Personal Computer
Computer Language
Series

FORTRAN Compiler

by Microsoft

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

CONTENTS

Preface

This is a reference manual for the IBM Personal Computer Fortran language.

You should have some prior knowledge of some dialect of Fortran. This manual is not a tutorial; rather, each section fully explains one part of the Fortran language. The manual is organized as follows:

- **Chapter 1.** "Introduction," introduces you to notational conventions, Fortran program structures, data types, expressions, and Fortran names.

- **Chapter 2.** "Compiling your Fortran Program," describes the mechanics of compiling, linking, and executing your Fortran programs.

- **Chapter 3.** "Compiler Metacommands," describes the compiler metacommands that process the Fortran source text.

- **Chapter 4.** "Statements," describes the control, program function and subroutine, I/O, and specification statements. In addition, DATA statements are described.

- **Chapter 5.** "I/O System," describes the Fortran I/O system, including the basic Fortran I/O concepts, and the FORMAT statement.

- **Chapter 6.** "Intrinsic Functions," describes the intrinsic functions available for use in a Fortran program.

- **Appendix A.** "Messages," provides a listing of messages that the computer may send you.

- **Appendix B.** "IBM Fortran and ANSI Fortran Differences" describes how IBM Fortran differs from the standard subset language.

- **Appendix C.** "The Linker (LINK) Program" provides information so you can link your program.

- **Appendix D.** "Linking Object Modules" describes how to link object modules with various compiler packages.

- **Appendix E.** "A Sample Session" shows an example of how to enter, execute, correct, reexecute, and run an IBM Personal Computer Fortran program.

CHAPTER 1. INTRODUCTION

Contents

Your IBM Personal Computer Fortran conforms to the Standard ANSI X3.9-1978 (FORTRAN 77) at the subset level and is referred to as the IBM Fortran in this manual. It also includes some features from the full level of ANSI X3.9-1978.

You will find that your IBM Fortran has also been enhanced to ease the conversion of existing Fortran 66 programs. For example, Fortran 66 semantics for DO loops are a compiler option.

The IBM Personal Computer Linker allows you to combine Fortran object modules with those of other languages, IBM Personal Computer MACRO Assembler and IBM Personal Computer Pascal Compiler, to facilitate writing applications that need different languages for different parts of your program.

The IBM Fortran consists of the Fortran compiler and a library of object modules that make up the Fortran run-time library. The first step in creating an executable Fortran program is compiling its (one or more) source modules. The object modules that result are then linked with the Fortran run-time library, producing a run file that can be invoked from the IBM Personal Computer DOS.

The object modules and libraries to be linked can include the output of the IBM Personal Computer MACRO Assembler and the IBM Personal Computer Pascal compiler, as well as the output of the IBM Fortran compiler.

Notational Conventions

The notational conventions used in this manual are as follows:

- Uppercase letters and special characters are entered in programs as shown. (For example, CONTINUE.)

- Lowercase italic letters and words are items which you should provide in actual statements described in the text. Once a lowercase item is defined, it retains its meaning for the entire discussion of the program. (For example, PROGRAM *pname*.)

For example, uppercase and lowercase in formats that describe editing of integers are denoted I*w*, where *w* is a nonzero, unsigned integer constant.

Thus, in an actual statement, a program might contain I3 or I44. The formats that describe editing of real numbers are F*w.d*, where *d* is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

- Brackets indicate optional items. For example, A [*w*] indicates that either A or A12 is valid (as a means of specifying a character format).

- The ellipsis (. . .) indicates that the optional item preceding the ellipsis may appear more than once.

For example, the computed GOTO statement described below indicates that the syntactic items denoted by *s*'s with commas separating them may be repeated any number of times:

GOTO (*s* [,*s*] . . .) [,] *i*

- Blanks normally have no significance in the description of Fortran statements. The general rules for blanks described in "Fortran Program Structure" in this chapter govern the interpretation of blanks.

Fortran Program Structure

Character Set

A Fortran source program consists of a sequence of characters, consisting of:

- Letters: the 52 uppercase and lowercase letters A through Z.
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- Special characters: the remaining printable characters of the ASCII character set.

Fortran interprets lowercase letters as uppercase letters in all contexts except in character constants and Hollerith fields. Thus, the following user-defined names are all indistinguishable to the IBM Fortran system:

```
ABCDE  abcde  AbCdE  aBcDe
```

The collating sequence for the IBM Fortran character set is the ASCII sequence.

Lines

A Fortran source program can also be considered a sequence of lines. Only the first 72 characters in a line are treated as significant by the compiler. The compiler ignores all characters after 72.

If a line is shorter than 72 characters, the compiler does treat its length as significant (for an illustration of this, "Character Expressions" later in this chapter, describes character constants).

Columns

The characters in a given line fall into columns. For example, the first character is in column 1, the second in column 2, and so forth.

A tab character may be placed in columns 1-6 of a line which causes the next character to be interpreted as being in column 7. This tab will be expanded to the appropriate number of blanks in the listing file. All other tabs are passed as is to the listing file.

The column in which a character resides is significant in Fortran:

Columns	Purpose
1-5	Statement labels and comment indicators
6	Continuation indicators
7-72	Source statements

Initial Lines

An initial line is any line that contains a blank or a 0 in column 6 and is not a comment line or a metacommand line. The first five columns of the line must either be all blank or contain a label. With the exception of the statement following a logical IF, Fortran statements begin with an initial line.

Note: A zero (0) in the continuation column on an initial line is used to improve the readability when continuation lines are used after the initial line.

Example

```
0IF  ((A.LT.0).AND.(B.LT.0)
1    .AND.(C.LT.0))
2    THEN
```

Blanks

The blank character, with the exceptions noted below, is not significant in a Fortran source program and may be used to improve the readability of Fortran programs. The exceptions are:

- Blanks within string constants
- Blanks within Hollerith fields
- A blank in column 6 distinguishes initial lines from continuation lines

Comment Lines

Comment lines do not affect the execution of the Fortran program in any way.

A line is treated as a comment if any one of the following conditions is met:

- A "C" (or "c") in column 1.
- An "*" in column 1.
- The line contains all blanks.

Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line.

> **Note:** Extra blank lines at the end of a Fortran program result in a compile-time error since Fortran interprets them as comment lines although they are not followed by an initial line.

Labels

A statement label is a sequence of from one to five digits that are unique in each program unit. At least one digit must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant.

Continuation Lines

A continuation line is any line that is not a comment line or a metacommand line that contains any character in column 6 other than a blank or a 0. The first five columns of a continuation line must be blanks.

A continuation line gives you more room to write a statement. If the statement will not fit on a single initial line, it may be extended to include up to nine continuation lines.

Statements

A Fortran statement consists of an initial line and up to nine continuation lines. The characters of the statement are up to 660 characters found in columns 7 through 72 of these lines. The END statement must be wholly written on an initial line and no other statement may have an initial line that appears to be an END statement.

Program Units

Subprograms and main programs are referred to as program units.

A subprogram begins with either a SUBROUTINE or a FUNCTION statement and ends with an END statement.

A main program optionally begins with a PROGRAM statement.

The Fortran language enforces a certain ordering among statements and lines that make up a Fortran compilation. In general, a compilation consists of, at most, one main program and zero or more subprograms (see Appendix D for more information on compilation of units and subroutines). The rules for ordering statements appear below.

Main Program and Subprogram

A main program begins with a PROGRAM statement, or any statement other than SUBROUTINE or FUNCTION statement, and ends with an END statement.

Statement Ordering

Within a program unit, whether a main program or a subprogram, statements must appear in an order consistent with the following rules:

1. A SUBROUTINE or FUNCTION statement, or PROGRAM statement if present, must appear as the first statement of the program unit.

2. FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement if present.

3. All specification statements must precede all DATA statements, statement function statements, and executable statements.

4. Within the specification statements, the IMPLICIT statement must precede all other specification statements.

5. All DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

6. All statement function statements must precede all executable statements.

Order of Statements within Program Unit

This chart is interpreted as follows:

- Statements shown above or below other statements must appear in the designated order.
- Comment lines or FORMAT statements may be interspersed with other statements that appear across from them.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="2">PROGRAM, FUNCTION, or SUBROUTINE 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>

Data Types

There are four basic data types in IBM Fortran:

1) Integer

2) Real

3) Logical

4) Character

This section describes the properties of, the range of values for, and the form of constants for each type.

The storage requirements of IBM Fortran data types for unformatted files and random access storage are:

Type	Storage (bytes)
LOGICAL	2 or 4
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4
INTEGER*2	2
INTEGER*4	4
CHARACTER	1
CHARACTER*n	n
REAL	4
REAL*4	4

Integer

The *integer* data type is a subset of the negative and positive numbers. An integer value is an exact representation of the corresponding integer. An integer variable occupies two or four bytes of storage.

An integer can be specified in IBM Fortran as INTEGER*2, INTEGER*4, or INTEGER. The first two specify, respectively, two- and four-byte integers. The third specifies either two- or four-byte integers, according to the setting of the $STORAGE metacommand (the default is four bytes).

A 2-byte integer can contain any value in the range -32767 to 32767. A 4-byte integer can contain any value in the range -2,147,483,647 to 2,147,483,647.

Integer constants are one or more decimal digits preceded by an optional arithmetic sign, plus (+) or minus (-). A decimal point is not allowed in an integer constant. The following are examples of integer constants:

```
      123      +123      -123      0
 00000123     32767     -32768
```

Real

The *real* data type consists of a subset of the real numbers: the single-precision real numbers. A single-precision real value is an approximation of the real number desired. A single-precision real value occupies four bytes of storage. The precision is six decimal digits.

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

```
+1.000E-2       1.E-2        1E-2
+0.01         100.0E-4     .0001E+2
```

All represent the same real number: one 100th.

The range of single-precision real values is:

```
3.0E-39 to  1.7E+38   (positive range)
1.7E+38 to -3.0E-39   (negative range)
```

A real constant consists of an optional sign followed by an integer part, a decimal point, a fraction part, and an optional exponent part. The integer and fraction parts consist of one or more decimal digits, and the decimal point is a period (.). Either the integer part or the fraction part may be omitted, but not both. Some sample basic real constants are:

```
-123.456    +123.456    123.456
-123.       +123        123.
-.456       +.456       .456
```

An exponent part consists of the letter "E" or "e" followed by an optionally signed integer constant of one or two digits. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts are:

```
E12    E-12    E+12    E0
```

Logical

The *logical* data type consists of the two logical values true and false. A logical variable occupies two or four bytes of storage.

A logical can be specified in IBM Fortran as LOGICAL*2, LOGICAL*4, or LOGICAL. The first two specify, respectively, two- and four-byte logicals. The third specifies either two- or four-byte logicals according to the setting of the $STORAGE metacommand (default is four bytes).

There are only two logical constants, .TRUE. and .FALSE. which represent the two corresponding logical values. The internal representation of .FALSE. is a word of all zeros (0's), and the internal representation of .TRUE. is a word of all zeros (0's) with a 1 in the least significant bit. If a logical variable contains any other bit values, its logical meaning is undefined.

The significance of a logical variable is unaffected by the $STORAGE metacommand, which is present primarily to allow compatibility with the ANSI requirement that LOGICAL, REAL, and INTEGER variables be the same size.

Character

The *character* data type is a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 127 characters.

A character variable occupies one byte of storage for each character in the sequence, plus one byte if the length is odd. Character variables are always aligned on word boundaries. The blank character is permitted in a character value and is significant.

A character constant is one or more characters enclosed by a pair of apostrophes. Blank characters are permitted in character constants. Each character counts as one. An apostrophe within a character constant is represented by two consecutive apostrophes without blanks between them. The length of a character constant is equal to the number of characters between the apostrophes, with double apostrophes counting as a single apostrophe character. Some sample character constants are:

```
'A'  ' '          'Help'
'A very long CHARACTER constant'     ''''
```

The last example, '''', represents a single apostrophe, '.

