FORTRAN-80 PROGRAMMING MANUAL

Manual Order Number: 9800481A

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

PRINTED IN U.S.A./A151/1279/5K CP

PREFACE

This manual describes the Intel-developed FORTRAN language (FORTRAN-80) for programming the 8080 and 8085 microcomputers. FORTRAN-80 is based on the ANSI FORTRAN 77 subset. In some instances, it incorporates features from the FORTRAN 77 full language; FORTRAN-80 also has features that exceed both versions of FORTRAN 77.

Appendix D lists FORTRAN-80 extensions to the FORTRAN 77 subset that can be found in the FORTRAN 77 full set; it also lists those FORTRAN-80 superset features that go beyond both versions of FORTRAN 77. In addition, these superset features are clearly marked in the text of this manual (shadowed in grey) and should not be used if you want to preserve total compatibility with FORTRAN 77 and portability among processors.

The FORTRAN-80 language is described in its entirety in this manual. Limitations or extensions related to a particular compiler are described in the operator's manual for that compiler. They are summarized in Appendix F of this manual. The operator's manual also includes considerations when running in different operating system environments (such as ISIS-II and RMX-80).

The experienced FORTRAN programmer can possibly begin programming after reviewing the summaries in the appendixes of this manual. The new programmer should read it through from the beginning. While the manual is primarily written as a programming reference, it does contain some instructional material.

Chapter 1 has a short program designed to provide an intuitive feel for the language. Chapter 2 introduces FORTRAN concepts. Chapters 3-6 describe FORTRAN-80 statements in detail. These statements are grouped functionally and include brief illustrative examples. Chapter 7 suggests some guidelines for improving FORTRAN programming techniques and recommends sources for further study of programming as a science. We also suggest that the beginning programmer read one two FORTRAN tutorials. Some recommended introductory texts are included in the bibliography at the end of Chapter 7.

Finally, all users of this manual should refer to the following documents as necessary:

ISIS-II FORTRAN-80 Compiler Operator's Manual	9800480
ISIS-II System User's Guide	9800306

X3.9-1977 FORTRAN

The latter document can be ordered from:

The American National Standards Institute, Inc.
1430 Broadway
New York, New York
10018

CONTENTS

ILLUSTRATIONS

Argument(s) - A collection of values and variables on which a computation is performed. Functions and subroutines are usually defined with *dummy* arguments that are replaced with *actual* values when the functions or subroutines are referenced.

Array - An ordered set of data that can be referenced collectively (by array name) or selectively (by array element name).

Array Element - An individual item within an array.

Association - May refer to association of arguments, of memory locations, or of symbols. Association of arguments is the replacement of dummy arguments with actuals when a procedure is referenced. Association of memory locations is the sharing of memory by two or more items. The symbol names of the items sharing memory are also said to be associated.

Common Memory - Memory shared by items in the same or different program units.

Compiler - The software tool for translating FORTRAN source code into machine-executable form.

Equivalenced Memory - Memory shared by items in the same program units.

Expression - A combination of operands, operators, and parentheses. May be arithmetic, character, relational, logical, or Boolean.

File - A collection of data records. May be external (any ISIS-recognized file) or internal (a character variable or character array element). Records can be accessed sequentially or directly.

Function - A routine that returns a value to the calling statement when it is referenced. Functions are called 'intrinsic' (FORTRAN predefined), 'statement' (user-defined, single-statement function), or 'external' (user-defined FUNCTION subprogram).

Length of Data - The number of bytes occupied by a data item. This can be one, two, or four bytes for integer and logical items and four bytes for real items. Character data occupies one byte per character.

Main Program - The main program is the first part of an entire FORTRAN program to be invoked. It may not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It may have a PROGRAM statement as its first statement.

Number Base - The representation used for numeric data. May be binary, octal, decimal, or hexadecimal.

Procedure - Another term for a function or subroutine. FUNCTION and SUBROUTINE subprograms are called 'external' procedures.

Program - The entire executable program includes the main program, all subprograms (FUNCTION, SUBROUTINE, BLOCK DATA) plus any compiler control lines, library procedures, and included files.

Program Unit - Another name for a main program or a subprogram. Every program unit must be terminated by an END statement.

Record - A sequence of values or characters.

Statement - A sequence of syntactic items: statement label, keyword, arguments, expressions, etc. A statement has an 'initial' line and up to nine 'continuation' lines.

Statement Label - A 1-5 digit integer in columns 1-5 of a statement's initial line. Can be given a symbolic name by the ASSIGN statement.

Subprogram - A block of code having a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement and the END statement as its last statement.

Subroutine - A group of statements for performing a frequently-used operation. The SUBROUTINE statement must be the first statement; the END statement must be the last.

Unit - A logical way of referring to a file. Once connected, it is the same as a file.

CHAPTER 1
INTRODUCTION TO FORTRAN

This chapter opens with a short example intended to give the newcomer to FORTRAN a feel for the language. The example is discussed in some detail. The chapter also includes a summary of FORTRAN-80 statements and their proper coding sequence.

1.1 An Introductory Example

A FORTRAN program generally performs three basic operations: receiving input, processing the data received, and returning output. The following short program, drawn from the statistical world of the sports fan(atic), shows typical FORTRAN statements for doing these operations.

The example calculates a baseball player's batting average using the equation:

$$\text{AVERAGE} = \frac{\text{HITS}}{\text{TIMES AT BAT}}$$

The baseball statistician, sitting at his console terminal, enters the name of a player, how often he has batted, and his total hits. The program returns a listing showing the player's name and batting average. To keep this example simple, the calculation is done only once.

1.1.1 Comment Lines

The first seven lines of the example are *comment lines*. Comment lines are used to document a program.

A comment line must have the letter 'C' or an asterisk (*) in column 1 followed by any characters accepted by FORTRAN in the remainder of the line. A completely blank line is considered a comment line also. For example, the 'C' need not be present in line 7 of the example. Comment lines have no effect on program execution.

```
C CALCULATE BATTING AVERAGES
C VARIABLES USED—
C    PNAME = PLAYER'S NAME
C    AB = TIMES AT BAT
C    HITS = TOTAL BASE HITS
C    AVG = BATTING AVERAGE
C
      CHARACTER*12  PNAME
      READ 10, PNAME, AB, HITS
10    FORMAT (A, 2(2X, F3.0))
      AVG = HITS/AB
      PRINT 20, PNAME, AVG
20    FORMAT (A, 5X, F4.3)
      END
```

Fig. 1-1 Batting Average Program

1.1.2 Type Statement

Every variable used in a FORTRAN program has a type — integer, real, logical, character, or Hollerith. The CHARACTER type statement says that the variable PNAME represents character data and may have up to 12 characters.

No type statement is needed for the other variables listed (AB, HITS, AVG). The FORTRAN variable-naming convention tells us implicitly that these variables are to be used to name real data. This convention is described later in section 2.2.2.1.

1.1.3 Input Statements

Following the CHARACTER statement in the example are two input statements. The first tells the program to read input data; the second describes the format of the input data.

The number 10 in the READ statement tells the program that the input format is found in statement 10. The rest of the statement lists the variables whose values will be specified by the person at the console. By default, the input is read from the console terminal.

The FORMAT statement (labeled statement 10) tells the program what kind of data to expect. 'A' indicates the first field of data is a string of alphanumeric characters having the length of PNAME, the player's name. '2(2X, F3.0)' is equivalent to

```
2X, F3.0, 2X, F3.0
```

and refers to the AB and HIT fields. '2X' indicates two blanks will be entered followed by a 3-digit floating-point (F) number whose decimal portion contains '0' digits. Clearly, no batter comes to the plate 79.3 times or has 22.8 hits.

One might ask at this point why we didn't specify these fields to be 'integer' data; that is,

```
2(2X,I3)
```

The reason is that FORTRAN truncates the remainder when one integer is divided by another. Since AB ≥ HITS, all averages would be '0' except for the rare player batting 1.000.

1.1.4 Value Assignment Statement

The actual batting average calculation is done by the next statement:

```
AVG = HITS/AB
```

This is one form of *assignment statement*, in which the variable 'AVG' is assigned the value of the expression 'HITS/AB.'

1.1.5 Output Statements

Following the calculation are two output lines. The first tells the program to write output data; the next describes the format of the output.

In the PRINT statement, the number 20 indicates that the output format is found in statement 20. 'PNAME' and 'AVG' are the items whose values are to be printed. By default, the output is written to the console.

Statement 20, the FORMAT statement, indicates the 'PNAME' field will be a string of characters of variable length, as in the FORMAT statement labeled 10. The name will be followed by five blanks (5X) and then the batting average will be printed. The 'AVG' field consists of four floating-point digits -- one integer digit and three decimal digits.

1.1.6 Program Termination

The final statement terminates the program. The END statement is an indicator to the FORTRAN compiler that it has reached the end of the program. Every program unit must be terminated by an END statement.

1.2 Summary Of FORTRAN-80 Statements

The statements available in FORTRAN-80 are listed below according to their main classifications. These include all statements available in the FORTRAN 77 subset and some from the FORTRAN 77 full language. To simplify comparison, the statements are listed in the same sequence as in Section 7 of the ANSI standard. All chapter references are to *this* manual, however, not to the ANSI standard.

Statements are classified as *executable* or *nonexecutable*. Executable statements do calculations, read or write I/O data, and control program execution. Nonexecutable statements define the characteristics or value of data and define program units.

1.2.1 Executable Statements

1. Arithmetic, logical, and character assignment statements; ASSIGN statement (Chapter 3);
2. Unconditional, assigned, and computed GO TO statements (Chapter 4);
3. Arithmetic and logical IF statements (Chapter 4);
4. Block IF, ELSE IF, ELSE, and END IF statements (Chapter 4);
5. CONTINUE statement (Chapter 4);
6. STOP and PAUSE statements (Chapter 4);
7. DO statement (Chapter 4);
8. READ, WRITE, and PRINT statements (Chapter 6);
9. REWIND, BACKSPACE, ENDFILE, OPEN, and CLOSE statements (Chapter 6);
10. CALL and RETURN statements (Chapter 5);
11. END statement (Chapter 4).

1.2.2 Nonexecutable Statements

1. PROGRAM (Chapter 2), BLOCK DATA (Chapter 3), FUNCTION, and SUBROUTINE (Chapter 5) statements;
2. DIMENSION, COMMON, EQUIVALENCE, IMPLICIT (Chapter 3), EXTERNAL, INTRINSIC, and SAVE (Chapter 5) statements;
3. INTEGER, REAL, LOGICAL, CHARACTER type statements (Chapter 3);
4. DATA statement (Chapter 3);
5. FORMAT statement (Chapter 6);
6. Statement function statement (Chapter 5).

1.2.3 Order of Statements

The following order must be observed in coding FORTRAN statements lines:

1. Comment lines can appear before or between statements. They cannot appear after the END statement.
2. The PROGRAM statement can appear only as the first statement of a main program. FUNCTION, SUBROUTINE, and BLOCK DATA can appear only as the first statement of a subprogram (Section 2.1.1).
3. FORMAT statements can appear anywhere before the END statement.
4. IMPLICIT statements must precede all other specification statements.
5. All specification statements (lists 1 and 2 in section 1.2.2) must precede all DATA statements, which must precede all statement function statements, which must precede all executable statements.
6. The last line of a program unit must be the END statement.

The rules for ordering FORTRAN statements are summarized in Figure 1-2.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td>DATA Statements</td></tr>
<tr><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Figure 1-2. Order of FORTRAN Statements

CHAPTER 2
FORTRAN CONCEPTS

The chapter discusses the concepts and terminology used to describe the *structure* and *elements* of a FORTRAN program.

2.1 FORTRAN Program Structure

2.1.1 Program Units and Procedures

The scope of many FORTRAN operations is defined to be a *program unit*. A program unit is either a *main program* or a *subprogram*.

A main program can start with a PROGRAM statement, though it need not. Subprograms start with either a FUNCTION, SUBROUTINE, or BLOCK DATA statement. A FORTRAN program must have one and only one main program and may have any number of subprograms (Figure 2-1).

Subroutines and functions are called *procedures*. Subroutines and 'external' functions are further defined to be *external procedures*. External procedures can be created outside a FORTRAN program also; for example, a FORTRAN program can call an external procedure written in PL/M-80 and stored in an ISIS-II system library. Procedures are discussed in detail in Chapter 5.

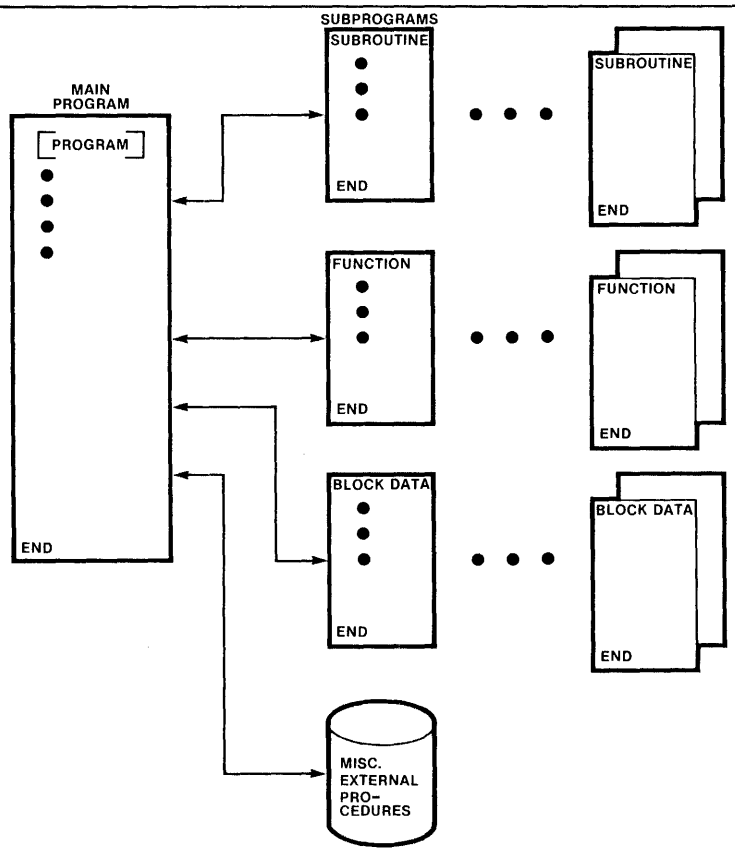


Fig. 2-1 Program Units

2.1.2 The PROGRAM Statement

The PROGRAM statement is used to name a program. This statement is optional, but when present must be the first statement of a main program. It has the format

```
PROGRAM name
```

where 'name' is the symbolic name of the program. Only one PROGRAM statement is allowed per program. The main program can contain any other statement *except* FUNCTION, SUBROUTINE, BLOCK DATA, SAVE, or RETURN.

The scope of symbolic names is discussed later in this chapter (section 2.5.5). For the moment, suffice it to say that the program name is 'global' to the entire executable program. It cannot be the same as the name of an external procedure, BLOCK DATA subprogram, common block, or 'local' symbol name in the main program.

2.1.3 Statements and Lines

A FORTRAN-80 source program is made up of *compiler controls* and FORTRAN *statements*.

Compiler controls direct the operation of a particular FORTRAN compiler, telling it what kind of output to produce, the form of list output, etc. Controls are discussed in detail in the compiler operator's manual, and a few will be mentioned in this manual where relevant. In general, controls can be embedded in FORTRAN source code and are identified by a '$' in the first character position ('column' 1).

All but two types of FORTRAN statement begin with a keyword and are identified by that keyword. For example, the PROGRAM statement just described begins with the keyword 'PROGRAM.' Only 'assignment' and 'statement function' statements do not begin with keywords. The 'AVG = HITS/AB' statement in Figure 1-1 is an example of an arithmetic assignment statement.

2.1.3.1 Statement Labels. Any statement can be labeled; any statement to be referenced from elsewhere in the program *must* be labeled. The label is a 1-5 digit, unsigned, nonzero, integer constant written anywhere in columns 1-5 of a statement's initial line. No two statements may have the same label within the same program unit.

2.1.3.2 Line Format. A FORTRAN statement consists of one or more *lines*. The first line of a statement is called the *initial* line; subsequent lines in the same statement are called *continuation* lines.

A FORTRAN line can have up to 72 characters. The first six character positions (referred to as 'columns' 1-6) contain information characterizing the line. The actual statement begins in column 7. A statement can extend over nine continuation lines (columns 7-72) for a total of 660 characters.

An integer anywhere in columns 1-5 is a statement label. If column 6 is blank or contains a '0,' the line is an initial line; if column 6 contains any other character, the line is a continuation line. Columns 1-5 of a continuation line must be blank.

Specific Intel FORTRAN compilers may allow exceptions to the standard line format. See Section F.2.4 of Appendix F for details.

2.2 FORTRAN Statement Elements

A FORTRAN statement can include the following elements:

- Statement identifier (keyword), such as PROGRAM or INTEGER
- Function identifiers, such as SQRT(A) or FLOAT(I)
- Constants, such as 3.142857 or 'STRING'
- Variables, such as A or AB
- Operators, such as * or .AND.
- Combinations of the above into value assignments, such as X=Y*Z, or into mathematical expressions, such as A*B+ SQRT(C)

Statement and function identifiers are the subjects of Chapters 3-6. Constants, variables, operators, and expressions are described in the remainder of this chapter and in Chapter 3.

2.2.1 Character Set

The FORTRAN-80 character set consists of the alphabetic characters A-Z, the digits 0-9, and the special characters listed below. The set of characters representable in the processor includes the printing ASCII graphics and the blank character. The collating sequence of the characters is that of the ASCII character set (Appendix E).

SPECIAL CHRACTERS

	Blank
=	Equal Sign
+	Plus
–	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Period
'	Single Quote
$	Dollar Sign
#	Pound Sign

Generally, blanks have no meaning in a FORTRAN statement and should be used to improve program readability. For example,

```
A=B*C+(D**2/E)
```

and

```
A  =  B*C  +  (D**2/E)
```

are equivalent statements.

Blanks are counted in the total characters allowed in a FORTRAN statement, however. They are also significant in character strings and in column 6 of the standard line format. They are not counted in the memory space occupied by a program.

2.2.2 Constants and Variables

The value of a *constant* does not change from one execution of a program to the next. The value of a *variable*, on the other hand, is subject to change during program execution or between runnings of the program. For example, in the statement

```
C = A**2 + B
```

the '2' is constant, whereas A, B, and C are variable and may change as the result of an earlier calculation or value assignment.

A constant appears as its actual value. A variable has a *symbolic name* that can be 1-6 alphanumeric characters. The first character must be alphabetic. Thus, all of the following are valid variable names:

```
K
XYZ
B52
ERROR8
STEP3
```

Every constant and variable has a *data type* and *length* associated with it. Arithmetic constants and variables also have an associated *number base.*

2.2.2.1 Data Types. Arithmetic constants and variables are of type *integer* or *real* (sometimes called 'fixed point' and 'floating point').

An integer constant is written without a decimal point and can be preceded by a '+' or '-' sign (0, 123, −34, +5). A '+' is assumed if no sign is present. Real constants include a decimal point and optional sign (5., .5, 0.5, .0005). An integer exponent preceded by an 'E' may follow a real constant and, in this case, the real constant need not have a decimal point (4E3). Again, the exponent may be signed.

4.2E3	(4.2×10^3 or 4200)
4.2E+3	(same as above)
4.2E-3	(4.2×10^{-3} or .0042)

The internal representation, the precision, and the range of real values conforms to the floating-point conventions established for the particular processor being used. See section F.1.

An integer variable name begins with one of the alphabetic characters 'I' through 'N.' Variables beginning with an alphabetic character other than I, J, K, L, M, or N are assumed to be type real. This implicit naming convention can be circumvented using *type statements*, however (see section 3.1).

Constants and variables can also be of type *logical* or type *character*.

Logical data may have only the values 'true' or 'false.' The possible forms of a logical constant are:

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

Character data is a string of any characters representable in the processor. The blank character is valid and significant in a character constant. A character constant has the form of a string of characters surrounded by apostrophes. An apostrophe within the string is represented by a double apostrophe. A character constant can have 1-255 characters.

```
'ARITHMETIC OVERFLOW ERROR'
'MURPHY'S LAW'
```

For the sake of compatibility with earlier versions of FORTRAN, FORTRAN-80 also supports *Hollerith* type data under the guise of arithmetic/logical types. The Hollerith type is summarized in Appendix C.

2.2.2.2 Data Length. A real value always occupies four bytes of memory (32 bits).

An integer may occupy one, two, or four bytes. For an integer variable, the length is specified when the variable type is defined (section 3.1) or by default (section F.2.6). If no number base is specified for an integer constant, the constant is assumed to be decimal and its length is the same as the integer variable default length. If a number base is specified for an integer constant (as described in the next subsection), the length is determined implicitly by the processor from the base and number of digits of the integer.

BASE	No. of Digits	Length
Binary	1-8	One Byte
	9-16	Two Bytes
	17-32	Four Bytes
Octal	1-3	One Byte
	4-6	Two Bytes
	7-11	Four Bytes
Decimal	1-3	One Byte
	4-6	Two Bytes
	7-11	Four Bytes
Hexadecimal	1-2	One Byte
	3-4	Two Bytes
	5-8	Four Bytes

A logical variable may occupy one, two, or four bytes depending on the length specification. The implicit length of a logical constant is not defined; the range of possible values does not depend on the length attribute.

Character data always occupies one byte for each character in the string. Each character in the string has a position numbered consecutively 1, 2, 3, etc., from left to right.

If a data representation requires several bytes of memory, the memory locations are consecutive.

2.2.2.3 Number Base. For arithmetic (integer or real) constants and variables, the *decimal* number base is assumed with the following exception. Integer constants can be *binary*, *octal*, or *hexadecimal* as well.

The possible forms of an integer constant are:

[*s*] *d*...

or

[*s*] #*d*...*b*

where:

s	is an optional + or − sign
d	is a digit or one of the letters A-F
b	designates the number base and is one of the letters D, B, O, Q, or H

If the number base specifier is absent or is the letter 'D,' the character string 'd...' can contain only the digits 0-9 and is interpreted as a decimal number. If the base specifier 'B' is present, the character string must contain only the digits 0 and 1 and is interpreted as a binary number. If 'O' or 'Q' is specified as the base, the number can contain only the digits 0-7 and is interpreted as an octal number. If 'H' is specified as the base, the number can contain the digits 0-9 or the letters A-F and is interpreted as a hexadecimal number.

