding dollar sign), and * (Asterisk fill).

6. Other features in FORMAT statements:

   a. All of the numeric formats (I,F,E,D,G) accept any of the numeric types of data (integer, real, double precision, or either part of complex).

   b. Comma termination of numeric input fields.

   c. Deeper nesting of parenthesized groups, to eight levels.

   d. The first T or F in a logical input field determines the value, rather than the first character (so .TRUE. is a permissible input field).

   e. The A format also works with double precision variables.

7. Features in the DATA statement:

   a. An unsubscripted array represents all of its elements.

   b. A long alphanumeric string may initialize any number of variables (or array elements).

c. Hexadecimal constants may be as long as required by the variable type.

d. Variables in labeled COMMON may be initialized in any program, not just in a BLOCK DATA subprogram.

8. DO control parameters may be negative or zero (except for the increment).

9. A statement function definition may reference array elements and Hollerith constants.

10. In EQUIVALENCE, a scalar may be followed by a position count enclosed in parentheses (in the same manner as an array name).

11. PAUSE and STOP may be followed by a decimal constant, rather than octal.

12. The END statement may be labeled, and simulates a STOP or RETURN if necessary.

Syntax Relaxations

1. The parenthesized list of statement numbers in an assigned GO TO is optional.

2. There may be a comma in a DO statement between the terminating statement number and the control variable.

3. There need not be a comma in the following places:

a. In a computed GO TO, after the right parenthesis.

b. In an assigned GO TO, before the left parenthesis (if any).