Fortran permits source lines of up to 72 columns. Shorter lines are padded to column 72 with blanks; therefore, when a character constant extends across a line boundary, its value is assigned as if the line was padded with blanks to column 72 and placed before the continuation line. Thus, the character constant:

```
200   CH = 'ABC
     XDEF'
```

has a length of 63.

Expressions

Fortran has four classes of expressions:

1) Arithmetic

2) Character

3) Relational

4) Logical

Arithmetic Expressions

An arithmetic expression produces a value that is either of type integer or real. The simplest forms of arithmetic expressions are:

- Unsigned integer or real constant
- Integer or real variable reference
- Integer or real array element reference
- Integer or real function reference

The value of a variable reference or array element reference must be defined for it to appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the above simple forms using parentheses and the arithmetic operators as follows:

Arithmetic Operators

Operator	Operation	Precedence
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	Intermediate
-	Subtraction or Negation	Lowest
+	Addition or Identity	Lowest

All of the operators are binary operators, appearing between their arithmetic expression operands. The + and - may also be unary, preceding their operand.

Operations of equal precedence are left-associative, except exponentiation, which is right-associative. Thus,

```
A/B*C
```

is the same as:

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

and:

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

is the same as:

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

Arithmetic expressions can be formed in the usual mathematical sense, as in most programming languages, except that Fortran prohibits two operators from appearing consecutively.

Thus,

```
A**-B
```

is prohibited, although:

```
A**(-B)
```

is permissible. Note that unary minus is also of lowest precedence so that:

```
-A*B
```

is interpreted as:

```
-(A*B)
```

Parentheses may be used in an expression to control the associativity and the order of operation evaluation.

Integer Division

The division of two integers results in a value that is the mathematical quotient of the two values, truncated to an integer. Thus, 7/3 evaluates to 2, (–7)/3 evaluates to –2, 9/10 evaluates to 0, and 9/(–10) evaluates to 0.

Type Conversions and Result Types

When all operands of an arithmetic expression are of the same type, the value produced by the expression is also of that type. When the operands are of different data types, the value produced by the expression is of a data type determined by the rank as follows:

Arithmetic Data Type	Rank
REAL	Highest
INTEGER*4	Intermediate
INTEGER*2	Lowest

When an operation has two arithmetic operands of different data types, the value of the data type produced is the data type of the highest-ranked operand. For example, an operation on an integer and a real element produces a value of data type real.

The data type of an expression is the data type of the result of the last operation performed in evaluating the expression. The data types of operations are classified as either INTEGER*2, INTEGER*4, or REAL.

Integer operations are performed on integer operands only. A fraction resulting from division is truncated in integer arithmetic, not rounded. Thus the following example evaluates to 0, not 1.

```
1/4 + 1/4 + 1/4 + 1/4
```

> **Note:** Memory for the type INTEGER (without the *2 or *4 extensions) is dependent on the usage of the $STORAGE metacommand. See Chapter 3 for details.

Real operations are performed on real operands or combinations of real and integer operands. When an operation has a real and an integer operand, the integer operand is first converted to a real data type by giving each a fractional part equal to zero, real arithmetic is used to evaluate the expression.

For example:

```
A = N/B
```

N is converted to a real data type and real division is performed on N and B.

When an expression contains mixed data types, the integer and real operations performed are evaluated in the order of precedence of the operators. For example:

```
Y = X * (I + J)
```

Integer addition is performed on I and J, the sum is changed to a real data type and real multiplication is performed on the result and X.

> **Note:** IBM Fortran does not check for integer overflow. Unpredictable results will occur if the limits of integer values are exceeded.

Character Expressions

A character expression produces a value that is of type character. The forms of character expressions are:

- Character constant
- Character variable reference
- Character array element reference
- Any character expression enclosed in parentheses

No operators result in character expressions.

Relational Expressions

Relational expressions compare the values of two arithmetic or two character expressions.

An arithmetic value may not be compared with a character value. The result of a relational expression is of type logical.

Relational expressions can use any of the operators shown below to compare values.

Relational Operators

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

All of the operators are binary characters, appearing between their operands. For example:

```
A  .GT.  B
X  .EQ.  7
```

The above examples are valid expressions. There is no relative precedence or associativity among the relational operands; therefore,

```
A .LT. B .NE. C
```

is not valid. The above example violates the type rules for operands. Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have one operand of type integer and one of type real. In this case, the integer operand is converted to type real before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating sequence. If operands of unequal length are compared, the shorter operand is considered as if it were extended to the length of the longer operand by the addition of blanks.

Logical Expressions

A logical expression produces a value that is of type logical. The simplest forms of logical expressions are:

- Logical constant
- Logical variable reference
- Logical array element reference
- Logical function reference
- Relational expression

Other logical expressions are built up from the above simple forms using parentheses and the logical operators as shown in the following chart:

Logical Operators

Operator	Operation	Precedence
.NOT.	Negation	Highest
.AND.	Conjunction	Intermediate
.OR.	Inclusive disjunction	Lowest

The AND and OR operators are binary operators, appearing between their logical expression operands. The NOT operator is unary, preceding its operand. Operations of equal precedence are left-associative, for example:

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

is equivalent to:

```
(A .AND. B) .AND. C
```

As an example of the precedence rules:

```
.NOT. A .OR. B .AND. C
```

is interpreted the same as:

```
(.NOT. A) .OR. (B .AND. C)
```

Two NOT operators cannot be adjacent to each other, although:

```
A .AND. .NOT. B
```

is an example of an allowable expression with two adjacent operators.

The meaning of the logical operators is their standard mathematical semantics, with .OR. being "nonexclusive," that is,

```
.TRUE.  .OR.  .TRUE.
```

evaluates to the value:

```
.TRUE.
```

Array Element Name

The array element name is used to reference one element of an array.

arr(*sub* [,*sub*]. . .)

The *arr* entry is the name of an array. The *sub* entry is a subscript expression.

```
C     ASSIGN THE 4TH ELEMENT OF
C     ARRAY A THE VALUE 3.8
      A(4,1,1)=3.8
```

A subscript expression is an integer expression used in selecting a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between 1 and the upper bound for the dimension it represents.

Function Reference

A function reference may appear in an arithmetic or logical expression. Execution of a function reference causes the function to be evaluated, and the resulting value is used as an operand in the containing expression.

fname ([*arg* [, *arg*]] . . .)