The following are valid integer constants:

```
0
23
+64101
-#1401D
#10001010B
-#10001010B
-#4567Q
+#AF2CH
```

2.2.3 Arrays

Frequently, the programmer will want to refer to a group of data by one name and still be able to refer individually to elements in the group as necessary. Such a group is called an *array*.

An *array name* is the symbolic name assigned the array when it is defined in a DIMENSION statement, type statement, or COMMON statement (Chapter 3). An *array element* is one member in the group of data. An *array element name* is an array name qualified by a *subscript* enclosed in parentheses.

The following table could be produced by rewriting the program in Figure 1-1 to generate four 1-dimensional arrays. The table lists three players, times at bat, hits, and batting averages. Thus the array definitions for these four arrays could be PNAME(3), AB(3), HITS(3), and AVG(3).

PNAME	AB	HITS	AVG
GEHRIG	49	14	.286
OTT	60	21	.350
RUTH	54	18	.333

We can refer to any element in these arrays by using a subscript. For example, the hits for 'OTT' can be referenced as HITS(2) and the batting average for RUTH as AVG(3).

2.2.4 Expressions and Operators

An *expression* is a combination of numbers, symbols, and operators. It may include parentheses and may also include functions (discussed in Chapter 5). Expressions appear in assignment statements (e.g., A = *B* + *C*) as controls in certain data processing statements (e.g., IF *FLAG .NE. 3* GO TO 250), and in subroutine calls (CALL SUB(*X*+*1*, *Y*).

FORTRAN has four kinds of expressions: *arithmetic*, *relational*, *logical*, and *character*. The first three use arithmetic, relational, and logical operators, respectively. FORTRAN-80 also allows *bitwise Boolean operations* using logical operators on integer values.

2.2.4.1 Character Expressions. A character expression consists of either a character constant, a character variable reference, or a character array element reference. The expression may be enclosed in parentheses.

2.2.4.2 Arithmetic Expressions. An arithmetic expression performs a numeric computation. This computation is limited by the range and precision of numeric values representable in the processor (see section F.1). If any part of the computation produces values outside this range, the results are undefined.

Arithmetic operands must identify values of type integer or real.

2.2.4.2.1 Arithmetic Operators. The arithmetic operators are:

Operator	Meaning
**	Exponentiation
/	Division
*	Multiplication
+	Unary or binary addition
−	Unary or binary subtraction

The following expressions calculate the perimeter, area, and diagonal length of a square with side length 'S.'

```
SPERIM = 4*S
SQAREA = S**2
SQDIAG = SQRT(2*(S**2))
```

2.2.4.2.2 Type, Length and Interpretation of Arithmetic Expressions. The type of an arithmetic expression is determined from the types of the operands. When a '+' or '−' precedes a single operand, the type and length of the resulting expression is the same as the operand's. When an arithmetic operator links a pair of operands, the data type, length, and interpretation are as shown in Figures 2-2 and 2-3.

In these figures, 'I' and 'R' represent operands of type integer and real. 'I1' is a single-byte integer, 'I2' is a 2-byte integer, etc. FLOAT is an intrinsic function for converting an integer to a real number. For expressions involving subtraction, multiplication, or division, replace the '+' in Figure 2-2 with −, *, or /.

OP1 \ OP2	I1	I2	I4	R
I1	I1=I1+I1	I2=I1+I2	I4=I1+I4	R=FLOAT(I1)+R
I2	I2=I2+I1	I2=I2+I2	I4=I2+I4	R=FLOAT(I2)+R
I4	I4=I4+I1	I4=I4+I2	I4=I4+I4	R=FLOAT(I4)+R
R	R=R+FLOAT(I1)	R=R+FLOAT(I2)	R=R+FLOAT(I4)	R=R+R

Fig. 2-2 Type, Length, and Interpretation of (OP1 + OP2)

Figure 2-3 shows the result of exponentiation.

OP1 \ OP2	I1	I2	I4	R
I1	I1=I1**I1	I2=I1**I2	I4=I1**I4	R=FLOAT(I1)**R
I2	I2=I2**I1	I2=I2**I2	I4=I2**I4	R=FLOAT(I2)**R
I4	I4=I4**I1	I4=I4**I2	I4=I4**I4	R=FLOAT(I4)**R
R	R=R**I1	R=R**I2	R=R**I4	R=R**R

Fig. 2-3 Type, Length, and Interpretation of (OP1 OP2)**

As these figures indicate, mixed-mode arithmetic is done by converting both operands to the same type (the type of the result) before performing the operation. This conversion is unnecessary when a real number is raised to an integer power.

In the case of an integer divided by another integer, the remainder is truncated.

The value of 1/3 is 0
The value of 8/3 is 2
The value of -8/3 is -2

If the magnitude of an arithmetic result is too large for the processor to represent in the number of bytes shown in these figures, the result is undefined. See section F.1.

2.2.4.3 Relational Expressions

Relational expressions compare two arithmetic or two character expressions and return a TRUE or FALSE result of type logical.

2.2.4.3.1 Relational Operators. The relational operators are:

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

Relational expressions are commonly used in the IF statement (Chapter 4).

```
IF (NUMB .GT. 99) STOP
IF (PNAME .EQ. 'GEHRIG') PRINT 20, PNAME, AVG
```

2.2.4.3.2 Interpretation of Arithmetic Relational Expressions. An arithmetic relational expression is TRUE if the values of the operands satisfy the relational condition set up by the operator, and is FALSE otherwise.

If the operands are of different types, type conversion is similar to that of arithmetic expressions. The relational expression

```
EXP1 operator EXP2
```

is evaluated as if it were written

```
(EXP1 - EXP2) operator 0
```

where '0' is the same type as (EXP1 - EXP2) and 'operator' is the same relational operator in both expressions.

2.2.4.3.3 Interpretation of Character Relational Expressions. A character relational expression is TRUE if the values of the operands satisfy the relational condition set up by the operator, and is FALSE otherwise.

If two character operands have different lengths, the shorter is 'extended' to the length of the longer by adding blanks on the right of the character string. The character expression EXP1 is considered to be less than EXP2 if the value of EXP1 precedes the value of EXP2 in the ASCII collating sequence, and vice versa (Appendix E).

2.2.4.4 Logical Expressions. A logical expression performs a logical computation and returns a TRUE or FALSE result of type logical. This expression can be a single logical operand (logical constant, logical variable reference, logical array element reference, logical function reference, or relational expression) or a combination of logical operands joined by logical operators and parentheses.

2.2.4.4.1 Logical Operators. The logical operators are:

Operator	Meaning
.NOT.	Logical negation
.AND.	Logical conjunction
.OR.	Logical inclusive disjunction
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence

2.2.4.4.2 Value and Length of Logical Expressions. The value of a logical operand involving *.NOT.* is as follows:

OP1	.NOT. OP1
TRUE	FALSE
FALSE	TRUE

The logical expression has the opposite value as its operand.

The following example passes control to line 10 if the logical variable DONE is not true. Otherwise, execution stops.

```
10  FLAG = FLAG + 1
      .
      .
      .
    DONE = (FLAG .GT. 99)
    IF (.NOT. DONE) THEN
    GO TO 10
    ELSE
    STOP
    ENDIF
```

The value when two logical operands are combined by *.AND.* is as follows:

OP1	OP2	OP1 .AND. OP2
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

If both operands are true, the logical expression is true.

This example achieves the same effect as the last example.

```
IF (FLAG .GT. 0 .AND. FLAG .LE. 99) GO TO 10
STOP
```

The value when two logical operands are combined by *.OR.* is as follows:

OP1	OP2	OP1 .OR. OP2
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

If either operand is true, the logical expression is true.

The following statement branches on either of two conditions.

```
IF (FLAG .EQ. 50 .OR. FLAG .LE. 10) GO TO 250
```

The value when two logical operands are combined by *.EQV.* is as follows:

OP1	OP2	OP1 .EQV. OP2
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE

If both operands are logically the same, the logical expression is true.

The following statement returns whenever the two logical operands are logically equivalent.

```
IF (FLAG1 .EQV. FLAG2) RETURN
```

The value when two logical operands are combined by *.NEQV.* is as follows:

OP1	OP2	OP1 .NEQV. OP2
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

If both operands are logically different, the logical expression is true.

The following statement continues execution if the two operands are not equivalent.

```
IF (FLAG1 .NEQV. FLAG2) CONTINUE
```

Logical data may occupy one, two, or four bytes. The length of the value of a logical expression can be determined from Figure 2-4. In this figure, 'L1' is a 1-byte logical value, 'L2' is a 2-byte logical value, etc. To obtain the length of expressions involving .AND., .EQV., and .NEQV., replace all occurrences of .OR. in this figure with the applicable logical operator. In the case of .NOT., the resulting value has the same length as the operand.

OP1 \ OP2	L1	L2	L4
L1	L1 = L1 .OR. L1	L2 = L1 .OR. L2	L4 = L1 .OR. L4
L2	L2 = L2 .OR. L1	L2 = L2 .OR. L2	L4 = L2 .OR. L4
L4	L4 = L4 .OR. L1	L4 = L4 .OR. L2	L4 = L4 .OR. L4

Fig. 2-4 Length of (OP1 .OR. OP2)

2.2.4.5 Bitwise Boolean Operations.

Logical operators can be used with integer as well as logical operands. In this case, the resulting value is the bitwise complement, conjunction, inclusive disjunction, equivalence, and exclusive disjunction of the integer operand. This FORTRAN-80 feature is a machine-dependent extension and assumes that the processor represents integers in two's complement form. If the lengths of the two operands differ, the shorter operand is extended with zeros on the left.

If the operator is .NOT. the length of the integer result is the same as the length of the operand. The length of an expression consisting of two integer operands is as shown in Figure 2-2.

Operations between logical values and integer (Boolean) values are not allowed.

2.2.4.6 Precedence of Operators.

Expressions are generally evaluated left to right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. When two operators have equal precedence, the leftmost is evaluated first.

Parentheses can be used to override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost are evaluated first.

```
15/3 + 18/9 = 5 + 2 = 7
15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
```

The following lists the precedence of operators in descending order:

- Parenthesized expressions
- Exponentiation: **
- Multiplication/Division: *, /
- Addition/Subtraction: +, − (unary and binary)
- Relational Operators: .LT.,.LE.,.EQ.,.NE., .GT.,.GE.
- Logical/Boolean .NOT.
- Logical/Boolean .AND.
- Logical/Boolean .OR.
- Logical/Boolean .EQV., .NEQV.

Thus, the expression

```
D .OR. A + B .GE. C
```

is interpreted as though it were written

```
D .OR. ((A + B) .GE. C)
```

One exception to the left-to-right rule is the case where two or more exponentiations occur together.

```
A**B**C
```

In this case, the exponentiation is interpreted right-to-left as though it were written

```
A**(B**C)
```

2.2.5 Scope of Symbols

A symbolic name consists of one to six alphanumeric characters, the first of which must be alphabetic. Symbolic names may be *global* (that is, they may have a scope of the entire program) or they may be *local* to a program unit or statement function. A local symbol name can, therefore, represent different entities in different program units or in different statement functions.

The following symbolic names are global to the entire program and cannot be used in a local context.

- Main program name
- Subroutine names
- External function names
- BLOCK DATA subprogram names

The following symbolic names are local to the program unit in which they appear:

- Array names
- Variable names

- Statement function names
- Intrinsic function names
- Dummy procedure names

Variables appearing as dummy arguments in a statement function have a scope of that statement only.

Common block names are generally global, but do admit exceptions. A common block name in a program unit may also be the name of any local entity other than an intrinsic function in a function subprogram. An intrinsic function name may be used as a common block name, however, if the program unit does not reference that function. If a name is used for both a common block and a local entity, its appearance identifies only the local entity except in COMMON and SAVE statements.

2.3 Notational Conventions

This manual uses the following notational conventions to describe FORTRAN-80 statements and concepts:

- Special characters from the FORTRAN character set, uppercase letters, and uppercase words (keywords) are to be written as shown, except where otherwise noted.
- Lowercase letters and words indicate nonspecific arguments for which specific items must be substituted in actual statements.
- Brackets [ ] are used to indicate optional items.
- Ellipses (...) indicate that the preceding optional items can appear one or more times in succession.
- Blanks are used to improve readability, but have no significance.

As an example of the notation used, the description

CALL *sub* [([*arg*[,*arg*]...])]

means the following forms of the CALL statement are permissible:

CALL *sub*
CALL *sub* ( )
CALL *sub* (*arg*)
CALL *sub* (*arg*, *arg*)
CALL *sub* (*arg*, *arg*, *arg*)
etc.

When the actual CALL statement is written, specific entities would be substituted for 'sub' and each 'arg.'

```
CALL ROUT1(A, 3.72, 4.12)
```

CHAPTER 3
DEFINING VARIABLES, ARRAYS, AND MEMORY

This chapter describes the statements used to specify the types and lengths associated with symbolic names, how to assign values to these symbols, how to structure memory, and how to assign values to a block of memory. The statements described are:

- Type statements: REAL, INTEGER, LOGICAL, CHARACTER, IMPLICIT
- Array definition statement: DIMENSION
- Assignment statements: arithmetic, logical, and character assignment, ASSIGN, DATA
- Memory definition statements: COMMON, EQUIVALENCE, BLOCK DATA

3.1 Type Statements

The type statements REAL, INTEGER, LOGICAL, and CHARACTER are used to confirm or override the type implied by a symbolic name (i.e., the convention that names beginning with the letters I-N refer to integers—Section 2.2.2.1). The convention itself can be confirmed or changed by the IMPLICIT statement.

Type statements can also specify data length or array dimension information.

Specifying the symbolic name of a variable, array, external function, or statement function in a type statement establishes the type of that name for all appearances in the program unit. The type of a name cannot be specified explicitly more than once in a program unit. Program names and subroutine names cannot appear in a type statement.

3.1.1 REAL Type Statement

The REAL type statement has the format:

REAL *name* [,*name*]...

where 'name' is the symbolic name of a real variable, array, array declarator, function, or dummy procedure.

Examples:

```
      REAL  TEMP

      REAL NUMB1, NUMB2, NUMB3
C REAL OVERRIDES IMPLICIT TYPING OF NUMB = INTEGER
```

3.1.2 INTEGER Type Statement

The INTEGER type statement has the format:

INTEGER [**len* [,]] *name* [,*name*]...

where 'name' is one of the forms

v[**len*]

ary[(*d*)][**len*]

and

v is an integer variable, function, or dummy procedure name

ary is an array name

ary(d) is an array declarator

len is the length in bytes of an integer variable or array element. Its value must be 1, 2, or 4.

The length specification immediately following the keyword INTEGER applies to each item in the statement not having its own length specification. A length specification immediately following an item is the length for that item only. For an array, the length applies to each array element. If no length is specified, the current default length of an integer variable or array element is assumed (see section F.2.6).

Examples:

```
INTEGER TOTALS

INTEGER*1  DIGITS(10)

INTEGER*4  TOTALS, SUBS*2, DIGITS(10)*1
```

3.1.3 LOGICAL Type Statement

The LOGICAL type statement has the format:

LOGICAL [**len* [,]] *name* [,*name*]...

where 'name' is one of the forms

v[**len*]

ary[(*d*)][**len*]

and

v is a logical variable, function, or dummy procedure name

ary is an array name

ary(*d*) is an array declarator

len is the length in bytes of a logical variable or array element. Its value must be 1, 2, or 4.

The length specification immediately following the keyword LOGICAL applies to each item in the statement not having its own length specification. A length specification immediately following an item is the length for that item only. For an array, the length applies to each array element. If no length is specified, the current default length of a logical variable or array element is assumed (see section F.2.6).

Examples:

```
LOGICAL*2 FLAG

LOGICAL*1 FLAGS(10)

LOGICAL*4 FLAG1, FLAG2, SWITCH(5)*1
```

3.1.4 CHARACTER Type Statement

The CHARACTER type statement has the format:

CHARACTER [**len*[,]] *name* [,*name*]...

where 'name' is one of the forms:

v[**len*]

ary[(*d*)][**len*]

and

v	is a variable name
ary	is an array name
ary(*d*)	is an array declarator
len	is the length (number of characters) of a character variable or character array element.

The length specification immediately following the keyword CHARACTER applies to each item in the statement not having its own length specification. A length specification immediately following an item is the length for that item only. For an array, the length applies to each array element. If no length is specified, the standard length of a character (one byte) is assumed.

Examples:

```
CHARACTER*15 STRING

CHARACTER*12 NAMES(50), CITIES(50), STATES(50)*5

CHARACTER LETTER
```

3.1.5 IMPLICIT Statement

An IMPLICIT statement defines the type and length for symbolic names (except intrinsic function names) that begin with the letter(s) specified by IMPLICIT. IMPLICIT types can be overridden, however, by type statements or by an explicit type specification in a FUNCTION statement. The length specified in an IMPLICIT statement can also be overridden by an INTEGER, LOGICAL, or CHARACTER statement containing the same symbolic name.

The IMPLICIT statement has the format:

IMPLICIT *typ* (*let* [,*let*]...) [,*typ* (*let* [,*let*]...)]...

where

typ is INTEGER[**len*] , REAL, LOGICAL[**len*], or CHARACTER[* *len*]

let is a single letter or a range of letters in alphabetical order (e.g., C, I-M, N-Z)

len is the length of the items (in bytes). This must be 1, 2, or 4 for INTEGER or LOGICAL, and is the length of a CHARACTER string. If no length is specified, a length of one is assumed for CHARACTER and the current default length for INTEGER and LOGICAL (section F.2.6).

The IMPLICIT statement applies only to the program unit in which it appears and *must precede all other specification statements in that program unit*. The program unit can have more than one IMPLICIT statement, but the same letter cannot be specified more than once.

Example:

```
IMPLICIT REAL(A-B, D-H), CHARACTER (C)
IMPLICIT INTEGER (I-N), LOGICAL (O-Z)
```

Unless these implicit definitions are overridden, the following symbols would have the types indicated.

AVG	(REAL)
CNAME	(CHARACTER)
FPNUM	(REAL)
INUM	(INTEGER)
PFLAG	(LOGICAL)

3.2 Array Definition

An array is defined by assigning a symbolic name to the array and specifying its dimensions. One way to do this is with type statements:

```
CHARACTER TICTAC(3,3)

LOGICAL TABLE(2,3,3)
```

Arrays can also be defined by the COMMON statement (section 3.4.2) and by the DIMENSION statement.

In any case, a symbol can be used only once in a program unit as an array name *in an array declarator* (section 3.2.1). The symbol 'TICTAC' in the example above could not be defined in a DIMENSION statement as well as in the CHARACTER type statement. The array name could, of course, appear as a reference or array element name elsewhere:

```
TICTAC(3,2) = 'X'
```

By 'array element name' we mean an array name qualified by a subscript in parentheses as shown in the example above. An array name not qualified by a subscript identifies the entire array, with one exception. In an EQUIVALENCE statement, an array name not qualified by a subscript identifies the first element of the array.

An array name is local to the program unit in which it is declared.

3.2.1 DIMENSION Statement

The format of the DIMENSION statement is:

DIMENSION *ary(d)* [,*ary(d)*]...

where each 'ary(d)' is an array declarator of the form

ary(d [,*d*] ...)

and

ary is the symbolic name of the array

d is a dimension declarator

Dimension declarators are discussed in more detail below (section 3.2.3). In general, they indicate the number of dimensions in the array and the number of elements (or upper bound) of each dimension. The maximum number of dimensions is seven.

Examples:

```
LOGICAL TABLE
INTEGER ARRAY
DIMENSION  TABLE(2,3), ARRAY(3,3,3)
```

3.2.2 Kinds of Array Declarators

An array declarator ('ary(d)' in the DIMENSION format) is either a *constant*, *adjustable*, or *assumed-size* array declarator.

In a constant array declarator, each of the dimension bounds is a constant. An adjustable array declarator contains one or more variables as bounds:

```
ARRAY(3,MIDDLE,THIRD)
```

An assumed-size array declarator is either constant or adjustable, but the upper bound of the last dimension is an asterisk.

```
ARRAY(3,3,*)
```

An array name may be used as a dummy argument in a FUNCTION or SUBROUTINE subprogram. Thus, a program can have *actual* array declarators and *dummy* array declarators. An actual array declarator must be a constant array declarator, whereas dummy array declarators may be constant, adjustable, or assumed size. Like actual array declarators, dummy declarators are permitted in DIMENSION or type statements, but unlike actuals, they cannot appear in COMMON statements. A variable name used as a dimension bound of an array must also appear in the subprogram's dummy argument list or in a common block in the subprogram. The latter requirements can be avoided using the asterisk feature for the last dimension, thereby gaining some program efficiency.

3.2.3 Properties of Arrays

The examples following the description of the DIMENSION statement specify the *types* of the array names and, by implication, the types of the elements in the arrays. They also specify (by default) the *lengths* of the elements.

The remaining properties of the array are determined from the dimension declarator. These are:

- The number of dimensions in the array
- The size of each dimension
- The total number of array elements

The number of dimensions equals the number of dimension declarators in the array declarator. Thus, the array

```
TABLE(4,4)
```

has two dimensions.

The size of a dimension is the same as the value 'd' in the array declarator format; it is also the same as the 'upper dimension bound.' The lower dimension bound is assumed to have the value *one*. The upper bound may be an asterisk in the case of an assumed-size array declarator.

The size of an array can generally be computed as the product of the sizes of the dimensions specified by the array declarator. Thus

```
ARRAY(3,3,3)
```

would have 27 elements. The number of elements in an assumed-size dummy array can be determined as follows:

- If the actual argument corresponding to the dummy array is an array name, the size is that of the actual array;
- If the actual argument is an array element name with a subscript value of 'p' in an array of size 'n,' the size of the dummy array is n + l-p.

Array elements are stored sequentially. For example, in the following sequence

```
DIMENSION TABLE(3,3)
TABLE(3,1) = 2.9
TABLE(2,3) = 7.3
```

'2.9' would be assigned to the third storage location in the block whose low address is 'TABLE,' and '7.3' would be assigned to the eighth location.

```
(1,1)(2,1)2.9(1,2)(2,2)(3,2)(1,3)7.3(3,3)
```

The total number of bytes in an array is the number of elements in the array multiplied by the number of bytes occupied by each element.

3.2.4 Referencing Array Elements

Array elements are referenced by qualifying an array name with a subscript in the form

ary (*s* [,*s*] ...)

where 'ary' is an array name and 's' is a subscript. The number of subscripts must equal the number of dimensions in the array declarator.

Each subscript is an integer expression in the range $1 \leq s \leq$ upper-bound. If the upper dimension bound of a dummy array is an asterisk, the value of the corresponding subscript must not exceed the size of the corresponding actual array.