The *fname* entry is the user- or system-defined name of an external, intrinsic, or statement function.

The *arg* entry is an actual argument. An actual argument can be an arithmetic expression, an array, or a user- or system-defined function or subroutine. Here are some examples of function references in expressions:

```
1+SIN(.5)

A+B+USERFN(9)

.TRUE..AND.BITFN(3)

VAR1/XYZ(3,A)
```

See Chapter 4 for more information pertaining to formal parameters. The number of actual arguments must be the same as in the definition of the function, and the corresponding types must agree.

Precedence of Expressions

When arithmetic, relational, and logical operators appear in the same expression, their relative precedence is as follows:

Relative Precedence of Operator Classes

Operator	Precedence
Arithmetic	Highest
Relational	Intermediate
Logical	Lowest

Evaluation Rules and Restrictions for Expressions

Any variable, array element, or function referenced in an expression must be defined at the time of the reference. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Arithmetic operations which are not mathematically meaningful; such as dividing by 0, are prohibited. Other prohibited operations are raising a 0-valued operand to a 0 or negative power and raising a negative-valued operand to a power of type real.

Fortran Names

A Fortran name is used to denote a variable, array, function, or subroutine.

A Fortran name consists of an initial alphabetic character followed by a sequence of up to five alphanumeric characters. Blanks may appear within a Fortran name, but have no significance.

Any valid sequence of characters can be used for any Fortran name. There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords by the Fortran compiler are not confused with user-defined names.

The compiler recognizes keywords by their context and in no way restricts the use of user-defined names. Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error (as long as it conforms to the rules that all arrays must obey); however, this is not recommended.

Scope of Fortran Names

The scope of a name is the range of statements in which that name is known, or can be referenced, within a Fortran program unit. In general, the scope of a name is either global or local, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope can be used in more than one program unit (a subroutine, function, or the main program) and still refer to the same entity. In fact, names with global scope can only be used as global names in a single, consistent manner within the same program. All subroutine, function subprogram, and common names, as well as the program name, have global scope. Therefore, a function subprogram cannot have the same name as a subroutine subprogram or a common data area. Similarly, no two function subprograms in the same program can have the same name.

Even though there are no reserved names, in IBM Fortran, using certain names will conflict with those names in the library and will cause link errors. The following names should *not* be used as global names:

ABS	COS	ICLRER	MOVESL	SMULOK
ACOS	COSH	IDIM2	MOVESR	SQRT
AINT	DIM4	IDIM4	NINT2	TAN
ALOG	EXP	IGETER	NINT4	TANH
ALOG10	FILLC	ISIGN2	PUTCH	TNSR
AMOD4	FILLSC	ISIGN4	PUTSTR	UADDOK
ANINT	FTRANS	LOCKED	REAC	UM46OK
ASIN	GETCH	MOD2	SADDOK	UMULOK
ATAN	GETSTR	MOD4	SIGN4	UNLOCK
ATAN2	IABS2	MOVEL	SIN	UTLR
CNVR	IABS4	MOVER	SINH	

> **Note:** In addition to the above, any six character names ending with QQ should not be used.

The names of variables, arrays, formal parameters, and statement functions all have local scope.

A name with local scope is only visible (known) within a single program unit. A name with a local scope can be used in another program unit with a different meaning, or with a similar meaning, but the name is in no way required to have similar meanings in a different scope.

One exception to rules about scope is the name given to common data blocks. It is possible to refer to a "global" common block name in the same program unit in which an identical "local" name appears. This is permitted because common block names are always enclosed in slashes, such as /**JUDY**/, and can, therefore, be distinguished from ordinary names by the compiler.

Another exception to the rules about scope relates to parameters of statement functions. The scope of statement function parameters is limited to the single statement forming that statement function. Any other use of those names within that statement function is not permitted, but any other use outside that statement function is permitted.

Undeclared Fortran Names

When a user name that has not appeared before is encountered in an executable statement, the compiler classifies that name from the context of its use. If the name is used in the context of a variable, the compiler creates an entry into the symbol table for a variable of that name. Its type is inferred from the first letter of its name. Variables beginning with the letters I, J, K, L, M, or N are considered integers, while all others are considered reals; these defaults can be overridden by an IMPLICIT statement (see Chapter 4).

If an undeclared name is used in the context of a function reference, a symbol table entry is created for a function of that name. Its type will be inferred in the same manner as for a variable.

Similarly, a symbol table entry is created for a newly encountered name that is the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

CHAPTER 2. COMPILING A FORTRAN PROGRAM

Contents

What You Need

To successfully compile Fortran programs on your IBM Personal Computer, you need:

- Your Fortran package:
  - Three 5-1/4 inch master diskettes, one marked FOR1, one marked FOR2, and one marked LIBRARY
    - FOR1 should contain the file:
      - FOR1.EXE
    - FOR2 contains the file:
      - FOR2.EXE
    - LIBRARY contains the files:
      - FORTRAN.LIB
      - FORTRAN.ARF (Automatic Response File)
      - LINK.EXE
      - DOS Sample Batch Procedures
  - This manual: the *IBM Personal Computer Fortran* reference manual
- A minimum of 128K bytes of machine-resident storage
- Two diskette drives
- A printer (highly recommended)
- A display (an IBM Personal Computer Monochrome Display, a monitor, or a TV with an RF modulator)

- The *IBM Personal Computer Disk Operating System (DOS)* reference manual and its diskette

- One 5-1/4 inch diskette which we will call the *scratch* diskette

  The first time through, you will also need:

  – Three 5-1/4 inch diskettes to make backup copies of the master diskettes

Backing Up the Master Diskettes

We strongly recommend that you back up your Fortran master diskettes as soon as possible by making copies of FOR1, FOR2, and LIBRARY. We also recommend that you use these copies for your day-to-day operations, and put the master diskettes in a safe place.

Setting Up the Diskettes: FOR1 and FOR2

Now that you have made copies of FOR1 and FOR2, you will need to copy COMMAND.COM from the DOS diskette onto FOR1 and FOR2. This is because when FOR1 and FOR2 are loaded, they may overwrite COMMAND.COM in storage (COMMAND.COM is loaded when the system is started).

COMMAND.COM will then be automatically reloaded when either FOR1 or FOR2 is finished executing and is in drive A.

Setting Up the Diskettes: LIBRARY

Now that you have made a copy of the LIBRARY, you will need to copy COMMAND.COM from the DOS diskette onto the LIBRARY diskette. This is done for the same reason as above. The IBM Personal Computer Linker Version 1.10 on this diskette is an upward compatible version of the IBM Personal Computer Linker Version 1.00 on the DOS diskette. The linker on your LIBRARY diskette must be used to successfully run a Fortran program on the IBM Personal Computer.

In addition, it may be used in place of the Linker supplied with the IBM Personal Computer DOS Version 1.00.

See "Appendix C" for full details on the IBM Personal Computer Linker Version 1.10.

Using the EDLIN Program

You can use the Line Editor (EDLIN) which is provided as a part of the IBM Personal Computer DOS to create, change, and display *source files* or *text files*. Source files are unassembled or uncompiled programs in source language format. Text files appear in a legible format.

The EDLIN program is a line text editor that:

- Deletes, edits, inserts, and displays lines
- Searches for, deletes, replaces, and displays text
- Creates new files and saves them
- Updates old files and saves both updated and original files

The text of files created or edited by EDLIN is divided into lines of varying length, up to 253 characters-per-line.

Line numbers are dynamically generated and displayed by EDLIN during the editing process, but are not actually present in the saved file.

When you insert lines, all line numbers following the inserted text advance automatically by the number of lines inserted. When you delete lines, all line numbers following the deleted text decrease automatically by the number of lines deleted. Consequently, line numbers always go consecutively from 1 through n (the last line).

Starting the Compilation

We recommend the following sequence of steps as a general rule:

1. Format your scratch diskette. See the *IBM Personal Computer Disk Operating System (DOS)* reference manual for information about formatting.

2. Put your program onto the scratch diskette either by copying it from the diskette it is already on, or by creating a new program using the line editor (see "EDLIN" in the *IBM Personal Computer Disk Operating System (DOS)* reference manual).

3. Give your program the filename extension ".FOR", for Fortran.

You are now ready to compile your Fortran program.

Notes:

1. You may enter compiler commands using all uppercase letters, all lowercase letters, or a combination of uppercase and lowercase letters.

2. A sample program session of the following paragraphs is presented in "Appendix E."

Starting the Compiler: FOR1

FOR1 is the first pass of the compiler. FOR1 reads your source file and checks it for syntactic correctness. It generates two intermediate files which are stored on the scratch diskette in the space not occupied by your source program.

These files are called:

PASIBF.SYM – the symbol table
PASIBF.BIN – the intermediate binary code

FOR1 also creates your source listing file. If you have a printer with your system, you may print a copy of the source listing to aid you in debugging.

Use these steps for the FOR1 portion of the compilation of your program:

1. Change the default drive (in response to the A>) to B by entering:

   ```
   B:
   ```

2. Insert the scratch diskette containing your program into drive B.

3. Insert the FOR1 diskette into drive A.

4. Enter:

   ```
   A:FOR1
   ```

FOR1 will be loaded into the computer. After a short time, the compiler will display a heading and the following prompt:

```
Source filename [.FOR]:
```

Notes:

1. The name shown within the brackets is the *default* filename extension that your IBM Fortran will use if you do not choose a filename extension of your own.

2. Although your IBM Fortran will supply a default filename extension whenever you do not supply one, all extensions may be overridden by explicitly specifying the filename with the new extension.

3. The default diskette drive is the DOS default drive; it may be explicitly overridden by including the drive identification as part of the filespec. Also just the drive identification may be given when a complete default filespec exists.

4. The PASIBF files are always created on the DOS default drive.

Source filename is the name of the file in which you have stored your program. For example, assuming you responded with "myfile" to the prompt, the display shows:

```
Source filename [.FOR]: myfile
```

It is not necessary to enter the .FOR filename extension because the compiler automatically looks for .FOR. After you enter your source filename, you will see this prompt:

```
Object filename [MYFILE.OBJ]:
```

Object filename is the name you want the object (machine-readable) file to have. If you wish to have your object file stored under the name MYFILE.OBJ, you can simply press the Enter key, or you may give the file another name .**OBJ**. For our example, assume we have simply pressed the Enter key:

```
Object filename [MYFILE.OBJ]:
```

The next prompt will look like this:

```
Source listing [NUL.LST]:
```

Source listing is the name you wish to give to the file that will contain the compiled source listing. If you do *not* want a listing, press the Enter key. This will give you the default filename **NUL.LST**, which tells the compiler not to create a source listing file.

For our example, assume we do want a listing file and enter:

```
Source listing [NUL.LST]: myfile
```

> **Note**: The compiler will add the default extension and produce the listing file, MYFILE.LST.

The last prompt is:

```
Object listing [NUL.COD]:
```

Object listing is the name you wish to give to the file that will contain the disassembled object file listing. If you do not want a listing, press the Enter key. This will give you the default filename **NUL.COD**, which tells the compiler not to create a source listing file.

For our example, assume you have entered:

```
Object listing [NUL.COD]: myfile
```

The specification of the filename extension is not necessary because the compiler would have provided it as the default extension and produced the file, MYFILE.COD.

This is what the completed session would look like if you use the filenames for the above examples.

```
Source filename [.FOR]: myfile
Object filename [MYFILE.OBJ]:
Source listing [NUL.LST]: myfile
Object listing [NUL.COD]: myfile
```

As soon as you enter the last filename, the compiler begins its first pass through your program. If the program contains any syntax errors, the compiler shows the errors on the display as well as in the listing file (see "Compiler Listing" at the end of this chapter).

> **Note:** Before pressing Enter after any of these responses, you may continue the response with a comma and the answer to what would be the next prompt; you do not have to wait for that prompt. If you end any with a semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

When it has completed its first pass, the compiler displays a message with the number of errors it has found. The message will look like this if you send the source listing to a file:

```
Pass One      No Errors Detected
              10 Source Lines
```

If there were errors, they are shown on the display along with one or both of these messages:

```
Pass One       3 Errors Detected
              120 Source Lines
```

If the compiler has indeed found errors, you must locate and fix those problems in your source program and rerun FOR1 *before* you continue the compilation with FOR2.