Examples:

```
ARRAY(2,6) = A
ARRAY(I + J, 3) = 8
ARRAY(M,M+N,M-N) = A + SQRT(B)
```

Figure 3-1 can be used to calculate which element in the storage sequence of array elements is being referenced. In this figure, 'n' is the number of dimensions ($1 \leq n \leq 7$), 'd' is the value of the upper dimension bound, and 's' is the subscript expression.

n	Dimension Declarator	Subscript	Element Referenced
1	(d1)	(s1)	s1
2	(d1,d2)	(s1,s2)	1 + (s1-1) +(s2-1)*d1
3	(d1,d2,d3)	(s1,s2,s3)	1 + (s2-1) +(s2-1)*d2 +(s3-1)*d2*d1
. . .	. . .	. . .	. . .
n	(d1,...,dn)	(s1,...,sn)	1+(s1-1) +(s2-1)*d1 +(s3-1)*d1*d2 +... +(sn-1)*dn-1 +dn-2*...d1

Fig. 3-1 Subscript Reference

3.3 Assignment Statements

The type statements correlate a type and length with a symbolic name. The statements described in this section assign *values* to variables, arrays, or array elements.

Assignment statements are used to assign arithmetic, logical, or character values to variables and array elements. These statements have no keyword.

The ASSIGN statement is used to assign a numerical statement label to an integer-variable symbol name.

The DATA statement is used to *initialize* variables, arrays, and array elements to specific values.

3.3.1 Arithmetic Assignment Statement

The arithmetic assignment statement closely resembles a conventional arithmetic formula. Its format is:

v = exp

where

v is the name of a variable or array element of type integer or real

exp is an arithmetic expression

The '=' in FORTRAN has the sense 'is assigned the value' rather than 'is equal to.' Thus

I = I + 1

is a perfectly correct FORTRAN statement.

Execution of an arithmetic assignment statement causes evaluation of the expression 'exp' according to the rules listed in Chapter 2 (see Figures 2-2 and 2-3), conversion of 'exp' to the type of 'v,' and definition and assignment of 'v' with the resulting value, as shown in Figure 3-2. IFIX and FLOAT in this figure are intrinsic functons for converting a real number to an integer and an integer to a real number, respectively.

TYPE OF 'v'	TYPE OF 'exp'	RESULT
INTEGER	INTEGER	exp
REAL	REAL	exp
INTEGER	REAL	IFIX (exp)
REAL	INTEGER	FLOAT (exp)

Fig. 3-2 Result of 'v = exp'

Example:

```
      I = 3
      C = I + SQRT(25.0)
      C = C**2
C AS  A RESULT OF THESE CALCULATIONS C = 64.0
```

If the value of 'exp' is too large to be assigned to 'v' the result is undefined. This may happen when the length of 'v' is too short to contain the processor representation of the integer value (see section F.1).

If the length of 'v' is longer, the length of 'exp' is converted to the length of 'v' while preserving its value.

Example:

```
      INTEGER*1 M(1000)
      INTEGER*4 N
      N = 65
      M(200) = N
C LENGTH OF 'N' IS CONVERTED TO ONE BYTE
```

3.3.2 Logical Assignment Statement

The logical assignment statement assigns the value .TRUE. or .FALSE. to a logical variable or array element. It has the format

v = exp

where

v is the name of a logical variable or logical array element

exp is a logical expression

Examples:

```
      LOGICAL FLAG, TABLE(3,3)
      FLAG = (INT1 .NE. 1 .AND. INT2 .EQ. 1)
C FLAG IS .TRUE. IF BOTH CONDITIONS ARE TRUE AND
C OTHERWISE IS .FALSE.
      TABLE(1,3) = .FALSE.
      TABLE(1,2) = FLAG
```

3.3.3 Character Assignment Statement

The character assignment statement assigns a character constant, variable name, or array element name to a character variable or array element. Its format is:

v = char

where

v is the name of a character variable or character array element

char is a character constant, character variable name, or character array element name.

None of the character positions being defined in 'v' can be referenced in 'char.' The two sides of the assignment may have different lengths, however. If 'v' is longer than 'char,' the latter is padded on the right with blank characters. If 'v' is shorter, 'char' is truncated on the right until it fits into 'v.'

Examples:

```
CHARACTER*10 NAMES(4), MGR
MGR = 'STENGEL'
NAMES(1) = 'GEHRIG'
NAMES(2) = 'OTT'
NAMES(3) = 'RUTH'
NAMES(4) = MGR
```

3.3.4 ASSIGN Statement

The ASSIGN statement is the only way to assign a statement label to a symbolic name. The symbolic name can then be referenced in a GO TO statement or as a format identifier in an input/output statement. To use the symbolic name in any other context, it must first be redefined as an integer value in an arithmetic assignment statement.

The ASSIGN statement has the format:

ASSIGN *stl* TO *name*

where

stl is a statement label (1-5 digits)

name is an integer variable name

The statement label must be the label of an *executable* statement or a *FORMAT* statement in the same program unit as the ASSIGN statement. The variable 'name' must not be declared as length INTEGER*1.

An integer variable defined with a statement label value may be redefined with the same or a different statement label value, as well as with an integer value.

Examples:

```
      ASSIGN 1010 TO LOOP1
      GO TO LOOP1
      .
      .
      GO TO LOOP1(1000,1010,1020)

      IF(.NOT.DONE) THEN
      ASSIGN 20 TO WRFORM
      ELSE
      ASSIGN 25 TO WRFORM
      END IF
      WRITE (6,WRFORM) PNAME, AVG
20    FORMAT...
25    FORMAT...
```

3.3.5 DATA Statement

The DATA statement gives the initial values of variables, arrays, and array elements. Dummy arguments and functions cannot be initialized by DATA. Shared, or 'common,' memory can be initialized by DATA statements if the DATA statements are part of a BLOCK DATA subprogram (section 3.4.3).

The DATA statement must appear in a program unit after the specification statements and before any statement function or executable statements.

The DATA statement has the format:

DATA *nlist /clist/* [[,] *nlist /clist/*]...

where 'nlist' is a list of variable names, array names, and array element names, and 'clist' has the form:

[*r**]*c*[,[*r**]*c*]...

where

c is a constant (including a Hollerith constant)

r is a 'repeat' character and is a nonzero, unsigned, integer constant; 'r*c' is equivalent to 'r' successive repetitions of the constant 'c.'

Items in DATA lists must agree in *number*, *type*, and *length*.

'Nlist' and 'clist' must have the same *number* of items, as the lists correspond one-to-one. If 'nlist' contains an array name without a subscript, 'clist' must have one constant for each element of that array (but see section F.2.9.2). Any subscript that *is* specified must be an integer constant.

The *type* of a name specified in 'nlist' must agree with the type of the corresponding constant in 'clist,' except that an item of any type can be initialized to a Hollerith constant.

Given a *length* 'g' of a variable or array element in 'nlist,' then the length 'n' of its corresponding initial Hollerith constant in 'clist' must be less than or equal to 'g.' If 'n' is less than 'g,' the constant is padded on the right with blanks until the lengths are equal. Note that initialization of a character in a variable or array element initializes that entire item.

A variable or array element cannot be initialized more than once in a program. If two symbols are associated, only one may be initialized.

Examples:

```
      DATA  I,J,K /10,20,30/

      CHARACTER*10  NAMES(3)
      DATA  NAMES /'GEHRIG', 'OTT', 'RUTH'/

      INTEGER*1  ZEROS (10)
      DATA ZEROS /10*0/ I,J,K /10,20,30/
     &NAMES(2) /'OTT'/
C  AMPERSAND USED TO CONTINUE STATEMENT

      LOGICAL TABLE(3)
      DATA TABLE /.TRUE., .TRUE., .FALSE./
```

3.4 Memory Definition

The DATA and assignment statements assign values to specific items. FORTRAN also includes three statements for establishing memory areas and initializing these areas. These are the EQUIVALENCE, COMMON, and BLOCK DATA statements.

EQUIVALENCE is used to associate two or more items in memory, such as associating a variable name with an array element. Its scope is the program unit in which it appears. The COMMON statement can be used to associate items in *different* program units, allowing common use of data and memory through an entire program (for example, a common data base or table). BLOCK DATA defines a BLOCK DATA subprogram, which can assign initial values to items in common memory.

3.4.1 EQUIVALENCE Statement

The EQUIVALENCE statement allows items in a program unit to share memory. All entities listed in the EQUIVALENCE statement share the same start address in memory (even if they are of unequal lengths).

The format of the EQUIVALENCE statement is:

EQUIVALENCE (*nlist*) [,(*nlist*)] ...

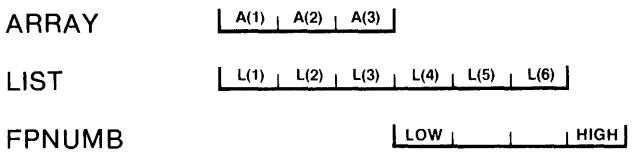
where 'nlist' is a list of variable names, array names or array element names. The latter may only be subscripted by integer constants. The use of an array name unqualified by a subscript is the same as a reference to the first element of the array. Function names and the names of dummy arguments may not be listed.

Equivalenced items may have different data types, although this is not recommended. The EQUIVALENCE statement does no type conversion.

Example:

```
INTEGER*1, ARRAY(3), LIST(6)
REAL FPNUMB
EQUIVALENCE (ARRAY,LIST), (LIST(4),FPNUMB)
```

The resulting memory allocation would be:



The EQUIVALENCE statement must not cause the same storage item to occur more than once in a memory sequence, nor can it result in the splitting of a memory sequence.

Examples:

```
C THE FOLLOWING EXAMPLES ARE INVALID
      DIMENSION ARRAY(3)
      EQUIVALENCE (ARRAY(1),FPNUMB),(ARRAY(2),FPNUMB)
C FPNUMB OCCURS MORE THAN ONCE IN MEMORY SEQUENCE
      REAL TABLE(2), ROOTS(3)
      EQUIVALENCE (TABLE(1),ROOTS(1)),(TABLE(2),
     &ROOTS(3))
C MEMORY SEQUENCE IS SPLIT
```

3.4.2 COMMON Statement

Common memory blocks allow data and memory to be shared throughout an entire program. The COMMON statement defines common blocks that may be either named or unnamed (that is, 'blank').

The format of the COMMON statement is:

COMMON [/[*cb*]/] *nlist* [[,]/ [*cb*]/ *nlist*]...

where

cb is the name of a common block.

nlist is a list of variable names, array names, or array declarators. Function names and the names of dummy arguments cannot be listed.

The items in 'nlist' following a common block name (or omitted name) are declared to be in that block (or in blank common). If a common block name is omitted, the statement refers to the blank common block. If the first common block name is omitted in the above format, the slashes may be omitted also. The slashes must be present, however, if blank common is specified as other than the first common block.

The same common block name (or blank) can appear in other COMMON statements, either in the same program unit or in other program units. This is how items in different program units are associated in memory. All common blocks having the same name also have the same starting address in memory. The same is true of all declarations of blank common. Only one blank common can exist in the final linked program.

3.4.2.1 Common Block Memory Sequence.

A common block memory sequence consists of the memory of all items listed in the COMMON statement(s) for that common block, in the order of their appearance within the COMMON statement(s).

An EQUIVALENCE statement may cause a common block to be extended. This is done by adding memory beyond the highest location in the common block. An EQUIVALENCE statement must not cause two different common blocks in the same program unit to be associated. Names associated with a name in a common block are considered to be part of that common block.

3.4.2.2 Named and Blank Common Blocks.

Named and blank common blocks are treated differently in several respects.

- Within a program, all common blocks having the same name must also be the same size. Blank common blocks may be different sizes.
- Executing a RETURN or END statement sometimes causes items in named common blocks to become undefined. This cannot occur with blank common.
- DATA statements in a BLOCK DATA subprogram can only initialize items in named common blocks.

Examples:

```
            COMMON /BLOCK1/A,B,ROOTS // TABLES(3,3)
            LOGICAL LOGICS(3,3),Z
            COMMON X,Y,Z
C PREVIOUS STATEMENT EXTENDS BLANK COMMON
            COMMON /BLOCK1/C
C 'BLOCK1' IS NOW EXTENDED BY LENGTH OF 'C'
            EQUIVALENCE (Z,LOGICS(1,1))
C BLANK COMMON EXTENDED AGAIN—BY 8 LOGICAL
C ARRAY ELEMENTS
```

3.4.3 BLOCK DATA Subprograms

BLOCK DATA subprograms are used to initialize variables and array elements in named common blocks. The first statement of such a subprogram is the BLOCK DATA statement, which may or may not name the BLOCK DATA subprogram. The last statement must be the END statement. The only other statements permitted in a BLOCK DATA subprogram are IMPLICIT, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, and the type statements.

3.4.4 BLOCK DATA Statement

The format of the BLOCK DATA statement is:

BLOCK DATA [*name*]

where 'name' is the symbolic name of the BLOCK DATA subprogram.

Since 'name' is global, it must not be the same as the name of an external procedure, main program, common block, or another BLOCK DATA subprogram. Only one unnamed BLOCK DATA subprogram is permitted per executable program.

Only an item in a common block may appear in a DIMENSION, EQUIVALENCE, or type statement in a BLOCK DATA subprogram. Only an item in a *named* common BLOCK can be initialized in a BLOCK DATA subprogram.

If a named common block is initialized, all items in the block must be listed, even if they are not all initialized. More than one named common block may have items initialized in a single block data subprogram, but the same named common block may not be specified in more than one block data subprogram.

Examples:

```
      BLOCK DATA BLK1
      LOGICAL FLAGS(3)
      INTEGER ZEROS(10),RESULTS
      COMMON /BLOCK1/ FLAGS,ZEROS,RESULT
      DATA FLAGS/.TRUE., .TRUE., .FALSE./
     &ZEROS/10*0/
      END
```

CHAPTER 4
PROGRAM EXECUTION CONTROLS

FORTRAN includes 16 statements, or statement variations, for controlling program execution. These are statements that transfer control (GO TO, IF, and their variations), regulate execution loops (DO, CONTINUE), and terminate program execution (PAUSE, STOP, END).

4.1 Transferring Program Control

The statements in this group pass control to another part of the program, in some cases only when a stated condition is met. These alternatives are usually referred to as conditional and unconditional branching. Some statements also allow an alternative set of operations to be performed if the stated conditions are *not* met.

The statements that transfer program control are:

Unconditional GO TO	Block IF
Computed GO TO	ELSE IF
Assigned GO TO	ELSE
Arithmetic IF	END IF
Logical IF	

4.1.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the next statement to be executed. It has the format

GO TO *stl*

where 'stl' is the statement label of an executable statement in the same program unit as the GO TO statement.

Example:

```
GO TO 1010
```

4.1.2 Computed GO TO Statement

The computed GO TO statement branches to one of several executable statements based on the value of a controlling integer expression. The format of the computed GO TO is

GO TO (*stl* [,*stl*] ...)[,]*exp*

where

exp is an integer expression

stl is the statement label of an executable statement in the same program unit as the computed GO TO.

The same statement label may appear more than once in the statement. If the integer expression has a value in the range $1 \leq exp \leq n$ (where 'n' is the number of statement labels in the list), control passes to the statement pointed to by 'exp.' If 'exp' is outside this range, execution continues with the statement following the GO TO.

Examples:

```
      GO TO (1010,1020,1030) K
C  IF K = 2, FOR EXAMPLE, CONTROL PASSES TO STATEMENT
C  1020
      INTEGER*1 SWITCH
      SWITCH = K/J
      GO TO (10, 500, 500, 10, 10) SWITCH
C  NOTE THAT 'J' MUST BE .LE. 'K' IN THIS EXAMPLE
      GO TO (10, 500, 600, 500) K*L + 1
```

4.1.3 Assigned GO TO Statement

The assigned GO TO statement is used with the ASSIGN statement. The assigned GO TO is similar to the computed GO TO, but in this case the control is an integer variable name. Before the assigned GO TO is executed, the variable name must be defined with the value of a statement label by an ASSIGN statement in the same program unit.

The format of the assigned GO TO statement is

GO TO *name* [[,] (*stl* [,*stl*]...)]

where

name is an integer variable name

stl is the statement label of an executable statement in the same program unit as the assigned GO TO.

The same statement label may appear more than once in the statement. If the parenthesized list of statement labels is present, the statement label assigned to 'name' must be one of the labels in the list.

Examples:

```
ASSIGN 10 TO START
GO TO START
ASSIGN 999 TO DONE
GO TO DONE (500, 600, 999)
```

4.1.4 Arithmetic IF Statement

The arithmetic IF statement behaves similarly to a computed GO TO. Control is transferred to one of three possible statements based on the value of a controlling expression. The format of the arithmetic IF is

IF (*exp*) *s1, s2, s3*

where

exp is an integer or real expression

s is the statement label of an executable statement in the same program unit as the arithmetic IF.

The same statement label may appear more than once in the statement.

If the value of 'exp' is less than zero, control passes to the first statement listed; if it equals zero, control passes to the second statement; if 'exp' is greater than zero, control passes to the third statement.

Examples:

```
IF (A + B) 1010, 1020, 1030

SWITCH = A**2 - B**2
IF (SWITCH) 100,200,300
```

4.1.5 Logical IF Statement

We have given several examples of logical IF statements already in this manual. In effect, if the logical expression evaluated is TRUE, a specified statement is executed next. If the logical expression is FALSE, execution continues with the statement following the logical IF statement.

The format of the logical IF statement is

IF (*exp*) *stmt*

where

exp is a logical expression

stmt is any executable statement except DO, a block IF, ELSE IF, ELSE, END IF, END, or other logical IF

A function reference in the controlling logical expression is permitted to affect parameters in the statement 'stmt.'

Examples:

```
IF (SWITCH .EQ. 1) GO TO FINISH
IF (SWITCH .NE. 1) WRITE (6,20) TOTALS

LOGICAL DONE
DONE = (A**2 .GT. B**2)
IF (DONE) PAUSE
IF (.NOT. DONE) GO TO START
```

4.1.6 IF, ELSE IF, and ELSE Blocks

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control program execution.

Together with other executable FORTRAN statements, they can form 'IF blocks,' 'ELSE IF blocks,' or 'ELSE blocks,' the first statements of which must be IF, ELSE IF, or ELSE, respectively.

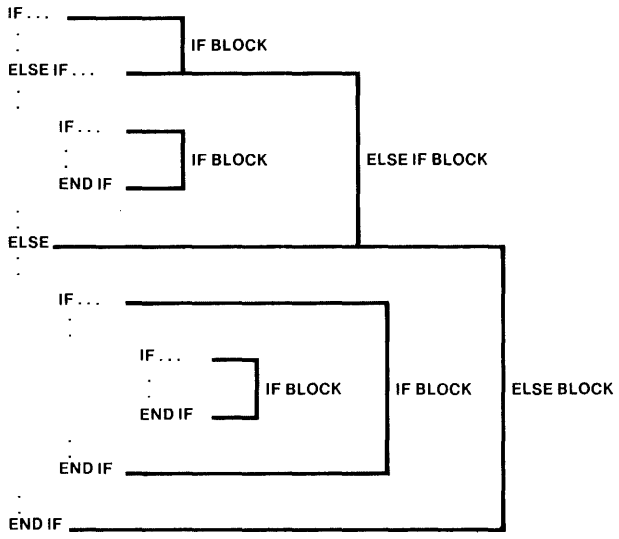
These blocks may be 'nested.' For example, an IF block may contain another IF block, which may contain another IF block, etc. These blocks may also be empty. For example, there may be no executable statements at all between a block IF statement and its corresponding END IF statement.

An IF block consists of all the executable statements after the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement on the same nesting level as the block IF statement.

An ELSE IF block consists of all the executable statements after the ELSE IF statement up to, but not including, the nest ELSE IF, ELSE, or END IF statement that has the same nesting level as the ELSE IF statement.

An ELSE block consists of all the executable statements after the ELSE statement up to, but not including, the next END IF statement that has the same nesting level as the ELSE statement.

For each block IF statement, there must be a corresponding END IF statement in the same program unit.



4.1.7 Block IF Statement

The format of the block IF is

IF (*exp*) THEN

where 'exp' is a logical expression.

If the value of 'exp' is TRUE, execution continues with the first statement of the IF block. If the IF happens to be empty and 'exp' is TRUE, control passes to the next END IF statement on the same nesting level as the block IF statement. If 'exp' is FALSE, control passes to the next ELSE IF, ELSE, or END IF on the same nesting level as the block IF statement.

Control cannot be transferred into an IF block from outside the IF block (but see section F.2.9.3).

If the last statement in the IF block does not transfer control to another executable statement, control passes to the next END IF statement that has the same nesting level as the block IF statement.

An example if shown following the description of the END IF statement.

4.1.8 ELSE IF Statement

The ELSE IF statement has the format

ELSE IF (*exp*) THEN

where 'exp' is a logical expression.

If 'exp' is TRUE, normal execution continues with the first statement of the ELSE IF block. If 'exp' is FALSE, control passes to the next ELSE IF, ELSE, or END IF statement having the same level as the ELSE IF statement.

Control cannot be transferred into an ELSE IF block from outside the block (but see section F.2.9.3). The statement label, if any, of the ELSE IF statement cannot be referenced by another statement.

If the last statement in the ELSE IF block does not pass control to another executable statement, control passes to the next END IF statement having the same nesting level as the ELSE IF statement.

An example is shown following the description of the END IF statement.

4.1.9 ELSE Statement

Executing the ELSE statement has no effect; execution simply continues. Its format is

```
ELSE
```

The ELSE block is terminated by an END IF statement of the same level as the ELSE statement. This END IF statement must appear before the appearance of an ELSE IF or another ELSE statement of the same nesting level.

Control cannot be transferred into an ELSE block from outside the block (but see section F.2.9.3). The statement label, if any, of the ELSE statement cannot be referenced by another statement.

An example is shown following the description of the END IF statement.

4.1.10 END IF Statement

Executing the END IF statement has no effect; normal execution continues. The END IF statement acts as a terminator for IF, ELSE IF, and ELSE blocks. Each block IF statement must have a corresponding END IF statement, that is, an END IF statement of the same nesting level as the block IF statement.

The format of the END IF statement is

```
END IF
```

Example:

```
      IF (FLAG .EQ. 3) THEN
            WRITE (6,20) NAME(3), PAY(3)
20          FORMAT...
      ELSE IF (FLAG .EQ. 2) THEN
            WRITE (6,40) FEDTAX
40          FORMAT...
      ELSE
            WRITE(6,60) FICA
60          FORMAT...
      ENDIF
```

4.2 Loop Control Statements

Frequently, a series of operations must be repeated several times (for example, reading a series of entries from an input device and extracting information selectively). Rather than repeat the statements to perform these operations for each entry, one can create a loop that causes the same statements to be performed over and over until all entries have been read and processed. This is the function of the DO statement.

The CONTINUE statement described in this section is ordinarily used with the DO statement, though it is not limited to this use.

4.2.1 Operation of a DO Loop

The first statement of a DO loop is the DO statement itself. The last statement is a labeled statement whose label is specified in the DO statement. These two statements define the range of the DO loop. The statements making up the body of the DO loop are executed a specific number of times, as defined in the DO statement.

The DO statement includes three values: an initial loop index value, a loop termination value, and an amount by which the initial value is to be incremented or decremented. Each time the loop is performed, a 'DO variable,' previously initialized to the initial index value, is increased or decreased by the increment/decrement value until the loop termination value is reached. Program execution then continues with the statement following the last statement of the DO loop.

This sequence describes the most common operation of a DO loop. A DO loop can also terminate operation as the result of a RETURN statement executed within the loop, a transfer of control outside the loop, execution of a STOP statement in the program, or program termination for any other reason.

Program control cannot be transferred into a DO loop.

4.2.2 DO Statement

The format of the DO statement is

DO *stl* [,] *var* = *e1, e2* [,*e3*]

where

stl is the statement label of an executable statement and is the label of the last statement in the DO loop.

var is the name of an integer variable, called the 'DO variable.'

e is an integer expression.

In this format, 'e1' is the initial loop index value, 'e2' is the loop termination value, and 'e3' is the loop increment/ decrement amount. If 'e3' is not specified, an increment of one is assumed. The values of 'e1' and 'e2' can be specified such that no iterations are performed (but see the discussion of the DO77/DO66 compiler controls in section F.2.5).

The last statement of a DO loop, statement 'stl,' must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the last statement of the DO loop is a logical IF statement, it can contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

DO loops may be nested, that is, a DO loop can contain another DO loop, etc. If a DO statement appears within the range of another DO loop, the loop specified by the second DO statement must be within the range of the outer DO loop. DO loops can share the same last statement.

If a DO statement lies within an IF block, ELSE IF block, or ELSE block, the range of the DO loop must be entirely within that block.

If a block IF statement is within the range of a DO loop, its corresponding END IF statement must also be within the range of the DO loop.

Examples:

The first example demonstrates the looping process.

```
        N = 0
        DO 100 I = 1,10
        J = I
        DO 100 K = 1,5
        L = K
100     N = N + 1
110     CONTINUE
```

When looping is completed, I = 11, J = 10, K = 6, L = 5, and N = 50. Note the use of blanks to isolate the nested DO loop in this example.

The following example is another program to compute batting averages for 25 players (one team). Compare this program to Figure 1-1...

```
        CHARACTER*12 PNAME
        DO 30  I = 1,25
        READ 10, PNAME, AB, HITS
10      FORMAT (A12, 2(2X, F3.0))
        AVG = HITS/AB
        PRINT 20, PNAME, AVG
20      FORMAT (A12, 5X, F4.3)
30      CONTINUE
C LOOP CANNOT TERMINATE WITH STATEMENT 20
C BECAUSE 'FORMAT' IS NONEXECUTABLE
```

4.2.3 CONTINUE Statement

The format of the CONTINUE statement is

```
CONTINUE
```

as shown in the preceding example.

This statement has no effect on program execution, which simply continues with the next executable statement.

4.3 Program Termination Statements

FORTRAN provides three statements for halting or terminating program execution. The PAUSE statement halts execution, but allows execution to resume. The STOP statement terminates program execution. The END statement marks the end of a program unit. It terminates a main program and acts as a RETURN from a subprogram.

4.3.1 PAUSE Statement

The format of the PAUSE statement is

```
PAUSE [msg]
```

where 'msg' is a string of not more than five digits, or is a character constant. At the time the PAUSE is executed and program execution ceases, 'msg' is displayed on the console terminal.

Program execution must be resumable following the pause. Resumption is not under program control, however, and might be initiated, for example, by an external interrupt such as a key being pressed. If execution is resumed, the normal execution sequence is continued with the statement following the PAUSE statement.

Examples:

```
PAUSE
PAUSE 1234
PAUSE 'BREAK 12'
```

4.3.2 STOP Statement

The format of the STOP statement is

```
STOP [msg]
```

where 'msg' is a string of not more than five digits, or is a character constant.

STOP terminates execution of the executable program. When this happens 'msg' is displayed on the console terminal.

Examples:

```
STOP
STOP 22
STOP 'CHECK  SUM'
```

4.3.3 END Statement

The format of the END statement is

```
END
```

The END statement marks the end of a program unit. If executed in a main program, it terminates the program. If executed in a subprogram, it has the effect of a RETURN statement and returns to the main program. *The last line of every program unit must be an END statement.*

An END statement is written only in columns 7 through 72 of an initial line and must not be continued. No other statement in a program unit can have an initial line that appears to be an END statement.

CHAPTER 5
FUNCTIONS AND SUBROUTINES

Functions and subroutines reduce coding, break programs into readily-visible logical structures, conserve storage, avoid the tedium and increased probability of error in repetitive coding, and eliminate the coding of commonly-used mathematical functions.

The term 'function' refers to a statement or subprogram that returns a value when it is referenced. A subroutine is a subprogram that does not return a value, but may alter the values of variables outside itself.

All functions and subroutines are called 'procedures.' These include:

- *Intrinsic*, or predefined, FORTRAN functions;
- Single-statement, user-defined functions (*statement* functions);
- User-defined function subprograms (*external* functions), which are identified by their initial FUNCTION statement;
- *Subroutine* subprograms identified by their initial SUBROUTINE statement.

External functions and subroutines are referred to collectively as 'external procedures.' These may be FUNCTION or SUBROUTINE subprograms defined within the program, or they may be procedures created elsewhere (e.g., PL/M and assembly language procedures) and linked to the program where appropriate.

When a procedure is defined, it usually includes 'dummy arguments' used to hold the place of 'actual arguments' to be substituted when the procedure is referenced or called. The use of dummy and actual arguments will become clearer in the remainder of this chapter.

5.1 Intrinsic and Statement Functions

5.1.1 Intrinsic Functions

FORTRAN provides a number of predefined functions for performing common operations such as square root calculation, type conversion, trigonometric calculations, etc. The complete list of available intrinsic functions can be found in Appendix B. This appendix shows the names of the various intrinsic functions, their function definitions, type of arguments, and type of results. For those intrinsic functions that have more than one argument, all arguments must be of the same type. The IMPLICIT statement has no effect on the types of intrinsic functions.

An intrinsic function is referenced by specifying it in an expression.

```
A = 33 + SQRT(B)
```

The resulting value depends on the value of the actual argument(s) used in the reference (for example, the actual value of 'B' in the expression above). The actual arguments that constitute the argument list must agree in type, number, and order with the specifications in Appendix B and may be any expression of the specified type.

Arguments for which the result is not mathematically defined or which exceed the numerical range of the processor cause results not defined in this manual. Restrictions on the range of arguments and results for intrinsic functions are listed in the notes in Appendix B.

If the name of an intrinsic function appears in the dummy argument list of a FUNCTION or SUBROUTINE subprogram, the name is considered to have no relation to the intrinsic function within the scope of the program unit and the name itself loses its intrinsic quality. The data type associated with the symbolic name is specified as normal (by default or by a type statement).

If the name of an intrinsic function is to be used as an actual argument in an external procedure reference, the name must first be specified in an INTRINSIC statement.

5.1.2 INTRINSIC Statement

The INTRINSIC statement confirms that a symbolic name represents an intrinsic function and allows that name to be used as an actual argument. Only one appearance of a symbolic name in all of the INTRINSIC statements in a given program unit is allowed. A symbolic name may not appear in an INTRINSIC statement and an EXTERNAL statement in the same program unit.

The format of the INTRINSIC statement is

INTRINSIC *func* [,*func*]...

where 'func' is an intrinsic function name.

The names of certain intrinsic functions cannot be used as actual arguments and, therefore, cannot appear in INTRINSIC statements. These are the functions for type conversion (INT, IFIX, FLOAT, REAL, ICHAR) and the functions for choosing a largest or smallest value (MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, AMIN0, MIN1).

Examples:

```
INTRINSIC SQRT
INTRINSIC EXP, LOG, LOG10
```

5.1.3 Statement Functions

Straightforward mathematical functions like

$f(x) = ax^2 + bx + c$

are defined in FORTRAN using statement functions. These functions have no keyword; the format is

func ([*dum* [,*dum*] ...]) = *exp*

where

func is the symbolic name of the statement function

dum is a dummy argument to be replaced by an actual argument when the function is referenced

exp is an expression

As an example, the mathematical function above would be written

```
F(X) = A*(X**2)+ B*X + C
```

Substituting the actual argument '3' for the dummy argument 'X' in this function would result in the value '9A + 3B + C.'

The statement function name and the expression 'exp' may be of different types. The type of the value returned is as shown in Figure 3-2.

The dummy argument list indicates the order, number and type of arguments for the statement function. The names of dummy arguments have a scope of the statement function only, and each name may appear only once in the dummy argument list. The type of a dummy argument name is the same as it would be if the name were used outside the statement function.

The name of a dummy argument can be used to identify other dummy arguments of the same type in other statement-function statements. The name can also be used to identify a variable of the same type or a common block within the same program unit, but they have no other relationship.

5.1.3.1 Referencing Statement Functions

A statement function is referenced by specifying its symbolic name (with any required actual arguments).

```
DATA  A, B, C /10.0, 10.0, 3.8/
FSUM(X) = A*(X**2) + B*X + C
TOTAL = 33 + FSUM(3.0)
```

This operation substitutes the value '3.0' for every occurrence of 'X' in the function definition. At the end of the operation, the value of 'TOTAL' is '156.8.'

5.1.3.2 Statement Function Limitations

All statement functions must follow all specification statements and must precede all executable statements.

The symbolic name of a statement function is local and cannot be a symbolic name in any specification statement, except in a type statement or as the name of a common block in the same program unit. The name is also prohibited from being used in an EXTERNAL statement or as an actual argument.

A statement function cannot be of type character. If it has a dummy argument of type character, the length specification of that argument must be an integer constant.

Each operand in the statement function expression 'exp' must be one of the following:

- A constant:
  ```
  FPROD(C) = C*3.1412
  ```
- A variable reference:
  ```
  FPROD(C) = C*3.1412 + AV12
  ```
- An array element reference:
  ```
  FSUM(C) = C + 2/TABLE(1,3)
  ```
- An intrinsic function reference:
  ```
  FHYP(A,B) = SQRT(A**2 + B**2)
  ```
- A reference to a statement function:
  ```
  FTOTAL(C) = C/3 + FUNC(3.8)
  ```
- An external function reference:
  ```
  FSUB(C) = 3*C – EXFUN(3.0,3.0,2.5)
  ```
- A dummy procedure reference:
  ```
  FSUB(C) = 3*C – EXFUN(X,Y,Z)
  ```

A statement function may be referenced only in the program unit where it is defined. The statement function may not reference another statement function if that other function is defined later in the program unit. Furthermore, a statement function in a FUNCTION subprogram must not reference the name of the subprogram. A reference to an external function in the expression of a statement function must not cause a dummy argument of the statement function to become undefined or redefined.

5.2 External Procedures

This section describes external procedures, specifically procedures defined by FUNCTION and SUBROUTINE subprograms. It also describes statements related to external procedures: RETURN, SAVE, EXTERNAL, and the subroutine CALL.

5.2.1 FUNCTION Statement

The FUNCTION statement introduces a FUNCTION subprogram. The FUNCTION statement must always be the first statement of the subprogram and the subprogram must be terminated with an END statement.

The format of the FUNCTION statement is

[*typ*] FUNCTION *func* ([*dum* [,*dum*]...])

where

typ is either INTEGER [**len*], REAL, or LOGICAL [**len*], *len* being 1, 2, or 4.

func is the symbolic name of the subprogram and is an external function name.

dum is a dummy argument and is either a variable, array, or dummy procedure name.

The FUNCTION subprogram name, appearing as a variable within the subprogram, must become defined or redefined each time the subprogram is executed. The value of this variable when a RETURN or END is executed is the value of the function. An external function in a subprogram may use dummy arguments to return values in addition to the function value returned.

If 'typ' is not specified, the type and length of 'func' are determined by the default conventions described in sections 2.2.2.1 and 2.2.2.2.

5.2.1.1 Referencing External Functions.

An external function is referenced by specifying its name in an expression (along with all necessary actual arguments).

The actual arguments in the reference must agree in order, number, type, and length with their corresponding dummy arguments. One exception is the use of a subroutine name as an actual argument; subroutine names do not have an associated type. Actual arguments must also be one of the following:

- An expression
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name

An actual argument may also be a dummy argument as long as the dummy is part of a dummy argument list within the subprogram containing the external function reference.

5.2.1.2 FUNCTION Subprogram Limitations

A FUNCTION statement may be used only as the first statement of a FUNCTION subprogram. The subprogram itself can consist of any other statement except a SUBROUTINE, BLOCK DATA, or PROGRAM statement.

The name of the FUNCTION subprogram is global and cannot be the same as any other name in the subprogram, except for its use as a variable in the body of the subprogram. Within the subprogram, the only nonexecutable statement in which the name may appear is a type statement, and even this is not permitted if the type is specified in the FUNCTION statement. A FUNCTION subprogram name may not be type character.

The symbolic name of a dummy argument in a function subprogram is local to the program unit and cannot be used in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement (except as a common block name).

A function specified by a subprogram can be referenced within any other external procedure or in the main program. A function subprogram must not reference itself, however, either directly or indirectly (but see the description of the REENTRANT compiler control in section F.2.3).

When an external function reference is executed, the function must be part of the program. External functions created outside the program must be linked to the program before it is executed. External procedure linkage is described in the FORTRAN compiler operator's manual.

Example:

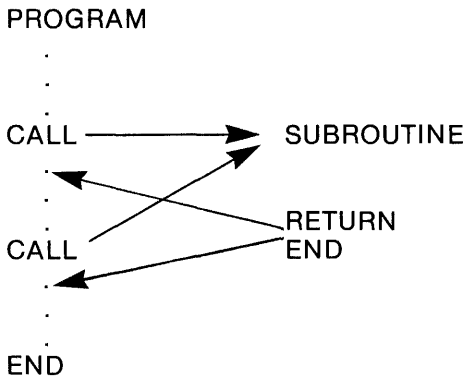
```
C  THE FOLLOWING EXAMPLE TOTALS THE
C  VALUES IN AN ARRAY OF LENGTH I

      FUNCTION TOTAL(ARRAY, I)
            DIMENSION ARRAY(I)
            TOTAL = 0.0
            DO 100 K = 1, I
                TOTAL = TOTAL + ARRAY(K)
100         CONTINUE
            RETURN
      END
```

5.2.2 Subroutines

A subroutine is used primarily to avoid redundant blocks of code. A subroutine can be called from anywhere in a program, with actual arguments substituted for the dummy arguments specified in the subroutine definition. The subroutine performs its operations, then returns control to the point of call.

The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram. The subprogram (being a program unit) must be terminated by an END statement and can contain any statement except another SUBROUTINE statement, or a FUNCTION, BLOCK DATA, or PROGRAM statement. At some point in the subprogram, the RETURN statement can be specified to tell the subroutine when to return control to the point of call.

The subroutine is called by the CALL statement.



Subroutines, being external procedures, can be defined outside the program. By the time a program containing a CALL to the subroutine is executed, however, the subroutine must be part of the calling program, either by SUBROUTINE subprogram definition or by being linked to the program. External procedure linkage is described in the FORTRAN compiler operator's manual.

5.2.3 SUBROUTINE Statement

The SUBROUTINE statement is used only as the first statement of a SUBROUTINE subprogram. Its format is

SUBROUTINE *sub*[([*dum*[,*dum*]...])]

where

sub is the symbolic name of the subroutine.

dum is a dummy argument and is either a variable, array, or dummy procedure name.

If no dummy arguments are specified, either of the forms 'SUBROUTINE *sub*' or 'SUBROUTINE *sub*()' is acceptable.

The symbolic name of the subroutine is a global name and must not be the same as any other name in the program unit.

The symbolic name of a dummy argument is local to the program unit and cannot be used in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

An example is shown following the RETURN statement.

5.2.4 RETURN Statement

The RETURN statement returns control to the calling program unit. It may appear only in a FUNCTION or SUBROUTINE subprogram. These subprograms may have one or more RETURN statements, or they may have none at all. The END statement terminating such a program unit has the same effect as a RETURN.

The format of the RETURN statement is simply

RETURN

When a RETURN is executed in a FUNCTION subprogram, the value of the function must be available to the referencing program unit.

Whenever RETURN is executed, the association between the dummy arguments of an external procedure and the current actual arguments is terminated (but see section 5.2.5, the SAVE statement).

Example:

```
C THREE NUMBERS ARE ADDED AND THE FLAG 'POSTOT' SET TO
C TO 1 IF THEIR TOTAL IS POSITIVE
          SUBROUTINE POSTOT(A,B,C)
               IF ((A + B + C) .GE. 0) THEN
                POSTOT = 1
                ELSE
                POSTOT = 0
                END IF
          RETURN
          END
```

5.2.5 SAVE Statement

The SAVE statement can be used to make sure that common variables within a FUNCTION or SUBROUTINE subprogram do not become undefined when a RETURN or END is executed.

The format of the SAVE statement is

SAVE */cb/* [,*/cb/*]...

where 'cb' is a named common block. Naming the common block saves all items in that block. A specific common block cannot be listed more than once in a single SAVE statement. Entities in blank common never become undefined as the result of executing RETURN or END.

Example:

```
SAVE /BLOCK1/, /BLOCK2/
```

5.2.6 EXTERNAL Statement

The EXTERNAL statement confirms that a symbolic name represents an external or dummy procedure and allows that name to be used as an actual argument.

The format of the EXTERNAL statement is

EXTERNAL *proc* [,*proc*] ...

where 'proc' is the name of an external or dummy procedure.

If an intrinsic function name is specified in an EXTERNAL statement, that name can no longer be used to specify an intrinsic function in the program unit, but instead becomes the name of an external procedure.

A symbolic name can be specified only once in all the EXTERNAL statements in a program unit.

Example:

```
EXTERNAL HYP, POSTOT, SIN
```

5.2.7 CALL Statement

The CALL statement is used to reference a subroutine. A subroutine can be referenced within any other external procedure or in the main program. A subprogram must not reference itself either directly or indirectly (but see the description of the REENTRANT compiler control in section F.2.3).

The format of the CALL statement is

CALL *sub* [([*arg*[,*arg*]...])]

where

sub is the symbolic name of a subroutine or dummy procedure.

arg is an actual argument.

The actual arguments in the CALL statement must agree in order, number, type, and (if applicable) length with the corresponding dummy argument list of the referenced subroutine. If the name of a subroutine is specified as an actual argument, the type conformity rule does not apply since subroutines do not have an associated type.

Each actual argument must be one of the following:

- An expression
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name

An actual argument in a CALL statement may be a dummy argument name if that name appears in a dummy argument list within the subprogram containing the CALL.

Examples:

```
C THE  FOLLOWING STATEMENTS COULD BE USED TO CALL
C THE SUBROUTINE DEFINED IN THE 'RETURN STATEMENT'
C EXAMPLE (SECTION 5.2.5)

      CALL  POSTOT(3.2,—2.7, 0.08)
      CALL  POSTOT(X, 5.2**I,—Y)
```

5.3 Arguments And Common Blocks Revisited

Arguments and common blocks are the means of communicating between procedures and statements referencing the procedures. Data can be passed to a statement function or intrinsic function by an argument list. Data can be passed between external procedures and other program units by argument lists or can be shared in common blocks.

FORTRAN has a number of very specific rules governing the use of common blocks and argument lists. We touched on the most important when function and subroutine references were discussed. This section repeats those rules plus a number of others related to arguments and common blocks.

5.3.1 Common Blocks

Common blocks reduce storage requirements by allowing two or more subprograms to share the same memory. This sharing may be limited by the rules for defining and referencing data.

The variables and arrays in a common block can be defined and referenced in all subprograms that contain a declaration of that common block. Association is by memory location rather than by name, so the names of variables and arrays of a given common block may be different in the different subprograms. However, the data referenced and the common block name used to reference the data must be of the same type.

An integer variable ASSIGNed a statement label can only be referenced in the program unit containing its ASSIGN statement.

5.3.2 Dummy And Actual Arguments

A dummy argument is used in the argument list when defining a procedure. An actual argument is used in the corresponding argument list when the procedure is referenced.

Dummy arguments are used by statement functions, FUNCTION subprograms, and SUBROUTINE subprograms to specify the types of actual arguments and whether the argument is a single value, array, or procedure. In the case of a statement function, the dummy argument must be a variable.

Dummy argument names can be used anywhere an actual name of the same 'class' (that is, variable, array, or dummy procedure) and type can be used, unless explicitly prohibited. Dummy argument names cannot appear in EQUIVALENCE, DATA, SAVE, INTRINSIC, or COMMON statements (except as common block names). A dummy name also cannot be the same as a FUNCTION, SUBROUTINE, or statement function name declared in the same program unit.

Actual arguments list the entities to be associated with corresponding dummy arguments for a particular procedure reference. Actual arguments may be constants, function or subroutine references, or expressions, including parenthesized expressions (if the associated dummy argument is not defined during execution of the external procedure). An actual argument cannot be the name of a statement function defined later in the program unit containing the reference.

5.3.3 Association Of Arguments

When a procedure reference is executed, the actual and dummy arguments are associated; the first actual argument replaces all occurrences of the first dummy argument, etc. Therefore, the arguments must agree in order, number, type (except when the actual argument is a subroutine name), and length (where applicable).

If an actual argument is associated with a dummy appearing in an adjustable dimension, the actual argument must be defined with an integer value when the procedure is referenced.

If an actual argument is an expression, the expression is evaluated before association. If the actual is an array element name, its subscript is evaluated before association.

NOTE

The subscript value remains constant as long as the arguments are associated, even if the subscript contains variables redefined during the association.

Argument association can be carried through more than one level of procedure reference. A valid association exists at the last level, however, only if the association is maintained through all intermediate levels. The association normally terminates when a RETURN or END is executed. The association is not retained from one procedure reference to the next.