> **Note:** If you have not copied COMMAND.COM onto your FOR1 diskette, you will now be asked to insert the DOS diskette.

Continuing the Compilation: FOR2

When your corrections (if any were necessary) are completed, and you have run the corrected program through FOR1 again, you are ready to complete the compilation.

FOR2 is the second pass of the Fortran compiler. During this second pass, the compiler reads the **.SYM** and **.BIN** files made by FOR1 and creates the two object files, **.COD** and **.OBJ**. This is the optimization pass.

FOR2 creates, writes, reads, and deletes a file called **PASIBF.TMP** on the DOS default drive (the intermediate link text) as well as reading and deleting **PASIBF.SYM** and **PASIBF.BIN**.

Some programs may compile correctly during FOR1, but may produce errors during FOR2. See Appendix A, "Messages," for these errors.

Your scratch diskette with the FOR1 files on it should still be in diskette drive B.

Use the following steps for the FOR2 portion of the compilation of your program:

1. Remove the FOR1 diskette from drive A.

2. Insert the FOR2 diskette into drive A.

3. Enter:

```
A:FOR2
```

FOR2 requires no input from you. FOR2 is loaded and after a short time starts to run. It generates the object file and listing. When the compilation is completed, FOR2 gives you a message similar to this:

```
Code Area Size = #0116   (278)
Cons Area Size = #005E   (94)
Data Area Size = #000E   (14)

Pass Two      No Errors Detected.
```

The Code Area Size is the total number of bytes taken up by your program (in our example, 278 bytes). Cons Area Size is the number of bytes taken up by the constants in your program. The Data Area Size refers to the static allocated data. This area always starts at offset #2. All three sizes are given in both hexadecimal and decimal.

If errors are detected during the second pass, see Appendix A, "Messages", in this book, and correct the errors. Then rerun FOR1 and FOR2, if necessary.

Linking

We recommend that you read the *IBM Personal Computer Disk Operating System (DOS)* reference manual for an explanation of linking. Also, see "Appendix C" in this manual.

Use the following steps for the Linker portion of the compilations of your program.

1. Remove FOR2 from drive A.

2. Insert your copy of LIBRARY diskette into drive A.

3. Enter:

   A:link

   The Linker is loaded and after a short time the Linker displays a heading and the following prompt:

   Object Modules [.OBJ]:

   Object Modules is the name of the machine-readable file(s) created by FOR2.

   As with FOR1, the **.OBJ** extension is *not* needed here. If you used our example names, you would enter:

   Object Modules [.OBJ]: myfile

   The next prompt is:

   Run File [MYFILE.EXE]:

   Run File is the name you wish to give to the file that will contain the executable (machine-readable, relocatable) code for your program. If you wish to have your *Run File* stored under the name **MYFILE.EXE**, simply press the Enter key or give the file another name, to which the IBM Personal Computer Linker will add the filename extension **.EXE** (this filename extension may not be overridden).

For our example, assume we have just pressed the Enter key:

```
Run File [MYFILE.EXE]:
```

The next prompt is:

```
List File [NUL.MAP]:
```

MAP file is the name you wish to give to the file that will contain Linker printed output. If you do not want a **MAP file**, press the Enter key. This will give you the default filename **NUL.MAP**, which tells the Linker not to create a **MAP file**.

For our example, assume we do want a MAP file and have entered:

```
Map File [NUL.MAP]: myfile
```

The Linker adds the default extension and produces the MAP File, **MYFILE.MAP**.

The next prompt is:

```
Libraries [.LIB]:
```

Libraries refers to the runtime routines needed by IBM Fortran to run your program. All these routines are included in FORTRAN.LIB. In IBM Fortran as well as IBM Personal Computer Pascal, the names of the libraries needed are supplied by the object module. In response to this prompt, you may press the Enter key:

```
Libraries [.LIB]:
```

When you link a Fortran program, the Fortran library is brought in automatically. IBM Fortran will look by default for the library on drive A.

Here is what this sequence of prompts would look like if you use our example filenames:

```
B>a:link
IBM Personal Computer Linker
Version 1.10 (C) Copyright IBM Corp 1982
Object Modules [.OBJ]: myfile
Run File [MYFILE.EXE]:
List File [NUL.MAP]: myfile
Libraries [.LIB]:
```

The linker now begins to link the program. When linking has been completed, you have the Run File named "myfile" stored on your scratch diskette in drive B. We recommend that you display the diskette directory for the scratch diskette to confirm that the run filename is there (it will have the **.EXE** filename extension). Using our example filename, you would see **MYFILE.EXE** listed in the directory.

Running Your Fortran Program

To run your program, simply enter your run filename, without the **.EXE** filename extension. For example, enter:

```
myfile
```

You may want to copy this file to another diskette once you are sure that it does what you intended it to do.

Optional FOR1 Command Lines

FOR1 can also be run using the following command line (substituting, of course, your filenames for the four files shown):

```
FOR1 Source File,Object File,Source
List,Object List;
```

When you use a command line, the FOR1 prompts described in the above example are not displayed if an entry for all four files are specified or if the command line ends with a semicolon.

If an incomplete list is given and no semicolon is used, the compiler prompts for the remaining unspecified files. Each prompt displays its default which may be accepted by pressing the Enter key, or overridden with an explicit filename or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files are defaulted without further prompting.

Certain other variations of this command line are permitted.

Examples are as follows:

1) `FOR1 module`

Source is *module*.FOR. A prompt is given, showing the default of *module*.OBJ. After the response is entered, a prompt is given showing the default of NUL.LST. After the response is given, a prompt is displayed showing the default of NUL.COD.

2) `FOR1 module;`

If the semicolon is added, no further prompts are displayed. The source of *module*.FOR is compiled; the object is produced in *module*.OBJ; no listing or object listing file is produced.

3) FOR1 module,,;

This is similar to the above example, except that the listing is produced in *module*.LST.