5.3.3.1 Agreement Of Argument Lengths

If a dummy argument is type character, its actual argument must be the same type and length. If the dummy is an array name, the length requirement applies to each element in the array.

If the dummy argument is type integer, the arguments must again agree in type and length. If an integer constant is used as an actual argument, its length is determined as shown in section 2.2.2.2. That is, an integer without a base suffix has the default integer variable length; if the base of the integer constant is stated explicitly, the processor determines its length implicitly.

5.3.3.2 Variables As Dummy Arguments

A variable dummy argument can be associated with an actual variable, array element, or expression. If the actual argument is a variable name or array element name, its dummy argument can be defined or redefined within the subprogram. Otherwise, if the actual argument is a constant, function reference, or expression, its dummy argument must not be redefined within the subprogram.

5.3.3.3 Arrays As Dummy Arguments

An array dummy argument can be associated with an actual array name or array element name. The number and size of dimensions in an actual argument array declarator can be different from the dimensions in its associated dummy array declarator.

If the actual argument is an *array name*, the size of the dummy argument array must not be greater than the size of the actual argument array.

If the actual argument is an *array element name* with a subscript 'p,' the dummy array element with a subscript 'q' becomes associated with the actual array element with the subscript value (p + q-1). The size of the dummy array must not be greater than the size of the actual array plus one minus the subscript value of the array element.

5.3.3.4 Procedures As Dummy Arguments

If a dummy argument is used as if it were an *external function*, the associated actual argument must be an intrinsic function, external function, or dummy procedure. If a dummy argument appears in a *type statement and an EXTERNAL statement*, the actual argument must be the name of a function or dummy procedure. If the dummy argument is referenced as a *subroutine*, the actual argument must be the name of a subroutine or dummy procedure and must not appear in a type statement or be referenced as a function.

NOTE

In a given program unit, determining whether a dummy procedure is associated with a function or a subroutine may not be possible. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, examining the subprogram is not enough to determine whether the symbolic name should be associated with a subroutine or a function.

A dummy argument associated with an intrinsic function has no automatic type association. Therefore, the type of the dummy argument must agree with the type of the result of all actual arguments associated with it. An intrinsic function name used as a dummy external function name loses its property as an intrinsic function within the subprogram. A dummy argument associated with an intrinsic function and used as a procedure name in a function reference must have arguments that agree in number and type with those specified for the intrinsic function.

5.3.3.5 Argument Association Limitations

If dummy arguments in the same subprogram are associated as the result of a reference to that subprogram, neither dummy can be redefined during execution of the subprogram. For example, an external function beginning with

```
FUNCTION F(A,B,C)
```

could not be referenced by

```
F(X, Y, X)
```

because 'A' and 'C' would become associated.

If a subprogram reference causes a dummy argument to become associated with an item in a common block in the same subprogram, neither the dummy nor the common item can be redefined within the subprogram.

CHAPTER 6
INPUT/OUTPUT

The FORTRAN input/output (I/O) statements transfer data between a processor and external units or within the processor itself. These statements can specify the external units to be used, the variables whose values are being entered or output, and the format of I/O data.

The first group of I/O statements are the file-handling statements (OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE). As their names imply, these statements are used for connecting and disconnecting files, positioning files, and marking the end of a file.

The external unit to be used and the list of variables to be input or output is supplied by the READ, WRITE, and PRINT data-transfer statements. When the input or output is formatted, these statements are used with the FORMAT statement.

Port I/O for a particular device is provided by the corresponding Intel FORTRAN compiler. See the discussion of the INPUT and OUTPUT intrinsic subroutines in section F.2.2.

Before any of these statements are discussed, the properties of records, files, and units are reviewed. They can be described only in general here, however, since many of the specifics related to them are dependent on the processor or operating environment.

6.1 Records, Files, And Units

6.1.1 Record Properties

A record is simply a sequence of values or characters. The length of a record is generally the same as the sum (in bytes) of the items written into the record, unless it is stated specifically in a record length specifier (section 6.2.1.8). Records are classified as formatted or unformatted.

A formatted record can be any sequence of *characters* representable in the processor (except escape, carriage return, or line feed characters). An unformatted record can be a sequence of values containing both *character and noncharacter* data. These two kinds of records can only be read or written by formatted and unformatted I/O statements, respectively.

6.1.2 File Properties

The main properties of a file are:

- That is may exist;
- That it may have a name;
- That it has a position;
- That it may be external or internal to the processor;
- That it may be accessed sequentially or directly;
- That it may have records of specific length (for direct access files);
- That it may have formatted or unformatted contents.

6.1.2.1 File Existence

At the time an executable program is running, a certain set of files is available. These files are said to *exist*, and the particular files that exist are determined by the operating system or environment in which the program is running.

A file may exist without containing any data; an example would be a newly-created file having only a name.

All FORTRAN I/O statements can refer to existing files. Some statements (OPEN, CLOSE, WRITE, PRINT, and ENDFILE) can also refer to files that do not yet exist and are in the process of being created.

6.1.2.2 File Name

FORTRAN has no standard file-naming convention. Such conventions are system dependent. For example, in the ISIS-II environment all of the following are valid file names:

MYFILE	1-6 character file name
MYFILE.SRC	file name plus 1-3 character extension
:LP:	device name, in this case line printer
:F1:PROG.OBJ	device name plus file name plus extension

6.1.2.3 File Position

Once a file has been connected to a unit, it has a position. The 'initial point' of a file is the position preceding the first record. Its 'terminal point' is the position just after the last record. If the file is positioned within a record, that record is the 'current record.'

Executing certain I/O statements affects the position of the file. Some circumstances can cause the file position to become indeterminate.

6.1.2.4 External And Internal Files

An external file is a file that can be connected to an external unit. An internal file is a character variable, character array, or character array element. Internal files allow you to transfer data within processor memory.

An internal file cannot be specified by one of the file-handling statements (OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE). It can only be read or written by sequential- access, formatted I/O statements that do not specify 'list-directed formatting.'

An internal file has the following properties:

- Each record is a character array element.
- The length of the file depends on its kind. If the file is a character variable or array element, it is a single record whose length is the length of the variable or array element. If the file is a character array, every record has the same length as an array element in the array and the file has as many records as array elements.
- A record may be read only if it has been defined. A variable or array element record is defined by writing the record (or by making it the target in an assignment statement). If the number of characters written is less than the length of the record, the characters are left-adjusted in the record and the remainder of the record is filled with blanks.
- An internal file is always positioned at its initial point before a data transfer.

6.1.2.5 File Access

An external file can be accessed *sequentially* or *directly*. An internal file can only be accessed sequentially.

Some external files may be allowed more than one access method, depending on the operating environment. The access method is determined when the file is connected to a unit.

6.1.2.5.1 Sequential Access File. A file connected for sequential access has the following characteristics:

- The records are a totally ordered set, having the order in which they were written.
- The records are either all formatted or all unformatted.

If the file may also be connected for direct access, the order of a direct-access read is the same as the order of the sequential write.

6.1.2.5.2 Direct Access Files. A file connected for direct access has the following characteristics:

- The order of its records is the order of their record numbers. Its records can be read or written in any order.
- Its records are either all formatted or all unformatted.
- Its records can be read or written only by direct-access I/O statements. List-directed formatting cannot be used.
- All records of the file are the same length.
- Each record of the file has a unique record number, specified when the record is written. A record cannot be deleted or its number changed, but is can be rewritten.

6.1.3 Unit Properties

A 'unit' is a logical way of referring to a file. Like files, units can exist or not for an executable program. All FORTRAN I/O statements can refer to existing units. The CLOSE statement can also refer to nonexistent units.

A unit has the property of being *connected* or *disconnected*. If connected, it refers to a file. All I/O statements except OPEN and CLOSE must reference a unit connected to a file.

Typically, a file is connected by the OPEN statement and disconnected by the CLOSE statement. Depending on the operating environment, some units may also be *preconnected*, meaning they can be referenced by I/O statements without first having to be connected by the OPEN statement. A preconnected file becomes connected the first time it is referenced by an I/O statement. For example, in the ISIS-II environment the console output device and console input device are always preconnected. See the discussion of the UNIT run-time control in section F.3.

A unit must not be connected to more than one file at a time, and vice versa. A file may, of course, be disconnected by the CLOSE statement and then reconnected to the same or a different unit by the OPEN statement.

NOTE

The only way to refer to a disconnected file is by naming it in an OPEN statement. Consequently, an unnamed file may not be able to be reconnected once it has been disconnected.

6.2 File-Handling Statements

6.2.1 OPEN Statement

The OPEN statement can be used to connect an existing file to a unit, create a preconnected file, create a file and connect it to a unit, or change certain specifiers in the file/unit connection.

The format of the OPEN statement is

OPEN (*open-list*)

where 'open-list' is a list of specifiers separated by commas. The list of specifiers is:

[UNIT =] *unit*	Unit specifier
IOSTAT = *stname*	I/O status specifier
ERR = *stl*	Error specifier
FILE = *fname*	File name specifier
STATUS = *stat*	File status specifier
ACCESS = *acc*	Access method specifier
FORM = *fmat*	Formatting specifier
RECL = *reclen*	Record length specifier
BLANK = *blnk*	Blank specifier
CARRIAGE = *car*	Carriage control specifier

The unit specifier 'unit' must be present and the unit specified must exist. All other specifiers are optional except that the record length (RECL) must be specified if the file is being connected for direct access. Some specifiers have default values.

The following sections 6.2.1.1 through 6.2.1.10 describe each of the 'open-list' specifiers in detail.

6.2.1.1 Unit Specifier

The format of the unit specifier is

[UNIT =] *unit*

where 'unit' identifies an external unit. If the optional 'UNIT =' is omitted, the unit specifier must be the first item in 'open-list.'

An external unit identifier must be an integer expression whose value is in the range $0 \leq \text{unit} \leq 255$.

Examples:

```
OPEN (UNIT=3)
OPEN (4)
```

6.2.1.2 I/O Status Specifier

The format of the input/output status specifier is

IOSTAT = *stname*

where 'stname' is an integer variable or integer array element name.

Executing an OPEN statement containing this specifier causes 'stname' to become defined with a zero value if no error condition exists, or with a processor-dependent positive integer value if an error condition does exist.

Example:

```
OPEN (4, IOSTAT = ERRFLG)
```

6.2.1.3 Error Specifier

The format of the error specifier is

ERR = *stl*

where 'stl' is the label of an executable statement in the same program unit as the OPEN statement.

If the processor discovers an error condition while executing the OPEN statement, the following steps occur:

1. The OPEN operation terminates;
2. The position of the file specified by OPEN becomes indeterminate;
3. If the OPEN statement has an IOSTAT specifier, 'stname' is set to reflect the error condition;
4. Execution continues with the statement named by the ERR specifier.

Example:

```
OPEN  (4, IOSTAT=ERRFLG,ERR=1010)
```

6.2.1.4 File Name Specifier

The format of the file name specifier is

FILE = *fname*

where 'fname' is the name of the file to be connected, expressed as a character-type constant or variable. The file name must be valid for the particular system in which the program is executing. If FILE is omitted, the unit becomes connected to a processor-determined file.

Example:

```
OPEN(UNIT=3, FILE='MYPROG.SRC')
```

6.2.1.5 File Status Specifier

The format of the file status specifier is

STATUS = *stat*

where 'stat' is a character expression evaluating to 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. If the STATUS specifier is omitted, the default value is 'UNKNOWN'.

If 'OLD' or 'NEW' is specified, the FILE specifier must be present also. An 'OLD' file must exist already; a 'NEW' file cannot exist already.

The 'SCRATCH' option must not be specified with a named file. When it is specified with an unnamed file, the file is connected to the specified unit for the duration of program execution or until a CLOSE statement is issued for the same unit.

If 'UNKNOWN' is specified, the file status is processor dependent.

Example:

```
OPEN (3, FILE='MYPROG.SRC', STATUS='NEW')
```

6.2.1.6 Access Method Specifier

The format of the access method specifier is

```
ACCESS = acc
```

where 'acc' is a character expression evaulating to 'SEQUENTIAL' or 'DIRECT' (see section 6.1.2.5). If the ACCESS specifier is omitted, the default is 'SEQUENTIAL'.

If the file already exists, the specified access method must be allowable for that file. For a new file, the processor creates the file with the specified access method. If the access method is 'DIRECT', the record length specifier (6.2.1.8) must also be present in 'open-list.'

Example:

```
    OPEN (3, FILE='MYPROG', STATUS='NEW',
   & ACCESS='SEQUENTIAL')
```

6.2.1.7 Formatting Specifier

The formatting specifier states whether a file is being connected for formatted or unformatted input/output. Its format is

```
FORM = fmat
```

where 'fmat' is a character expression evaluating to 'FORMATTED' or 'UNFORMATTED'. If the FORM specifier is omitted, the default value is 'UNFORMATTED' when the file is being connected for direct access, and 'FORMATTED' when the file is being connected for sequential access.

If the file already exists, the specified formatting must be legal for that file. For a new file, the processor creates the file with the specified formatting.

Example:

```
    OPEN (3, FILE='MYPROG', STATUS='NEW',
   & ACCESS='SEQUENTIAL', FORM='FORMATTED')
```

6.2.1.8 Record Length Specifier

The record length specifier identifies the length of each record in a file being connected for direct access. Its format is

```
RECL = reclen
```

where 'reclen' is a positive integer expression.

If the file is being connected for formatted I/O, 'reclen' is the number of characters. If the file is being connected for unformatted I/O, the length is measured in bytes.

If the file already exists, the length specified must be that used when the file was created. In the case of a new file, the processor creates a file with the specified length for each record.

The RECL specifier must be included in the OPEN statement when a file is being connected for direct access.

Example:

```
    OPEN (3, FILE='CARDS', STATUS='NEW', ACCESS='DIRECT',
   & FORM='FORMATTED', RECL=80)
```

6.2.1.9 Blank Specifier

The format of the blank specifier is

```
BLANK = blnk
```

where 'blnk' is one of the character constants 'NULL' or 'ZERO'. If the BLANK specifier is omitted, the default value is 'NULL'.

If 'NULL' is specified, all blanks in numeric formatted input fields are ignored, except that a field of all blanks has the value zero. If 'ZERO' is specified, all blanks except leading blanks have the value zero.

The specifier is permitted only for files being connected for formatted input/output.

Example:

```
    OPEN (UNIT=3, FILE='TOTALS', STATUS='NEW',
   & FORM='FORMATTED', BLANK='ZERO')
```

6.2.1.10 Carriage Control Specifier

The carriage control specifier states whether the first character of each record of formatted output is to be used to determine vertical spacing (section 6.4.2.1) or not. Its format is

```
CARRIAGE = car
```

where 'car' is a character expression evaluating to 'FORTRAN' or 'NULL'. If the CARRIAGE specifier is omitted, the default value is 'NULL'.

If 'FORTRAN' is specified, the first character of formatted output may be interpreted as a vertical spacing identifier for a printer. If 'NULL' is specified, the first character is simply output as the first character of the new record. Note that a file written with CARRIAGE = 'FORTRAN' cannot be read in later by a FORTRAN program.

Example:

```
    OPEN (UNIT=6, FILE='MYPROG.OBJ', STATUS='OLD',
   & FORM='FORMATTED', CARRIAGE='NULL')
```

6.2.1.11 Opening A Connected Unit

The OPEN statement can be specified for a unit already connected to an existing file. That existing file is assumed to be the value of 'fname' if the FILE specifier is not included in the OPEN 'open-list.'

If the file to be connected is the same as the connected file, the effect of OPEN depends on whether or not the file was preconnected. If the file was not preconnected, or if it was preconnected and has already been referenced by an I/O statement, only the BLANK specifier (and RECL for sequential files) may differ from existing attributes. If the file was preconnected, but no I/O has been performed on the file, the properties specified in the OPEN statement become the properties of connection. Subsequent OPEN statements can change only the BLANKS and RECL attributes as described above.

If the file to be connected is *not* the same as the file already connected, the currently connected file is closed and the new file is opened with this unit. The connected file is deleted if it was previously opened with status 'SCRATCH', but is not deleted if its status is 'OLD', 'NEW', or 'UNKNOWN'.

If a file is already connected to a unit, no OPEN statement connecting that file to a different unit can be executed.

6.2.2 CLOSE Statement

The CLOSE statement is used to disconnect a particular file from a unit. Its format is

CLOSE *(close-list)*

where 'close-list' is a list of specifiers separated by commas. The list of specifiers is:

[UNIT =] *unit*	Unit specifier
IOSTAT = *stname*	I/O status specifier
ERR = *stl*	Error specifier
STATUS = *stat*	File disposition specifier

The unit specifier must be present; all other specifiers are optional and can be specified only once.

The IOSTAT and ERR specifiers have the same interpretation as in sections 6.2.1.2 and 6.2.1.3, respectively. The UNIT and STATUS specifiers are described in the following sections.

6.2.2.1 Unit Specifier

The unit specifier has the same interpretation as in section 6.2.1.1. Execution of the CLOSE statement containing this specifier need not occur in the same program unit as its corresponding OPEN statement, however. If the specified file does not exist, CLOSE has no effect.

Once a unit has been disconnected by the CLOSE statement, it may be reconnected to the same file or a different file within the same program. Similarly, once a file has been disconnected, it may be reconnected to the same or a different unit, so long as the file still exists.

Example:

```
CLOSE (3, IOSTAT = ERRFLG, ERR = 1020)
```

6.2.2.2 File Disposition Specifier

The format of the file disposition specifier is

STATUS = *stat*

where 'stat' is a character expression evaluating to 'KEEP' or 'DELETE'. If this specifier is omitted, the default value is 'DELETE' for a file that previously had a status of 'SCRATCH', and 'KEEP' otherwise. Under no circumstances can 'KEEP' be specified for a file opened with 'SCRATCH' status.

If 'KEEP' is specified for an existing file, the file continues to exist after the CLOSE statement is executed. Otherwise, 'KEEP' has no effect.

If 'DELETE' is specified, the file ceases to exist after the CLOSE statement is executed.

Following normal program termination, all connected units are closed. Scratch units are deleted; all others are closed with disposition 'KEEP'.

Example:

```
CLOSE (3, ERR = 1020, STATUS = 'KEEP')
```

6.2.3 BACKSPACE Statement

The BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. Its possible formats are

BACKSPACE *unit*
BACKSPACE *(arg-list)*

where 'unit' is an external unit specifier and 'arg-list' is a list of arguments separated by commas. The list of arguments is:

[UNIT =] *unit*	External unit specifier
IOSTAT = *stname*	I/O status specifier
ERR = *stl*	Error specifier

The argument list must include an external unit specifier (section 6.2.1.1) and may contain an I/O status specifier (section 6.2.1.2) and an error specifier (section 6.2.1.3). The file being backspaced must be connected for sequential access.

If the file has no preceding record, the BACKSPACE statement has no effect. If the end-of-file condition has occurred, the file is positioned such that the last record of the file becomes the preceding record.

Backspacing over a record written using list-directed formatting is not allowed. Backspacing a nonexistent file or unit is prohibited also.

Examples:

```
BACKSPACE 3
BACKSPACE (3, ERR = 1020)
```

6.2.4 REWIND Statement

The REWIND statement causes the file connected to the specified unit to be repositioned at its initial point. The file must be connected for sequential access. The possible formats for the REWIND statement are

REWIND *unit*
REWIND*(arg-list)*

where 'unit' is an external unit specifier and 'arg-list' is a list of arguments as described for the BACKSPACE statement (section 6.2.3).

If the specified file is already positioned at its initial point, or if the file is connected but does not exist, the REWIND statement has no effect. If an end-of-file condition has occurred, the file can still be rewound.

Examples:

```
REWIND 3
REWIND (3, IOSTAT = ERRFLG, ERR = 1030)
```

6.2.5 ENDFILE Statement

When the ENDFILE statement is executed, the record preceding the ENDFILE becomes the last record of the file. No data-transfer I/O statement can be executed on the file without first issuing a BACKSPACE or REWIND statement. The file must be connected for sequential access when ENDFILE is issued.

The possible formats of the ENDFILE statment are

```
ENDFILE unit
ENDFILE (arg-list)
```

where 'unit' is an external unit specifier and 'arg-list' is a list of arguments as described for the BACKSPACE statement (6.2.3).

If the file can be connected for direct access also, only those records appearing before the end-of-file record can be read during subsequent direct access operations.

If a file is preconnected to a unit but does not yet exist, specifying the unit in an ENDFILE statement causes the file to be created.

Examples:

```
ENDFILE 3
ENDFILE (3, ERR = 1040)
```

6.3 Data-Transfer I/O Statements

Once a file has been connected to a unit, data in the file can be read using the READ statement, or data can be written into the file using the WRITE or PRINT statements. Note that the keyword 'PRINT' does not imply that an output file is connected to a line printer, nor does the keyword 'WRITE' imply that it is not.

6.3.1 READ Statement

The READ statement reads data from a specified unit. Its possible formats are

```
READ (ctl-list) [in-list]
READ f [,in-list]
```

where

ctl-list	is a list of control information specifiers
in-list	is a list of the data to be read
f	is a format identifier, and is the same as the 'FMT = *f*' specifier in 'ctl-list.'

The list of control information specifiers is:

[UNIT =] *unit*	Unit specifier
[FMT =] *f*	Format specifier
REC = recno	Record number specifier
IOSTAT = *stname*	I/O status specifier
ERR = *stl*	Error specifier
END = *stl*	End-of-file specifier

6.3.1.1 Control Information List

The control information list must contain a unit specifier. If the second form of the READ statement shown above is used (that is, if no unit is specified), the unit read is the default unit.

The 'ctl-list' may contain, at most, one of each of the other specifiers.

The following sections 6.3.1.1.1 through 6.3.1.1.6 describe control list specifiers in detail.

6.3.1.1.1 Unit Specifier. The unit specifier has the same interpretation as for the OPEN statement (section 6.2.1.1). In addition, for data-transfer I/O statements, 'unit' may be an asterisk. If this is the case, the specifier identifies a particular processor-determined external unit.

The unit specifier may also point to an internal file (that is, 'unit' may be the name of a character variable, character array, or character array element name). In this case, 'ctl-list' must contain a format identifier (other than asterisk) and may not contain a record number specifier.

If the optional 'UNIT =' is omitted, the unit specifier must be the first item in 'ctl-list.'

Example:

```
READ (2) PNAME, AB, HITS
```

6.3.1.1.2 Format Specifier. If 'ctl-list' contains a format specifier, the READ statement is a *formatted* I/O statement; otherwise, it is an *unformatted* I/O statement.

The format of this specifier is

[FMT =]*f*

where 'f' is one of the following:

- The label of a FORMAT statement in the same program unit as the READ;
- An integer variable ASSIGNed the label of a FORMAT statement;
- A character array name, character variable name, or character expression containing a format specification;
- An asterisk (*) specifying list-directed formatting (section 6.4.4);
- An integer, real, or logical array containing a format specification as Hollerith data.

If the optional 'FMT =' is omitted, the format specifier must be the second item in 'ctl-list' and the first item must be the unit specifier without the optional characters 'UNIT =.'

If the asterisk option is selected, 'ctl-list' must not include a record number specifier. If the unit specifier is an internal file, the format specifier must be present, but cannot be an asterisk.

Examples:

```
      READ (2,25) PNAME, AB, HITS
25    FORMAT . . .

      READ 25, PNAME, AB, HITS
25    FORMAT . . .

      ASSIGN 25 TO INFMT
      READ (2,INFMT) PNAME
25    FORMAT . . .

      READ (2,*) PNAME
```

6.3.1.1.3 Record Number Specifier. The record number specifier is included in 'ctl-list' if and only if the file to be read is connected for direct access. It has the format

REC = *recno*

where 'recno' is an integer expression whose value is positive. The value of this expression is the number of the record to be read.

Examples:

```
READ (2, REC = 20)
READ (2, REC = K)
READ (2, REC = K + 1)
```

6.3.1.1.4 Input/Output Status Specifier. The I/O status specifier, IOSTAT, is essentially interpreted as it was for the OPEN statement (section 6.2.1.2). In the case of data-transfer I/O statements, however, the variable 'stname' is also assigned a negative value at end-of-file.

6.3.1.1.5 Error Specifier. The error specifier has the same interpretation as for the OPEN statement (section 6.2.1.3).

6.3.1.1.6 End-Of-File Specifier. The format of the end-of-file specifier is

END = *stl*

where 'stl' is the label of an executable statement in the same program unit as the statement containing this specifier.

When the end-of-file is detected during a read operation, execution of the READ statement terminates, 'stname' is assigned a negative value (6.3.1.1.4), and execution continues with the statement specified by END.

If END is specified, the file must be connected for sequential access.

Example:

```
READ (2,25,IOSTAT = STFLG,ERR = 1200,END = 860) A,B,C
```

6.3.1.2 Input List

The list 'in-list' in the READ statement identifies the items whose values are to be read. An item in an input list must be a variable name, array name, or array element name. If an array name is listed, the entire array is read in normal array element ordering sequence. The name of an assumed-size dummy array must not appear in the input list.

6.3.1.3 Implied-DO List

An implied-DO list embedded in the READ statement allows a range of subscripts to be used for input list array elements. For example, half the items in an array can be read without specifying each individual array element to be read. The format of the implied-DO list is

(in-list, var = e1, e2, e3)

where 'var, e1, e2, and e3' have the same interpretation as for the DO statement (section 4.2.2) and 'in-list' is a list of input items as described above. The list 'in-list' may also contain additional implied-DO lists.

For READ statements, the DO variable 'var' must not appear as an item in 'in-list.'

Example:

```
C READ ONLY THE ODD ELEMENTS IN ARRAY 'TABLE'
      DIMENSION TABLE(60)
      READ (2, 20) (TABLE(N), N = 1, 59, 2)
20    FORMAT ...
```

6.3.2 WRITE Statement

The WRITE statement outputs data to a specified unit. The format of the WRITE statement is

WRITE (*ctl-list*) [*out-list*]

where

ctl-list is a list of control information specifiers

out-list is a list of the data to be written

The control information list is the same as for the READ statement (section 6.3.1.1 and following subsections) except that no END specifer is allowed. The output list 'out-list' is defined in the same manner as the 'in-list' portion of the READ statement, including the implied-DO option (sections 6.3.1.2 and 6.3.1.3).

Like input list items, an output list item may be a variable name, array name, or array element name. An output list item may also be an expression, including an expression involving operators or enclosed in parentheses.

Examples:

```
      WRITE (6,120) PNAME, AVG
120   FORMAT ...

      WRITE (6,120,IOSTAT=ERRFLG, ERR=2000)
     &PNAME + I, AVG + I
120   FORMAT ...

      DIMENSION PNAME(25), AVG(25)
C WRITE DOUBLE COLUMN PRINTOUT OF FIRST ITEMS OF
C EACH ARRAY
      WRITE (6,120) (PNAME(K), AVG(K), K=1, 10)
120   FORMAT (1X, A, 5X, F4.3)
```

6.3.3 PRINT Statement

The PRINT statement outputs formatted data to the default write unit. It has the format

PRINT *f* [,*out-list*]

where

f is a format identifier

out-list is a list of the data to be written

Note that the keyword 'PRINT' does not necessarily imply the default unit is a line printer or other print device.

The format specifier 'f' has the same meaning as for the READ statement (section 6.3.1.1.2). The list 'out-list' is defined for PRINT exactly as it is for WRITE (section 6.3.2).

Examples:

```
      PRINT 120, PNAME, AVG
120   FORMAT ...

      ASSIGN 120 TO OUTFMT
      PRINT OUTFMT, PNAME, AVG
120   FORMAT ...
```

6.4 Formatted And Unformatted Data Transfer

In the description of the OPEN statement we saw that a file can be connected for formatted or unformatted I/O (section 6.2.1.7). The defaults for the formatting specifier are 'UNFORMATTED' if the file is connected for direct access and 'FORMATTED' if the access method is sequential. The formatted or unformatted property is confirmed by the presence or absence of the format specifier

[FMT =] *f*

in READ, WRITE, or PRINT statements. in the case of formatted I/O, the I/O statement is normally used with a FORMAT statement.

6.4.1 Unformatted Data Transfer

The unit specified in a data-transfer statement involving unformatted data must be an external unit. Data is transferred *without* editing between the current record of the connected file and items in the I/O list. Exactly one record is read or written.

The number of items in an input list must not exceed the number of values in the record. The type of each value in the record must agree with the type of the corresponding input list item. The item and its value must also agree in length.

On output, if the file is connected for direct access and the values in the output list do not fill the record, the remainder of the record is undefined.

6.4.2 Formatted Data Transfer

During formatted data transfer, data is transferred *with* editing between the file and the I/O list. The editing is directed by some kind of format specification. Format specifications can be given:

- In FORMAT statements;
- As values of character arrays, character variables, or other character expressions;
- As Hollerith values assigned to integer, real, or logical arrays.

If the format specifier (section 6.3.1.1.2) in a formatted I/O statement is a character array name, character variable name, or other character expression, the value referenced must contain a valid format specification in its leftmost character positions. The format specification is described below (section 6.4.3). Character data may follow the right parenthesis that ends the format specification with no effect on the format specification itself. The same applies to Hollerith data in an integer, real, or logical array.

6.4.2.1 Printing Formatted Records

If a formatted record is printed on some external listing device, the first character of the record is not printed. It is used instead to indicate vertical spacing. The remaining characters of the record are then printed beginning at the left margin.

When this convention is specified, the first character is interpreted as follows:

Character	Vertical Spacing
Blank	One line
0	Two lines
1	Skip to next page
+	No advance

This interpretation is requested by the CARRIAGE specifier of the OPEN statement (section 6.2.1.10). If 'car' is specified as 'FORTRAN', the first character is interpreted as vertical spacing information and is not printed.

6.4.2.2 Format Control

The *edit descriptors* that make up the format specification list (section 6.4.3) are classified as either *repeatable* or *nonrepeatable*.

Both the format specification list and its corresponding I/O list are scanned left to right. One item in the I/O list corresponds to each repeatable edit descriptor. There is no corresponding I/O list item for nonrepeatable edit descriptors, and format control communicates directly with the I/O record. If a repeatable edit descriptor is repeated, say five times, it corresponds to five consecutive I/O list items.

If a format specification list ends before the I/O list, it *reverts* to its beginning (or to the left parenthesis matching the rightmost right parenthesis if the format specification contains nested parentheses). Repeat specifications have the same effect as during the first pass through the format specification list. A new record is begun each time format reversion occurs.

6.4.3 FORMAT Statement

The form of the FORMAT statement is

stl FORMAT ([*flist*])

where

stl	is a 1-5 digit statement label
flist	is a format specification list whose items are separated by commas

Each item in 'flist' must be a repeatable edit descriptor, a nonrepeatable edit descriptor, or a parenthesized 'flist.' An edit descriptor is repeated by prefixing it with a nonzero, unsigned integer constant called a 'repeat specification.' The entire 'flist' can also be prefixed by a repeat specification.

F5.3	Repeatable descriptor
5F5.3	Repeatable descriptor prefixed with repeat specification '5'
X	Nonrepeatable descriptor
3(2X,I5)	Entire 'flist' prefixed with repeat specification '3'

Note that the FORMAT statement with no 'flist' specified, 'FORMAT ( ),' can be used only if the I/O list is also empty. Conversely, if the I/O list is not empty, 'flist' must have at least one repeatable edit descriptor.

6.4.3.1 Edit Descriptors

6.4.3.1.1 Repeatable Edit Descriptors. Repeatable edit descriptors generally consist of a letter indicating the type of data involved and a number indicating the size of the data field and how it is to be divided. The repeatable edit descriptors are:

I*w*	Integer descriptor
F*w.d*	Real number descriptor
E*w.d*	Real number descriptor
E*w.d*E*e*	Real number descriptor
L*w*	Logical descriptor
A	Variable-length alphanumeric descriptor
A*w*	Fixed-length alphanumeric descriptor
B*w*	Binary descriptor
Z*w*	Hexadecimal descriptor

where

I, F, E, L and A	indicate the type of data being edited
B and Z	indicate the number base of data being edited
w	is a nonzero, unsigned integer constant representing the width of the entire edited field
d	is an unsigned integer constant representing the number of digits that should follow the decimal point.
e	is a nonzero, unsigned integer constant representing the width of the exponent field.

The I, F, and E edit descriptors are used to specify I/O of integer and real data. F and E serve the same function on input; E allows output of real numbers in scientific notation.

Certain general remarks apply to all three of these numeric editing descriptors.

- On input, leading blanks are not significant. Other blanks are treated according to the setting of the nonrepeatable descriptors BN and BZ and the value of the BLANK specifier in the OPEN statement (section 6.2.1.9).
- A decimal point in input data overrides the decimal point location specified by an F or E descriptor. The input field may also have more digits than are needed for the processor to approximate the data's value.
- On output, values are right-justified. If necessary, the field is blank-filled on the left.
- On output, if the number of characters exceeds the field width 'w,' or an exponent has more than 'e' digits, the entire field is filled with asterisks.

The B and Z edit descriptors specify data I/O in binary and hexadecimal notation, respectively. The internal representation of the data is output (e.g., the 'Z' format of '-1' is 'FF').

Integer Editing

An I/O list item matched with an 'I*w*' edit descriptor must be of type integer. The input list item is defined with integer data; the output list item must already be defined with integer data. The integer constant read or written always consists of at least one digit.

Examples:

```
      PRINT 20, INTNUM
20    FORMAT (I5)

      READ (2,20) INTNM1, INTNM2, INTNM3
20    FORMAT (2I5, I4)
```

'F' Descriptor Editing

An I/O list item matched with an 'F*w.d*' descriptor must have a real value. If the input to this descriptor contains no decimal point, the rightmost 'd' digits of the string are interpreted as the fractional part of the input value.

On input, a string of digits by the basic 'F' descriptor can be followed by an exponent consisting of a signed integer constant or the letter 'E' followed by an optionally signed integer constant.

Output edited by the 'F' descriptor is rounded to 'd' fractional digits, and may be modified by an established scale factor (see the description of the 'P' nonrepeatable edit descriptor, section 6.4.3.1.2). Leading zeros are not generated unless the output field would be blank otherwise.

Examples:

```
      READ (2,20) REALNM
20    FORMAT (F5.3)

      DIMENSION TABLE (10)
      PRINT 20, TABLE
20    FORMAT 5(F5.3, 2X, F5.3)
C THE TABLE WILL BE PRINTED OUT IN TWO COLUMNS
```

'E' Descriptor Editing

An I/O list item matched with an 'E*w.d*' or 'E*w.d*E*e*' descriptor must be a real variable. The exponent 'e' has no effect on input data.

On output, the format of the output field for a scale factor (section 6.4.3.1.2) of zero is:

$[\pm]\ [0].x_1\,x_2\ldots x_d\,exp$

where

$\pm$	signifies a plus or minus
$x_1 \ldots x_d$	are the 'd' most significant digits of the data's value after rounding
exp	is a decimal exponent having one of the following forms ('y' is a digit):

Edit Descriptor	Absolute value of 'exp'	Form of 'exp'
E*w.d*	$\|exp\| \le 99$	$\pm\ 0y_1\,y_2$
	$99 < \|exp\| \le 999$	$\pm\ y_1\,y_2\,y_3$
E*w.d*E*e*	$\|exp\| \le (10^{**}e)-1$	$E \pm y_1\,y_2\ldots y_e$

The sign in the exponent is always present. If the exponent is zero, it is prefixed by a plus. The 'E*w.d*' descriptor should not be used if *exp* exceeds 999.

Decimal normalization is controlled by the scale factor '*k*' (section 6.4.3.1.2). If $-d < k \le 0$, the number output will have exactly $|k|$ leading zeros and '$d - |k|$' significant digits following the decimal point. If $0 < k < d+2$, the number will have exactly k significant digits to the left of the decimal point and '$d - k + 1$, significant digits to the right of the decimal point. Other values of k are not legal.

Examples:

```
      READ (2,30) RLNUMB
30    FORMAT (E4.2)

      WRITE (6,110) RLOUT
110   FORMAT (E6.5E6)
```

Logical Editing

An I/O item matched with an 'L*w*' descriptor must be of type logical.

The input field includes an optional period followed by a 'T' (for TRUE) or 'F' (for FALSE). These characters may be followed by additional characters. For example, the logical constants '.TRUE.' and '.FALSE.' are acceptable inputs.

The output field consists of the letters 'T' or 'F,' based on the TRUE or FALSE value of the internal data.

Examples:

```
      DIMENSION TRUTH(4)
      READ (3,50) TRUTH(1), TRUTH(4)
50    FORMAT (2L6)

      WRITE (6,80) TRUTH(1)
80    FORMAT (L1)
```

Alphanumeric Editing

An I/O item matched with an 'A' or 'A*w*' descriptor must have type character or be defined with Hollerith data. If the field width 'w' is specified, the field consists of 'w' characters. Otherwise, the number of characters in the field is the length of the I/O list item.

On input, if the character string is longer than the specified width, the string is truncated on the right. If the specified width exceeds the length of the character string, the string is blank-filled on the right. The same is true on output, except that blank filling is done on the left.

Number Base Editing

An input item matched with a 'B*w*' or 'Z*w*' descriptor must consists only of binary or hexadecimal digits, respectively. In particular, such an input field cannot contain either a sign or a letter indicating the base.

If one of these descriptors is specified for output, 'w' characters are output in the number base indicated. Leading zeros are supplied on output and accepted on input, to be sure there are as many digits present as are needed to represent the data. For example, if 'I' is a single-byte integer whose value is '4,' it is output as 'bbb04' under a 'Z5' edit descriptor.

The number base edit descriptors can be specified for data of any type.

6.4.3.1.2 Nonrepeatable Edit Descriptors. The nonrepeatable edit descriptors are:

'$h_1\ h_2 \ldots h_n$'	Literal string descriptor
nH$h_1\ h_2 \ldots h_n$	*Hollerith string descriptor*
nX	Record position control descriptor
/	Record termination descriptor
kP	Scale factor descriptor
BN	Blank descriptor
BZ	Blank descriptor
$	Alternate record termination descriptor

where apostrophe ('), H, X, slash (/), P, BN, BZ, and the dollar sign ($) indicate the kind of editing and

h	is any character representable on the processor
n	is a nonzero, unsigned, integer constant
k	is an optionally-signed integer constant representing a scale factor

Apostrophe Editing

The apostrophe edit descriptor can be used only for output. It causes the characters enclosed in apostrophes to be written out literally. To indicate an apostrophe within the character field, show it as two consecutive apostrophes.

The width of the field is the length of the character string.

Example:

```
      WRITE (7,100) ITSTNO
100   FORMAT ('THIS IS THE TEST NUMBER', 2X, I2)
```

'H' Descriptor Editing

The Hollerith field descriptor is an alternate way to perform the same operation as apostrophe editing. Like apostrophe editing, it is used only for output. The '*n*H' descriptor causes the 'n' characters following the 'H' to be written out (including embedded blanks).

Because the Hollerith field descriptor relies on an accurate character count to produce the correct output, apostrophe editing is likely to be less error prone than this method.

Example:

```
      WRITE (7,100) ITSTNO
100   FORMAT (1H1, 19HTHIS IS TEST NUMBER, 2X, I2)
C FIRST H DESCRIPTOR CAUSES SKIP TO NEW PAGE
```

'X' Descriptor Editing

The '*n*X' descriptor indicates that the next character transferred to or from a record is the character 'n' positions from the current record position. On output, the effect is to insert 'n' blanks into the output record.

Example:

```
      WRITE (7,100) ITSTNO
100   FORMAT (1X, 'THIS IS TEST NUMBER', 2X,I2)
C FIRST X DESCRIPTOR CAUSES SINGLE SPACING BY
C INSERTING A BLANK AS THE FIRST CHARACTER OF
C THE RECORD
```

Slash Editing

The slash (/) edit descriptor acts as an end-of-record indicator. On input, the remainder of the current record is skipped or, if the file is positioned at the beginning of a record, the entire record is skipped.

On output, the current record is terminated and a new record is begun. The slash edit descriptor may also be used to write an empty record, which is a convenient way to provide blank lines on printed output.

The comma that normally separates format specification list items is not required before or after a slash.

Example:

```
      WRITE (7, 100)
100   FORMAT (1H1, '    PLAYER     AVERAGE'/)
CTHIS SLASH CAUSES BLANK LINE FOLLOWING HEADING
      WRITE (7,150) PNAME, AVG
150   FORMAT (1X, A12, 4X, F4.3)
```

Scale Factor (P) Editing

A scale factor is established by the '*k*P' edit descriptor, where '*k*' represents the scale factor. It is used with the 'F' and 'E' descriptors to edit real numbers. No comma is needed between the 'P' descriptor and an immediately following 'F' or 'E.'

```
1PE8.6E2
```

A scale factor of zero is assumed at the beginning of an I/O statement. Once it has been changed by the '*k*P' edit descriptor, the new scale factor remains in effect until the '*k*P' descriptor is issued again or until the end of the I/O statement.

On input, the scale factor has no effect if there is an exponent in the 'F' or 'E' edited field. Otherwise, the effect of the scale factor is that the externally-represented number equals the internally represented number multiplied by '10**k.' The same is true of output with 'F' editing. On output with 'E' editing, the basic real constant part of the quantity to be produced is multiplied by '10**k' and the exponent is reduced by 'k.'

As we saw in the description of 'E' descriptor editing, the output range of a multiplier printed in scientific notation with a scale factor of zero is 0.1 to 1.0. Changing the scale factor to one changes the multiplier range to 1.0 - 10.0. Changing the scale factor is useful for very large or very small 'E' edited numbers, but is generally not desirable for 'F' edited numbers. Following specification of a nonzero scale factor, the scale factor should probably be respecified as '0P' before the next occurrence of 'F' editing.

The following table of number representations illustrates instances where 'E' editing and the use of the scale factor would be most applicable. The column headings show the field descriptor used to produce each representation.

Real Number	F6.2	E10.5	1PE10.4
4.32	4.32	0.43200E+01	4.3200E+00
7255000.0	******	0.72550E+07	7.2550E+06
0.0065	0.01	0.65000E-02	6.5000E-03

Clearly, 'F' descriptor editing is preferable for simple numbers like '4.32.' Just as clearly, 'F' descriptor editing is inadequate for very large or very small numbers like '7255000' or '0.0065.'

'BN' and 'BZ' Editing

These two edit descriptors are used for input only and affect only numeric editing. They can be used to specify the interpretation of blanks, other than leading blanks. If 'BN' is specified, blanks are ignored except that a field of all blanks is treated as zero. If 'BZ' is specified, blanks are regarded as zeros.

Until the 'BZ' or 'BN' descriptor is specified (or, if neither is specified), the BLANK specifier in the OPEN statement (section 6.2.1.9) determines the interpretation of blanks.

Example:

```
      READ (2,50) INTNUM, FPNUM
50    FORMAT (BN, I5, 5X, F7.4)
```

Dollar Sign Editing

The dollar sign ($) edit descriptor is used primarily for interactive I/O through a console terminal. It leaves the terminal cursor at the position immediately following the I/O data just processed, rather than beginning a new line. If the format control

scanner encounters a dollar sign at the end of a format specification list, format control terminates without positioning the file to the beginning of the next record. The dollar sign edit descriptor has no effect on direct-access files.

Example:

```
      PRINT 25, PNAME
25    FORMAT (A20, $)
```

6.4.4 List-Directed Formatting

List-directed formatting is indicated by specifying an asterisk (*) in the format specifier of a data-transfer I/O statement's control list (section 6.3.1.1.2). It allows free form input and output, which is especially helpful if the I/O device is a console terminal. No FORMAT statement is required as all necessary formatting is done for the programmer.

A list-formatted file consists of a string of values and value separators. Each value is either a constant, a null value, or a constant or null value prefixed by a repeat specification in the form:

*r*c*
*r**

The first form is equivalent to 'r' successive appearances of the constant 'c;' the second is equivalent to 'r' null values. A null value can also be specified as no characters between value separators or no characters preceding the first value separator in a record. The null value is not produced by list-directed output.

Value separators can be a comma, slash, or one or more blanks between constants or following the last constant in a record. The slash separator is not produced on output. When encountered during list-directed input, it terminates execution of the input statement after assignment of the previous value. If the I/O list contains additional items, they are effectively assigned null values.

Any sequence of blanks is treated as a single blank except when it appears within a character constant. An end-of-record has the same effect as a blank.

6.4.4.1 List-Directed Input

Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting with a few exceptions. Blanks are never treated as zeros, and may not be embedded in constants (except character constants).

An input list item of type real is assumed to have no fractional part in its input form unless a decimal point appears within the field. An input list item of type logical must not include either slashes or commas in its input form.

An input list item of type character consists, in its input form, of a nonempty string of characters enclosed in apostrophes. Character constants can be continued from the end of one record to the beginning of the next. Though the end of a record normally has the effect of a blank in list-directed formatting, a blank is not inserted into the character constant in this case. Character constants are transferred left-justified and are truncated on the right if their length exceeds the width of the input list item.