4) FOR1 module,,,;

This is similar to the preceding examples, except that the object listing, *module*.COD, is also produced.

5) FOR1 module,,,

Using the same example, but without the semicolon, *module*.FOR is compiled, the object is produced in *module*.OBJ, a listing is produced in *module*.LST, but a prompt is given with the default of *module*.COD.

6) FOR1 module,NUL,;

No object is produced. The listing is produced in *module*.LST. No object listing is produced, and no further prompts are displayed.

Optional FOR2 Command Line

FOR2 can be made to operate as follows:

- Look for the PASIBF files on a drive other than the default drive.
- Pause before execution to allow diskette swapping.

The command line format is:

FOR2 [*drive*] [P]

The *drive* entry is the diskette drive where the PASIBF file is stored. The *drive* entry must be in the range of the valid DOS drive identifications. If not, the P entry is implied. If the *drive* entry is present, the following message will be displayed:

```
PASIBF.SYM and PASIBF.BIN are on Drive x.
```

The P entry tells FOR2 to pause before starting to execute. If the pause entry is present, the following prompt will be displayed:

```
Press enter to begin Pass two:
```

To begin Pass two, simply press the Enter key.

Compiling Using a Batch File

See the *IBM Personal Computer Disk Operating System (DOS)* reference manual for detailed description of the Batch command facility that can be used to automatically start the compiler.

A sample Batch command to display the listing is as follows:

> **Note:** In the following example, assume drive B: has source and space for a listing, and drive A: has DOS and the compiler.

Use EDLIN to create FORC.BAT.

1. A:MODE LPT1:132
2. B:
3. PAUSE (Insert FOR1 in drive A)

4. A:FOR %1,NUL

5. TYPE %1.LST

6. ERASE %1.LST

7. PAUSE (Insert FOR2 in drive A)

8. A:FOR2

Then enter:

```
FORC module
```

Note: Several files exist on the LIBRARY diskette with **.BAT** extensions. These batch procedures are provided as commented examples of the use of the Batch command facility.

Compiling Large Programs

You may find that there is not enough space on the scratch diskette to hold all the files produced by the compiler (an out of space error message is displayed).

In this event, the tips in the following paragraphs will aid you in compiling your program:

- You can send any of the FOR1 or FOR2 output files to one of the special DOS filenames which do not correspond to a disk file. These special filenames or device names are described "Device Identification" later in this chapter.

  The following FOR1 command line sends .OBJ to NUL (useful when debugging), .COD to NUL, and .LST to CON:

  ```
  FOR1 myfile,NUL,CON,NUL;
  ```

- By swapping diskettes, you can maximize the amount of space that files can occupy. The following is the optimum configuration:

  Assume the source filename is myfile and the DOS default drive is B:. During this session, you may be prompted to reinsert your DOS diskette.

  1. Insert the FOR1 diskette into drive A.

  2. Insert a blank formatted diskette into drive B.

  3. Enter:

     A:FOR1 A:myfile,A:

  4. When the listing file prompt appears, remove the FOR1 diskette and insert into drive A the diskette with the source file called myfile. Respond to the listing file prompt with a semicolon (;).

  5. When FOR1 has completed running, remove the source file diskette from drive A and insert the FOR2 diskette.

  6. Enter:

     A:FOR2 p

  7. When FOR2 displays the prompt:

     Press ENTER to begin Pass Two:

     Remove FOR2 from drive A and insert a blank formatted diskette.

  8. Press Enter.

  9. When FOR2 has completed running, you will have the final object file on drive A.

10. Move the object file diskette from drive A to drive B.

11. You may now link the object files as previously described.

    Note: Sending the .LST or .COD files to CON or AUX would not have affected this procedure.

Very large programs can be broken down into smaller *units* or *modules* and compiled separately with FOR1 and FOR2 (see Appendix D, in this book, for information about how **program units** are constructed and compiled separately). They can then be joined together by the Linker to create a single run file.

Device Identifications

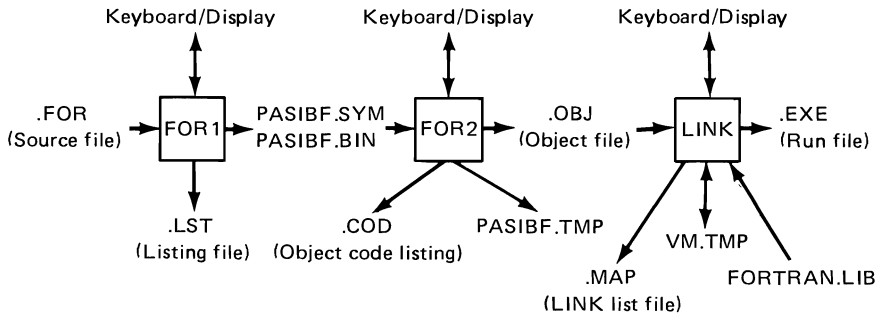
If you want to override the implied DOS default diskette drive, you can specify the device identification for your output devices.

The device names and designated identifications (IDs) are:

- Printer – LPT1, PRN
- Diskette drives – A:, B:
- Display/keyboard – CON (buffered output), USER (non-buffered)
- RS-232 (serial port) – AUX, COM1, LINE
- No output file – NUL

  Note: Suspension of the computer can occur by pressing the Ctrl and Num Lock keys. Press any character to resume. This can be useful to temporarily halt the display scrolling so it can be read, when a large amount of output is being generated.

The following diagram describes the flow of files through a Fortran compilation:



```
Student grades management program                                    Page   1
                                                                     11-09-81
                                                                     00:10:59
D Line# 1      7                        IBM Personal Computer FORTRAN Compiler V1.00
      1 $title:'Student grades management program'
      2 $storage:2
      3       PROGRAM grades
      4
      5 C Define arrays and variables
      6       DIMENSION scores(14,10),total(14)
      7       CHARACTER*5 names(14)
      8       INTEGER s
      9       CALL start(nostd,notst)
     10       CALL getgrd(nostd,notst,names,scores)
     11       CALL totit(nostd,notst,scores,total)
     12       CALL report(nostd,notst,names,scores,total)
     13       CALL rptavg(nostd,notst,scores,total)
     14       END

Name    Type         Offset P Class

NAMES   CHAR*5          618
NOSTD   INTEGER*2       702
NOTST   INTEGER*2       704
S       INTEGER*2     *****
SCORES  REAL              2
TOTAL   REAL            562
```

Compiler Listing

There are references throughout the following explanation to the IBM Fortran Compiler metacommands. Refer to Chapter 3 in this book for a complete description of the metacommands.