A null value has no effect on a corresponding input list item. The item retains its previous value or remains undefined, depending on its status before the null value was encountered.

6.4.4.2 List-Directed Output

The form of the values produced by list-directed output agrees in type with their corresponding output list items. The processor separates records as necessary, so long as the end of a record does not fall within a constant (except a character constant) and blanks are not embedded within constants.

Integer output constants are produced with the effect of 'I*w*' formatting, for some reasonable value of 'w.' Real constants are produced with the effect of 'F' or 'E' editing, depending on the magnitude of the value. Where reasonable, a scale factor of '1P' is used. Logical constants are 'T' for TRUE and 'F' for FALSE. Character constants are output simply as character strings without surrounding apostrophes.

Output records are single spaced.

CHAPTER 7
PROGRAMMING GUIDELINES

This chapter is by no means intended as an exhaustive discussion of programming techniques. It is intended simply as a guideline, primarily for the novice programmer. For those who wish to go more deeply into the science of FORTRAN programming, a number of references are listed in the bibliography at the end of the chapter.

The first section of this chapter deals with general guidelines applicable to programming in any language. The second section lists suggestions for programming in FORTRAN specifically.

7.1 Program Development

The recommended approach to program development is the so-called 'top-down' method. Essentially, this means defining a program in the broadest possible terms initially, and then working down through a series of increasingly detailed steps to final code. At each level, debugging is performed to whatever extent possible before going to the next level of refinement.

The first step in this approach is a thorough definition of the programming task.

7.1.1 Problem Definition

Before considering any actual programming, one must understand clearly and completely the problems involved. For example, a person may have to change the spark plugs on his car. This is fine, but he will be better prepared if he knows that inserting the new plugs by hand first and using a plug wrench only for final tightening reduces the possibility of stripping threads. And he will be better advised still if he is told to put a trace of 'anti-seize' compound on the threads first to make later removal easier, and limit the chance of breaking a plug.

Similarly, knowing that a program must print out payroll statistics is insufficient. Is it to print out only employee names and net pay? Is it to show taxes withheld (federal, social security, state, city, ad infinitum)? Should it show other deductions for the company's stock plan, pension plan, credit union, or whatever? Should it show the hours worked, splitting out overtime, shift differential, holiday, or vacation pay? Should it show accrued sick or vacation leave? And what format should be used to display all this information?

While the progammer need not state the specific algorithms to be used at this stage, he at least needs to know in detail what input the program will be receiving and in what form, and exactly what output the program is expected to produce.

7.1.2 Program Documentation

Every program should have *good* documention *from the beginning*!

At the earliest stage of program development, documentation would normally be a preliminary functional specification of the program. Ideally, this specification should be reviewed by one's programming peers for constructive criticism. Not only will gaps be filled in, but this review creates an environment for exchanging theories of programming, for coworkers to familiarize themselves with each other's projects, and for developing a feeling of teamwork within a programming group.

As the development of a program becomes more detailed, the documentation should become correspondingly detailed. The ultimate documentation is, of course, the final program code, which should include numerous useful comments, have meaningful mnemonic names for symbols, and make good use of blanks in statement lines to improve program readability. If the program is not severely limited by processor memory size, each program unit could be prefaced with a comment block. These comments could include more than a description of the program unit's function; for example, it might say who coded this unit originally and who made the latest changes.

7.1.3 Refining The Problem Definition

Once the problem is defined in detail, a series of refinements is begun, with each level in the series being increasingly detailed.

For the sake of example, let's assume the payroll statistics printout task has been defined as follows:

- The input will be a formatted file on diskette containing a record for each employee. This record contains all information related to hours worked, pay rates, deductions, etc.
- The program is to print out (on the line printer) only the employee's name (EMP field), the hours worked (HRS), the gross pay (GRPAY), and net pay (NETPAY).
- After the last record has been printed, the program is to print a summary showing the total number of employees, the total hours worked, and the total (gross) pay disbursed.

The first two levels of program development might look like:

Level One

Print out payroll statistics on line printer.

Level Two

Initialize variables.

Open files.

Read input record.

```
    If (last record) then

        Print totals
        Close files

    Else

        Print individual statistics
        Update totals
        Go to 'read record' statement
```

Note in level two that we are still using essentially English sentences with a few words here and there that begin to look like FORTRAN. The program structure is beginning to take shape, but at this point we're more concerned with logic than FORTRAN code.

The next pass is a more formal description of level two, introducing FORTRAN statements for English sentences.

Level Three

```
C  INITIALIZE VARIABLES NEEDED FOR TOTALS
         DATA TOTEMP, TOTHRS, TOTPAY /  3*0.0/

C  OPEN INPUT/OUTPUT FILES, READ EMPLOYEE RECORD
         OPEN (input file)
         OPEN (output file)
   10    READ (unit, 20) EMP, HRS, GRPAY, NETPAY
   20    FORMAT (flist)

C  PRINT TOTALS IF NO MORE RECORDS
         IF (no more records) THEN
         WRITE (unit, 40) TOTEMP, TOTHRS, TOTPAY
   40    FORMAT (flist)
         CLOSE (input file)
         CLOSE (output file)

C  OTHERWISE PRINT EMPLOYEE DATA AND UPDATE TOTALS
         ELSE
         WRITE (unit, 60) EMP, HRS, GRPAY, NETPAY
   60    FORMAT (flist)
         TOTEMP = TOTEMP + 1
         TOTHRS = TOTHRS + HRS
         TOTPAY = TOTPAY + GRPAY

C  READ NEXT EMPLOYEE RECORD
         GO TO 10
         END IF
         END
```

A number of details remain to be specified, but each of these levels is in some sense complete and can be debugged to the extent that it is complete. Thus we can confirm the accuracy of the program's logic, then I/O interfaces and basic calculations, and only at the end concern ourselves with such details as format specification.

7.1.4 Final Coding

At the level of detailed code, a number of steps can be taken to simplify writing and debugging the program, or to simplify the task of another programmer updating the program later.

- Take advantage of the built-in debugging aids available in the particular programming language. For example, PAUSE, STOP, and WRITE statements can be interspersed throughout initial FORTRAN code to trace program execution paths. These can be removed in the final version of the program.
- Avoid tricky programming. Code conservatively!
- Concentrate initially on making the program work. Beautiful printouts can be produced as a last step.
- Concentrate especially on getting the statement syntax correct the first time. Syntax details can be particularly annoying to FORTRAN programmers, but initial effort in this area can save a lot of grief in the long run.
- Again, use comment lines frequently. Use meaningful labels for variables. Use blanks to improve program listing readability.

This is just a 'starter' list. Certainly, any experienced programmer could add to this checklist. Rereading such a list frequently, like rereading programming manuals, is a good way to refresh or reconfirm programming knowledge.

7.2 FORTRAN Coding

Section 7.1 lists some general programming considerations. When coding in FORTRAN specifically, other points should be kept in mind.

7.2.1 Functions And Subroutines

The first point has already been made in Chapter 5, but is worth repeating. Use functions and subroutines in a program wherever it makes sense to do so. First of all, they reduce the amount of coding to be done, saving time and reducing the chance of error. They also save processor memory by allowing shorter programs. And, most importantly, they break a program into units that can be separately programmed and debugged and that also clarify its logical structure, making it easier to understand.

7.2.2 GO TO Statement

The GO TO statement should be used only when necessary. The ability to jump around at will within a program can be a strong temptation to neglect logical planning. No painter would worry about painting himself into a corner if he could escape by simply shouting 'go to exit.' When the GO TO seems necessary, consider first whether an alternative solution that would improve the logical structure of the program has been overlooked.

7.2.3 Crossing Unit Lines

The ability to divide a program into subprograms is a major benefit of FORTRAN. It also has some potential pitfalls. Be careful when using global variables, external procedures, and variables whose values have been computed outside the current program unit! Take advantage of the capability provided by common memory, but be aware of the interaction among all the program units that reference common memory!

7.2.4 Computing Variables And Constants

Complex calculations can frequently be simplified by breaking them into several steps and computing intermediate variables. This is particularly true if such variables are used several times after their value has been computed. Program execution time can be reduced by using intermediate variables and the program is generally more readable. Like most programming tools, however, the use of intermediate variables can be abused and requires good judgement.

When calculating a value for use in a DO loop, be sure the value is computed before the loop is entered. Otherwise, the program could compute the value again for each iteration of the loop. Consider the following short Examples:

```
C  EXAMPLE OF RECOMPUTED CONSTANT
        INTEGER*1  R
        DO  25  R = 1, 60
        X = (22/7)*(r**2)
        WRITE (4) X, R
25      CONTINUE

C  SAME EXAMPLE WITH PRECOMPUTED CONSTANT
        INTEGER*1  R
        PI = 22/7
        DO  25  R = 1, 60
        X = PI*(R**2)
        WRITE (4) X, R
25      CONTINUE
```

In the first case, the value '22/7' would have to be computed 60 times!

7.2.5 Reminders

We've already mentioned the use of comments, good mnemonic names, and blanks to improve program readability and understandability. We've also mentioned debugging aids available in the FORTRAN language (such as the PAUSE, STOP, and WRITE or PRINT statements). In addition, the programmer should explore other debugging tools that might be available in his system environment, such as the DEBUG command in Intel's ISIS-II or Intel's in-circuit emulator family, which includes ICE-80 and ICE-85 for the 8080 and 8085 microprocessors.

7.3 References

The following list suggests material for further reading. Some of the material, like the books by Ledgard and McCracken or the article by Ogdin, is essentially tutorial. Bear in mind that all of this material was written before FORTRAN 77 was specified and FORTRAN 'limitations' discussed in these works may no longer be a problem in the new version.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming*, Academic Press, New York, 1972.

Dijkstra, Edgar W., 'GO TO Statement Considered Harmful,' *Communications of the ACM*, Vol. 15, No. 10, Oct. 1972.

Henderson, P., and Snowdon, R., 'An Experiment in Structured Programming,' *BIT* 12, 1972.

Hilburn, J.L., and Julich, P.M., *Microcomputers/Microprocessors: Hardware, Software, and Applications*, Prentice-Hall, Inc., 1976.

Knuth, Donald E., *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

Ledgard, Henry F., *Programming Proverbs for FORTRAN Programmers*, Hayden, Rochelle Park, N.J., 1975.

McCracken, Daniel A., *A Simplified Guide to FORTRAN Programming*, John Wiley & Sons, New York, 1974.

Mills, Harlan B., *Mathematical Foundations for Structured Programming, Technical Report, FSC 72-6012,* IBM Federal Systems Division, Gaithersburg, Md., 1972.

Ogdin, Carol A., 'Software Design Course,' *EDN*, June 5, 1977.

Wirth, Niklaus, 'Program Development by Stepwise Refinement,' *Communications of the ACM*, Vol. 14, No. 4, April 1971.

APPENDIX A
FORTRAN-80 STATEMENT SUMMARY

A.1 Statement Sequence

The following order of statements and lines must be observed when coding a FORTRAN program.

1. Comment lines can appear before or between statements, but cannot appear after an END statement.
2. The PROGRAM statement, if used, must be the first statement of a main program. The first statement of a subprogram must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

Within a program unit that permits the following statements:

3. FORMAT statements can appear before the END statement.
4. IMPLICIT statements must precede all other specification statements.
5. All specification statements must precede all DATA statements, which must precede statement function statements, which must precede all executable statements.
6. The last line of a program unit must be an END statement.

These rules are summarized in Figure 1-2.

A.2 Statement Summary

In the following summary, any format item enclosed in square brackets is optional. Ellipses indicate the preceding item can be repeated indefinitely (within statement length limits).

ASSIGN Statement

Category: Executable

Text Reference: Section 3.3.4

Format	Function
ASSIGN stl TO name	Assign statement label 'stl' to integer variable 'name.'

Assignment Statement

Category: Executable

Types: Arithmetic, Character, Logical

Text Reference: Sections 3.3.1, 3.3.2, 3.3.3

Format	Function
v = e	Assign value of 'e' to 'v,' where 'v' is type integer, real, logical, or character.

BACKSPACE Statement

Category: Executable

Text Reference: Section 6.2.3

Formats	Function
BACKSPACE unit BACKSPACE (arg-list)	Position file connected to 'unit' before preceding record.

Notes:

'unit' is unit specifier

'arg-list' is following argument list:

[UNIT =] unit	unit specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier

BLOCK DATA Statement

Category: Nonexecutable

Text Reference: Sections 3.4.3, 3.4.4

Format	Function
BLOCK DATA [name]	Identify start of BLOCK DATA subprogram; optionally name subprogram 'name.'

CALL Statement

Category: Executable

Text Reference: Section 5.2.7

Format	Function
CALL sub [([a[,a]...])]	Call subroutine 'sub' with actual argument(s) 'a.'

CHARACTER Statement

Category: Nonexecutable, specification, type

Text Reference: Section 3.1.4

Format	Function
CHARACTER [*len[,]]name[,name]...	Specify name and length for character type variable or array.

CLOSE Statement

Category: Executable

Text Reference: Section 6.2.2

Format	Function
CLOSE (close-list)	Close file (disconnect unit) described by 'close-list.'

Notes:

'close-list' can be following fields:

[UNIT =] unit	unit specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier
STATUS = stat	file disposition specifier

Comment Line

Category: Nonexecutable

Text Reference: Section 1.1.1

Format	Function
'C' or '*' in column 1; any ASCII character in columns 2-72	Program documentation

COMMON Statement

Category: Nonexecutable, specification

Text Reference: Section 3.4.2

Format	Function
COMMON [/[cb]/] nlist[[,]/[cb]/nlist]	Name and define contents of common block(s) 'cb.'

CONTINUE Statement

Category: Executable

Text Reference: 4.2.3

Format	Function
CONTINUE	No effect unless this is terminal statement of a DO loop; then action depends on DO variable.

DATA Statement

Category: Nonexecutable

Text Reference: Section 3.3.5

Format	Function
DATA nlist/clist/[[,] nlist/ clist/]...	Assign values in 'clist' to items in 'nlist.'

DIMENSION Statement

Category: Nonexecutable, specification

Text Reference: Section 3.2.1

Format	Function
DIMENSION a(d) [,a(d)]...	Name array(s) 'a' and define dimension(s) 'd.'

DO Statement

Category: Executable

Text Reference: Section 4.2.2

Format	Function
DO stl [,] var = e1, e2 [,e3]	Define beginning of DO loop and set up loop counters.

Notes:

stl	label of last (executable) statement in DO loop
var	DO variable
e1	initial loop index value
e2	loop termination value
e3	loop increment/decrement amount

ELSE Statement

Category: Executable, block IF

Text Reference: Section 4.1.9

Format	Function
ELSE	Continue execution; provides alternate execution path from IF or ELSE IF.

ELSE IF Statement

Category: Executable, Block IF

Text Reference: Section 4.1.8

Format	Function
ELSE IF (exp) THEN	Continue execution if expression 'exp' is TRUE

END Statement

Category: Executable

Text Reference: 4.3.1

Format	Function
END	Terminate main program; return from subprogram; mark end of program unit.

END IF Statement

Category: Executable, block IF

Text Reference: Section 4.1.10

Format	Function
END IF	Mark end of IF block; continue execution.

ENDFILE Statement

Category: Executable

Text Reference: Section 6.2.5

Formats	Function
ENDFILE unit ENDFILE (arg-list)	Write end-of-file record on file connected to 'unit.'

Notes:

'unit' is unit specifier
'arg-list' is following argument list:

[UNIT =] unit	unit specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier

EQUIVALENCE Statement

Category: Nonexecutable, specification

Text Reference: Section 3.4.1

Format	Function
EQUIVALENCE (nlist) [,(nlist)]...	Allow entities in 'nlist' to share the same storage area.

EXTERNAL Statement

Category: Nonexecutable, specification

Text Reference: Section 5.2.6

Format	Function
EXTERNAL proc [,proc]...	Allows name of external/dummy procedure to be used as an actual argument.

FORMAT Statement

Category: Nonexecutable

Text Reference: Section 6.4.3

Format	Function
stl FORMAT ([flist])	Specify format of formatted I/O data.

Notes:

'flist' includes the following repeatable and nonrepeatable edit descriptors.

Repeatable		Nonrepeatable	
Iw	integer	'string'	literal
Fw.d	real no.	nHstring	Hollerith
Ew.d	real no.	nX	record position
Ew.dEe	real & exponent	/	record termination
Lw	logical	kP	scale factor
A	alphanumeric	BN	blank
Aw	alphanumeric	BZ	blank
Bw	binary	$	alternate record termination
Zw	hexadecimal		

FUNCTION Statement

Category: Nonexecutable

Text Reference: Section 5.2.1

Format	Function
[type] FUNCTION func ([d[,d]...])	Name the FUNCTION subprogram 'func,' define its type and dummy parameter(s) 'd.'

GO TO Statements

Category: Executable

Text Reference: Section 4.1.1, 4.1.2, 4.1.3

Formats	Function
GO TO s GO TO (s[,s]...)[,]exp GO TO i [[,](s[,s]...)]	Transfer control to statement labeled 's' or ASSIGNED to variable name 'i'.

Notes:

First format branches unconditionally.
Second format branches based on value of integer expression 'exp.'
Third format branches unconditionally, but statement label corresponding to 'i' must be included in list.

IF Statements

Cagetory: Executable

Text Reference: Sections 4.1.4, 4.1.5, 4.1.6, 4.1.7

Formats	Function

IF (e) s1, s2, s3
IF (e) st
IF (e) THEN
	Transfer control to a specified statement or perform specified action(s) based on the value of expression 'e'.

Notes:

In first format 'e' is an arithmetic expression and s1, s2, s3 are standard labels; control passes to:

s1 if e<0
s2 if e = 0
s3 if e>0

In second format, statement 'st' is executed if logical expression 'e' is TRUE. Third format introduces IF block; statements following IF-THEN are executed if logical expression 'e' is TRUE.

IMPLICIT Statement

Category: Nonexecutable, specification

Text Reference: Section 3.1.5

Format	Function
IMPLICIT type (l[, l]...)[,type(l[,l]...)]...	Define implicit typing for variable names whose first letter is 'l' or in range 'l,l.'

INTEGER Statement

Category: Nonexecutable, specification, type

Text Reference: Section 3.1.2

Format	Function
INTEGER [* len [,]] name [,name]...	Define 'name' to be of type integer with length 'len.'

INTRINSIC Statement

Category: Nonexecutable, specification

Text Reference: Section 5.1.2

Format	Function
INTRINSIC func [,func]...	Allow intrinsic function(s) 'func' to be used as actual argument(s).

LOGICAL Statement

Category: Nonexecutable, specification, type

Text Reference: Section 3.1.3

Format	Function
LOGICAL [* len [,]] name [,name]...	Define 'name' to be of type logical with length 'len.'

OPEN Statement

Category: Executable

Text Reference: Section 6.2.1

Format	Function
OPEN (open-list)	Open the specified file (connect file to unit).

Notes:

'open-list' consists of the following specifiers:

[UNIT =] unit	unit specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier
FILE = fname	file name specifier
STATUS = stat	file status specifier
ACCESS = acc	access method specifier
FORM = fmat	formatting specifier
RECL = reclen	record length specifier
BLANK = blnk	blank specifier
CARRIAGE = car	carriage control specifier

PAUSE Statement

Category: Executable

Text Reference: Section 4.3.1

Format	Function
PAUSE [msg]	Halt program execution; resume under control of external signal; 'msg' is 1-5 digits or a character constant.

PRINT Statement

Category: Executable

Text Reference: Section 6.3.3

Format	Function
PRINT f [,outlist]	Output items in 'outlist' to preconnected unit in format specified by 'f.'

PROGRAM Statement

Category: Nonexecutable

Text Reference: Section 2.1.2

Format	Function
PROGRAM name	Name main program 'name;' must be first statement if used.

READ Statement

Category: Executable

Text Reference: Section 6.3.1

Formats	Function
READ (ctl-list) [inlist] READ f [,inlist]	Input items in 'inlist' as directed by specified controls.

Notes:

'ctl-list' includes the following specifiers:

[UNIT =] unit	unit specifier
[FMT =] f	format specifier
REC = recno	record number specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier
END = stl	end-of-file specifier

Second format is for preconnected units; 'f' is the format specifier.

REAL Statement

Category: Nonexecutable, specification, type

Text Reference: Section 3.1.1

Format	Function
REAL name [,name]...	Define 'name' to be of type real.

RETURN Statement

Category: Executable

Text Reference: Section 5.2.4

Format	Function
RETURN	Return from FUNCTION or SUBROUTINE subprogram.

REWIND Statement

Category: Executable

Text Reference: Section 6.2.4

Formats	Function
REWIND unit REWIND (arg-list)	Reposition file connected to 'unit' at its initial point.

Notes:

'arg-list' includes the following specifiers:

[UNIT =] unit	unit specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier

SAVE Statement

Category: Nonexecutable, specification

Text Reference: Section 5.2.5

Format	Function
SAVE /cb / [,/cb/]...	Save data in common block 'cb' on return from subprogram.

Statement Function Statement

Category: Nonexecutable

Text Reference: Section 5.1.3

Format	Function
func ([d[,d]...]) = exp	Define function 'func' with dummy argument(s) 'd;' 'exp' is an expression.

STOP Statement

Category: Executable

Text Reference: Section 4.3.2

Format	Function
STOP [msg]	Terminate program execution; 'msg' is 1-5 digits or a character constant.

SUBROUTINE Statement

Category: Executable

Text Reference: Section 5.2.2, 5.2.3

Format	Function
SUBROUTINE sub [([d[,d]...])]	Define SUBROUTINE subprogram 'sub' with dummy argument(s) 'd.'

WRITE Statement

Category: Executable

Text Reference: Section 6.3.2

Format	Function
WRITE (ctl-list) [outlist]	Output item in 'outlist' as directed by controls in 'ctl-list.'

Notes:

'ctl-list' includes the following specifiers:

[UNIT =] unit	unit specifier
[FMT =] f	format specifier
REC = recno	record number specifier
IOSTAT = stname	I/O status specifier
ERR = stl	error specifier

APPENDIX B
INTRINSIC FUNCTIONS

The following table lists the intrinsic (or predefined) functions available with FORTRAN-80. An intrinsic function is executed in an expression by referencing its name followed by some argument in parentheses. If more than one argument is needed, they are separated by commas and all arguments must be of the same type. All angles are expressed in radians.

```
C     = SQRT(A**2 + B**2)
K     = I + MOD(M, N)
PAY   = BASE*40.0 + (1.5*BASE)*(AMAX1(0.0,HOURS-40.0))
```

The list of functions is qualified by the notes following the list. See also the discussion of intrinsic functions in section 5.1.1.

B.1 Intrinsic Function Summary

FORM	CATEGORY	FUNCTION	TYPE OF	
			ARGUMENTS	FUNCTION
INT (a) (note 1)	Type conversion	Convert *a* to type integer	Real	Integer
IFIX (a) (note 1)	Type conversion	Convert *a* to type integer	Real	Integer
REAL (a) (note 2)	Type conversion	Convert *a* to type real	Integer	Real
FLOAT (a) (note 2)	Type conversion	Convert *a* to type real	Integer	Real
ICHAR (a) (note 3)	Type conversion	Convert *a* to type integer	Character	Integer
AINT (a) (note 1)	Truncation	Truncate *a* to integer value	Real	Real
ANINT (a)	Rounding	Round *a* to nearest whole number	Real	Real
NINT (a)	Rounding	Round *a* to nearest integer	Real	Integer
IABS (a)	Absolute value	Return absolute value of *a*	Integer	Integer
ABS (a)	Absolute value	Return absolute value of *a*	Real	Real
MOD (a1, a2) (notes 1, 4)	Remaindering	Return remainder from *a1*/*a2*	Integer	Integer
AMOD (a1, a2) (notes 1,4)	Remaindering	Return remainder from *a1*/*a2*	Real	Real
ISIGN (a1, a2) (note 5)	Sign transfer	Transfer sign of *a2* to *a1*	Integer	Integer

FORM	CATEGORY	FUNCTION	TYPE OF ARGUMENTS	TYPE OF FUNCTION
SIGN (a1, a2) (note 5)	Sign transfer	Transfer sign of *a*2 to *a*1	Real	Real
IDIM (a1, a2)	Positive difference	Return *a*1-*a*2 if >0; otherwise 0	Integer	Integer
DIM (a1, a2)	Positive difference	Return *a*1-*a*2 if >0; otherwise 0	Real	Real
MAX0 (a1,...,an)	Largest value	Select largest value from list	Integer	Integer
AMAX1 (a1,...,an)	Largest value	Select largest value from list	Real	Real
AMAX0 (a1,...,an)	Largest value	Select largest value from list	Integer	Real
MAX1 (a1,...,an)	Largest value	Select largest value from list	Real	Integer
MIN0 (a1,...,an)	Smallest value	Select smallest value from list	Integer	Integer
AMIN1 (a1,...,an)	Smallest value	Select smallest value from list	Real	Real
AMIN0 (a1,...,an)	Smallest value	Select smallest value from list	Integer	Real
MIN1 (a1,...,an)	Smallest value	Select smallest value from list	Real	Integer
SQRT (a)	Square root	Return $\sqrt{a}$ for a >0	Real	Real
EXP (a)	Exponential	Return e**a	Real	Real
ALOG (a)	Natural logarithm	Return log (*a*) for *a*>0	Real	Real
ALOG10 (a)	Common logarithm	Return log 10 (*a*) for *a*>0	Real	Real
SIN (a) (note 6)	Sine	Return sine of *a*	Real	Real
COS (a) (note 6)	Cosine	Return cosine of *a*	Real	Real
TAN (a) (note 6)	Tangent	Return tangent of *a*	Real	Real
ASIN (a) (note 7)	Arcsine	Return arcsine of *a*	Real	Real
ACOS (a) (note 8)	Arccosine	Return arccosine of *a*	Real	Real
ATAN (a) (note 9)	Arctangent	Return arctangent of *a*	Real	Real
ATAN2 (a1,a2) (note 9)	Arctangent	Return arctangent of *a*1/*a*2	Real	Real
SINH (a)	Hyperbolic sine	Return hyperbolic sine of *a*	Real	Real
COSH (a)	Hyperbolic cosine	Return hyperbolic cosine of *a*	Real	Real
TANH (a)	Hyperbolic tangent	Return hyperbolic tangent of *a*	Real	Real

B.2 Notes On Intrinsic Functions

1. For an integer argument, 'int(a) = a.' For a real argument, two possibilities exist. If $|a| < 1$, int(a) = 0; if $|a| \geq 1$, 'int(a)' is the integer whose magnitude is the largest integer that does not exceed the magnitude of 'a' and whose sign is the same as the sign of 'a.' For example,

   int(-12.8) = −12

   For an argument of type real, 'IFIX(a)' is the same as 'INT(a).'

2. For a real argument, 'REAL(a)' is 'a.' For an integer argument, 'REAL(a)' is as much precision of the significant part of 'a' as a real datum can contain.

   For a real argument, 'FLOAT(a)' is the same as 'REAL(a).'

3. ICHAR provides a way to convert from characters to integers, based on the position of the character in the processor collating sequence (Appendix E). The first character in the collating sequence corresponds to position 0 and the last to position 'n-1,' where 'n' is the number of characters in the collating sequence.

   The value of ICHAR(a) is an integer in the range $0 \leq$ ICHAR(a) $\leq$ n-1, where 'a' is an argument of type character and length one. The value of 'a' must be a character capable of representation in the processor.

4. The result for MOD and AMOD is undefined when the value of the second argument is zero.

5. If the value of the first argument of ISIGN or SIGN is zero, the result is zero (which is neither positive nor negative).

6. The absolute value of the argument of SIN, COS, and TAN is not restricted to be less than 2*PI.

7. The absolute value of the argument of ASIN must be $\leq$ 1. The range of the result is -PI/2 $\leq$ result $\leq$ PI/2.

8. The absolute value of the argument of ACOS must be $\leq$ 1. The range of the result is $0 \leq$ result $\leq$ PI.

9. The range of the result for ATAN is -PI/2 $\leq$ result $\leq$ PI/2. If the value of 'a1' is positive, the result is positive, and vice-versa. If the value of 'a1' is zero, the result is zero if 'a2' is positive, and 'PI' if 'a2' is negative. If 'a2' is zero, the absolute value of the result is PI/2. Both arguments cannot be zero.

   The range of the result for ATAN2 is -PI $\leq$ result $\leq$ PI.

APPENDIX C
HOLLERITH DATA TYPE

The Hollerith data type is a carryover from FORTRAN 66. Generally speaking, the character data type provides a superior processing capability, and Hollerith has been retained in FORTRAN-80 primarily for compatibility with the earlier standard.

C.1 Hollerith As A Data Type

Although Hollerith is a data type, a symbolic name cannot be of type Hollerith. Hollerith data (other than Hollerith constants) are identified under the guise of an integer, real, or logical type name. It cannot be identified as type character.

Integer, real, or logical items can be defined with a Hollerith value using either the DATA or READ statements. Totally associated items then become associated with that Hollerith value also. When such a definition occurs, the defined item loses its integer, real, or logical characteristic.

C.2 The Hollerith Constant

The format of a Hollerith constant is

$$n\mathrm{H}h_1h_2...h_n$$

where 'n' is a nonzero, unsigned, integer constant and 'h' is any character representable in the processor. Blanks are significant in the character string following the 'H.'

Hollerith constants can appear only in a DATA statement and in the argument list of a CALL statement.

C.2.1 Hollerith Constants In DATA Statements

A Hollerith constant may appear in the 'clist' of a DATA statement; the corresponding entity in 'nlist' must have type integer, real, or logical.

For an entity of type integer, real, or logical, the number of characters 'n' in the corresponding Hollerith constant must be less than or equal to 'g,' where 'g' is the length of the storage unit of the entity. If 'n' is less than 'g,' the entity is initialized with the 'n' Hollerith characters extended on the right with 'g-n' blank characters.

Each Hollerith constant initializes exactly one variable or array element.

C.2.2 Hollerith Constants In CALL Statements

An actual argument in a CALL statement can be a Hollerith constant, so long as the corresponding dummy argument has type integer, real, or logical. This is an exception to the rule that actual and dummy arguments must agree in type.

C.3 Hollerith Format Specification

A format specification may be an array name of type integer, real, or logical. In this case, the leftmost characters of the specified entity must contain Hollerith data constituting a legal format specification. Blank characters may precede the format specification and data may follow the right parenthesis ending the specification with no effect.

A Hollerith format specification must not contain an apostrophe edit descriptor or an 'H' edit descriptor.

C.4 'A' Editing Of Hollerith Data

The 'Aw' edit descriptor can be used with Hollerith data if the corresponding I/O list item has type integer, real, or logical.

Editing is as described for 'A' editing (section 6.4.3.1.1) of character data, except that 'n' is the maximum number of characters that can be stored in the storage unit of the list item.

APPENDIX D
EXTENSIONS TO ANSI FORTRAN

This appendix lists differences between FORTRAN-80 and ANSI FORTRAN 77. Some of these are extensions to the FORTRAN 77 subset which are included in the FORTRAN 77 full language. Other extensions go beyond both versions of the ANSI standard. In two cases, the differences merely represent a tighter definition of language semantics in FORTRAN-80 than in the ANSI standard.

Differences between FORTRAN-80 and 1966 ANSI FORTRAN are summarized at the end of this appendix.

D.1 Standard Extensions To 1977 Subset

The following is a list of FORTRAN-80 extensions to the FORTRAN 77 subset that are found in the full language.

1. Arrays with seven dimensions;
2. The logical operators .EQV. and .NEQV.;
3. The PRINT statement;
4. The BLOCK DATA statement and BLOCK DATA subprograms;
5. Integer expressions in computed GO TO and DO statements;
6. Full input/output capability of FORTRAN 77, except for the INQUIRE statement;
7. List-directed formatting.

D.2 Nonstandard Extensions To 1977 FORTRAN

The following is a list of FORTRAN-80 extensions to the FORTRAN 77 subset that are *not* found in the full language.

1. Binary, octal and hexadecimal base integer constants;
2. Integers with lengths other than the standard length (that is, lengths of one and two bytes as well as four bytes);
3. Logical items with lengths other than the standard length (that is, one and two bytes as well as four bytes);
4. Bitwise Boolean operations on bit strings under the guise of integer values;
5. Hollerith data type constants;
6. Implicit extension of the length of an integer or logical expression to the length of the left-hand side in an assignment statement;
7. Hollerith format specifications in integer, logical, and real arrays;
8. A format descriptor ($) to suppress carriage return on a terminal output device at end of record;
9. Mixtures of type and length within a memory sequence (partial association of numbers in memory);
10. CARRIAGE specifier in OPEN statement for interpreting the first character of a record;
11. B and Z (binary and hexadecimal) edit descriptors.

D.3 More Specific Semantics Than 1977 FORTRAN

In the following areas, the definition of FORTRAN-80 is more explicit than the ANSI standard.

1. The character set and its collating sequence include the ASCII character set.
2. The standard length for real, integer, and logical type data is four bytes, but a particular processor can allow a different default size to be specified.

D.4 Differences From 1966 FORTRAN

The following lists summarize differences between 1966 ANSI FORTRAN and FORTRAN-80. Most differences represent additions to 1966 FORTRAN (except where indicated by an asterisk).

Data types:

Character constants, variables, and arrays
*No double precision constants, variables, arrays, or format specifiers
*No complex constants, variables, arrays, or format specifiers
Binary, octal, and hexadecimal notations for integer constants

Statements:

PROGRAM statement
BLOCK DATA statement with a subprogram name
END statement with a label
IMPLICIT statement
INTRINSIC statement
SAVE statement
Block IF THEN, ELSE IF, ELSE, END IF
PRINT statement
OPEN and CLOSE statements
CHARACTER type statement
INTEGER and LOGICAL type statements with length specifications

Input/Output:

*No G edit descriptor
Control information list in READ, WRITE, BACKSPACE, ENDFILE, and REWIND statements
Asterisk as a unit identifier; character arrays and variables as internal units; integer expressions for external units
General expressions in WRITE, PRINT
List-directed I/O (asterisk as format)
Character expression as format; Hollerith value in arithmetic array as format identifier
Edit descriptors BZ, BN, Ew.dEe, $, Bw, Zw
General integer expressions in implied DOs

Expressions

Bit-wise Boolean operations on integers
.NEQV. and .EQV. logical operators
Character expressions in assignments, output, relationals, and procedure arguments

Miscellaneous extensions:

- Seven-dimensional arrays
- Unsubscripted array names in DATA and EQUIVALENCE statements
- Optional commas in COMMON, DO, assigned GOTO, and computed GOTO statements
- Character constant in a PAUSE or STOP statement
- BLOCK DATA subprogram names in EXTERNAL statement
- Integer expressions in a computed GOTO
- Optional label in an assigned GOTO
- Integer expressions in a DO statement
- Asterisk in column 1 to identify a comment line

APPENDIX E
ASCII CODES

The following table lists the ASCII characters representable on Intel processors and their collating sequence.

ASCII CODES

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)	GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)	GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	–	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	32	]	5D
BS	08	3	33	^ (↑)	5E
HT	09	4	34	–(←)	5F
LF	0A	5	35	`	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
SO	0E	9	39	d	64
SI	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC1 (X-ON)	11	<	3C	g	67
DC2 (TAPE)	12	=	3D	h	68
DC3 (X-OFF)	13	>	3E	i	69
DC4 (TAPE)	14	?	3F	j	6A
NAK	15	@	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
SP	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B
&	26	Q	51	|	7C
'	27	R	52	} (ALT MODE)	7D
(	28	S	53	~	7E
)	29	T	54	DEL (RUB OUT)	7F
*	2A	U	55		

APPENDIX F
8080/8085 PROCESSOR DEPENDENCIES

Throughout this manual, aspects of the FORTRAN language have been said to be 'processor dependent' or 'compiler dependent.' This appendix summarizes the limitations and extensions to the FORTRAN language assumed by the 8080/8085 processors and compiler. See the *ISIS-II FORTRAN-80 Compiler Operator's Manual* for details.

F.1 Processor Limitations On Language

Most limitations imposed on the FORTRAN language are related to data lengths and the permissible range of data values. The following indicates the range of values possible for a given data length.

Length	Value Range
INTEGER*1	−128 to +127
INTEGER*2	−32768 to +32767
INTEGER*4	−32768 to +32767
LOGICAL*1	.TRUE. or .FALSE.
LOGICAL*2	.TRUE. or .FALSE.
LOGICAL*4	.TRUE. or .FALSE.
REAL	Approximately −3.37E+38 to +3.37E+38. The handling of magnitudes less than 1.17E−38 is not defined.

If no length is specified, the compiler defaults are INTEGER*2 and LOGICAL*1.

The maximum field width, 'w,' in the Fw.d, Ew.d, Iw, and Lw edit descriptors of the FORMAT statement is 32,767.

The length and interpretation of integer expression values is determined as follows:

- Addition, subtraction, multiplication, division, or exponentation is performed modulo 256 for two INTEGER*1 operands and modulo 65536 otherwise.
- Assignment is performed modulo 256 if the variable whose value is being assigned has type INTEGER*1 and modulo 65536 otherwise.
- The length of the value of integer expressions used as actual arguments (but which are not variables or array elements) is at least the default length of an integer variable.
- Subscript expression values are taken modulo 2**16.

In all of the cases listed above, overflow is ignored.

F.2 Compiler Extensions

The ISIS-II FORTRAN-80 compiler provides a number of features in addition to those defined as part of the FORTRAN language. Some of these features are provided by 'compiler controls.' This appendix mentions only those compiler controls that affect interpretation of FORTRAN source code. Controls affecting compiler output are not included here, but can be found in the compiler operator's manual.

F.2.1 Lowercase Letters

Except within Hollerith and character constants, a lowercase letter is considered to be identical to its corresponding uppercase letter.

F.2.2 Port Input/Output

The compiler provides two intrinsic subroutines for handling input/output through 8080/8085 I/O ports. When these subroutines are called, they generate 8080 IN and OUT instructions.

The form of the subroutine calls is

```
CALL INPUT (port, var)
CALL OUTPUT (port, exp)
```

where

port	is an integer constant in the range $0 \leq port \leq 255$
var	is an integer variable
exp	is an integer expression

The value read or written through the specified port is always a single-byte integer (INTEGER*1).

Examples:

```
CALL INPUT(1, TEST1)
CALL OUTPUT(2, 100)
```

F.2.3 Reentrant Procedures

External procedures can be made reentrant by setting the REENTRANT compiler control. Reentrant procedures can call themselves directly or indirectly. Local variables are allocated in stack storage when the procedure is entered, rather than being statically allocated. Local variables and arrays must not be initialized by DATA statements in reentrant procedures.

The REENTRANT control precedes the entire program and is coded in the form

```
$REENTRANT
```

where the '$' must be in column 1.

F.2.4 Free-form Line Format

Normally, FORTRAN source file lines must be in the standard line format. To simplify entering FORTRAN programs through a console terminal, however, the FORTRAN-80 compiler allows 'free-form' lines. To use this feature, simply insert the control line

```
$FREEFORM
```

into the program before the first program unit in the source file

If the FREEFORM compiler control is set, column 1 is interpreted as follows:

Column 1	Meaning
C or *	Comment line (same as standard)
0-9	Label followed by statement
Space or TAB	Unlabeled initial line of statement
&	Continuation line of statement
$	Compiler control line

Note in this format that a statement label, if present must begin in column 1. Statements can be written in columns 2-72. They can begin in column 1 if the first letter of the statement is not a 'C.'

F.2.5 Interpretation of DO Statements

The 1966 ANSI FORTRAN standard states that all DO loops must be executed at least once. The 1977 ANS standard allows zero iterations, if so specified by the values of the initial and terminal expressions ('e1' and 'e2' in the DO statement format). The preferred interpretation can be specified by choosing either the DO66 compiler control or the DO77 compiler control in the form

```
$DO77
```

If neither is specified, DO77 is assumed by the compiler. If specified, this control must precede all FORTRAN code.

F.2.6 Default Data Lengths

The STORAGE compiler control can be used to specify the default lengths (in bytes) to be used for integer or logical variables, array elements, and constants. The default can still be overriden by INTEGER or LOGICAL type statements or, in the case of integer constants, by the number of digits in an explicit number base specification. This compiler control is coded in the form

```
$STORAGE(INTEGER*length, LOGICAL*length)
```

where

length can be 1, 2, or 4

The '$' must appear in column one and the control must precede all program units in the source file.

If no STORAGE control is specified, the compiler assumes the following defaults:

```
$STORAGE(INTEGER*2,LOGICAL*1)
```

These defaults do not conform to the ANSI standard memory allocation. To be totally ANSI compatible, specify

```
$STORAGE(INTEGER*4,LOGICAL*4)
```

F.2.7 Including Source Files

Specified files can be included in a FORTRAN source file using the INCLUDE compiler control. This control causes subsequent source code to be input from the specified 'file' until an end-of-file is reached. At end-of-file, input resumes from the file being processed when the INCLUDE was encountered.

The included file may itself contain INCLUDE controls, up to a total of six files. An included file cannot contain an END statement, however. An INCLUDE control must be the rightmost control when specified in a list of controls.

F.2.8 RECL Specification For Sequential Files

To simplify terminal I/O, the FORTRAN-80 compiler allows both ACCESS='SEQUENTIAL' and 'RECL=reclen' to be specified in the same OPEN statement. In this case, lines (records) shorter than 'reclen' are automatically extended with blanks.

F.2.9 Flexibility In Standard Restrictions

The ANSI FORTRAN 77 standard prohibits certain constructions that cannot be checked (or are uneconomical to check) by the compiler, or that cannot be implemented by other processors. Although the FORTRAN-80 compiler generally follows the standard in prohibiting these constructions, it does allow certain meaningful constructions even though they are nonstandard. While this affords the programmer some additional flexibility, be aware that future compilers may implement checks in these areas.

F.2.9.1 Association of Memory Locations. Character, logical, and numerical items can be freely declared within the same common block and can be equivalenced. In particular, the compiler does not check whether character variables of different lengths are associated.

F.2.9.2 Partially Initialized Arrays. The DATA statement can be used to initialize arrays partially (starting at the first element). If the 'nlist' in the DATA statement format contains several unsubscripted array names, initialization begins with the first array and continues until all items in 'clist' have been used.

F.2.9.3 Transfers Into An IF Block. The 8080/8085 FORTRAN compiler does not check the formal restriction against transfers into an IF, ELSE IF, or ELSE block.

F.3 Unit Preconnection

The UNIT run-time control is used to preconnect units to a program so they need not be connected by the OPEN statement. This control is specified when the program is loaded and has the form

```
UNIT n = device
```

where 'n' is in the range 0-255 and 'device' is any device recognized by ISIS-II.

Example:

```
-:F1:MYPROG UNIT4 = :LP:, UNIT 5 = :F0:SYSIN
```

INDEX

The page numbers shown in italics in this index denote primary references.

NOTES

NOTES

intel® SOFTWARE PROBLEM REPORT

SUBMITTED BY:

Name _____

Company _____

Address _____

_____

Phone _____ Date _____

FOR INTERNAL USE ONLY

No. Fix Date

Date Vers/System

Notes

CHECK ONE ITEM IN EACH CATEGORY

Product
- ☐ Software
- ☐ Manual

Product Type
- ☐ Monitor
- ☐ Assembler
- ☐ Compiler
- ☐ Simulator
- ☐ Editor
- ☐ Utility
- ☐ _____

Machine Line
- ☐ 4004/4040
- ☐ 8008
- ☐ 8080
- ☐ 3000
- ☐ _____

System
- ☐ Intellec
- ☐ Timeshare Co.
  _____
- ☐ In-House Computer
  _____

Exact Product/Manual Name _____

Version Number (If not known, give date of receipt) _____

PROBLEM:

REPLY:

PROBLEM DOCUMENTATION ATTACHED IS:

- ☐ Output Listing
- ☐ Program Listing
- ☐ Paper Tape Program Sourc
- ☐ _____

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY IF MAILED
IN THE U.S.

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 621, SANTA CLARA, CA

Postage will be paid by Addressee:

Intel Corporation
Attn: Literature Department
3065 Bowers Avenue
Santa Clara, California 95051

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____
TITLE _____
COMPANY NAME/DEPARTMENT _____
ADDRESS _____
CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

:'D LIKE YOUR COMMENTS . . .

is document is one of a series describing Intel software products. Your comments on the back of this m will help us produce better software and manuals. Each reply will be carefully reviewed by the respon-le person. All comments and suggestions become the property of Intel Corporation.

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 621, SANTA CLARA, CA

Postage will be paid by Addressee:

Intel Corporation
Attn: Literature Department
3065 Bowers Avenue
Santa Clara, California 95051

NO POSTAGE
NECESSARY IF MAILED
IN THE U.S.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

Printed in U.S.A.