Every page of the Fortran source listing has a header at the top. In the upper left hand portion of the page, the first two lines contain the user's choice of program title and subtitle, set with the $TITLE and $SUBTITLE metacommands, respectively. If the $TITLE or $SUBTITLE is too long to fit on the heading line, they are both moved directly below the header.

The first three lines in the upper right hand portion of the page contain the page number, the date, and the time respectively. A new page can be started with $PAGE. The compiler name and version number appear on the first line below the header, along with the column labels. The compiler name and version number line, even if split, ends above column 72 of the input source lines. This is useful to spot lines that exceed column 72. In addition, these lines are truncated on the listing to show that the line is too long; therefore, part of the line is ignored. If the linesize is decreased to the point that the compiler name and version number can no longer fit on the line below the header, it is split in half and becomes part of the header.

The D Column Label

The D column contains the current nesting level of DO loops. It is incremented for the statement following the DO statement and is decremented on the terminal statement. If 2 or more DO loops share the same terminal statement, the inner most DO level is placed on that terminal statement. The column is empty when the DO nest level is 0. The number in this column can be used to find missing terminal statements. DO nest levels greater than 9 are represented as an asterisk (*).

The Line# Column

This column contains the listing line numbers, which are internally generated. The line number is used to identify runtime errors if $DEBUG is on.

Additional Listing Metacommands

Several other metacommands affect the listing. $LINESIZE:n and $PAGESIZE:n set the width and height; and, $LIST and $NOLIST can be used to turn the listing on or off (errors are always listed).

The metacommands themselves appear in the listing, except for $NOLIST.

```
      15 C Read in program parameters
      16        SUBROUTINE start(nstd,ntst)
      17        WRITE(*,5)
      18      5 FORMAT('0',10x,'Student Grades Management')
      19        WRITE(*,6)
      20      6 FORMAT(11x,'******* ****** **********')
      21        WRITE(*,8)
      22      8 FORMAT('0Input #Students and #Grades-Use 2I2')
      23        READ (*,10) nstd,ntst
      24     10 FORMAT(2i2)
      25        END

Name    Type         Offset P Class

NSTD    INTEGER*2         2 *
NTST    INTEGER*2         6 *
```

```
      26 C Read in student grades
      27        SUBROUTINE getgrd(nstd,ntst,names,scrbk)
      28        CHARACTER*5 names(nstd)
      29        DIMENSION   scrbk(nstd,ntst)
      30        DO 15 a=1,nstd
***** Error 137 -- integer variable expected
      31             WRITE(*,9) ntst
      32      9      FORMAT('0Name-A5  Grade-',I2,'F5.0')
      33             READ(*,20) names(a),(scrbk(a,j),j=1,ntst)
      34     20      FORMAT(a5,6f5.0)
      35     15 CONTINUE
      36        END

Name    Type         Offset P Class

A       REAL            824
```

Compiler Messages

A compilation with any errors cannot be used to generate code. Errors start with a number. Errors are listed by number in Appendix A.

Unrecoverable Errors

The compiler may find an error from which it cannot recover. In this case, it gives the message:

"? ERROR: error message"

```
D Line# 1      7                    IBM Personal Computer FORTRAN Compiler V1.00
J       INTEGER*2       864
NAMES   CHAR*5           10 *
NSTD    INTEGER*2         2 *
NTST    INTEGER*2         6 *
SCRBK   REAL             14 *


      37 C Make the totals
      38       SUBROUTINE totit(nstd,ntst,scrbk,total)
      39       DIMENSION scrbk(nstd,ntst),total(nstd)
      40       DO 10 i=1,nstd
1     41          total(i)=0
1     42          DO 10 j=1,ntst
2     43             total(i)=total(i)+scrbk(i,j)
2     44 10    CONTINUE
      45       END
```

Symbol Table

Two types of symbol tables are included in the IBM Fortran listing file, (1) a local symbol table is included after each program unit and (2) a global symbol table is included at the end of the compilation. Each symbol table includes information for each name defined in that program unit.

The type, offset, size, parameter or class may be given:

- The type is given for variables and functions and is one of the several available data types in IBM Fortran.

- The offset is the offset within an entity. If the name is a variable, the offset value is the offset within the static data area. If the name is within a common block, the offset value is the offset within the common block. If the name is a formal parameter, the offset value is the offset in the stack of the address of the actual argument. If the offset entry is filled with asterisks, the variable was defined but never referenced. The offsets are defined in bytes.

- The parameter (P) is either a blank or an asterisk. The presence of an asterisk indicates the name is a formal parameter.

- The class indicates if the name is other than a variable or if the variable is in a common block. The class can be a function, a subroutine, common, intrinsic, external, program, subroutine/function parameter, or the name of a common block surrounded by slashes.

In the global symbol table, the offset and parameter are replaced with a size entry. This is the size of the common block named in this compilation.

```
Name    Type          Offset P Class

I       INTEGER*2        880
J       INTEGER*2        884
NSTD    INTEGER*2          2 *
NTST    INTEGER*2          6 *
SCRBK   REAL              10 *
TOTAL   REAL              14 *


      46 C Report student scores
      47       SUBROUTINE report(nstd,ntst,names,scrbk,total)
      48       CHARACTER*5 names(nstd)
      49       DIMENSION scrbk(nstd,ntst),total(nstd)
      50       WRITE(*,45)
      51    45 FORMAT(13x,'Student  Test 1 Test 2 Test 3 Test 4 Test 5 Test 6')
      52       WRITE(*,50)
      53    50 FORMAT(13x,'*******  ****** ****** ****** ****** ****** ******')
      54       WRITE(*,52)
```

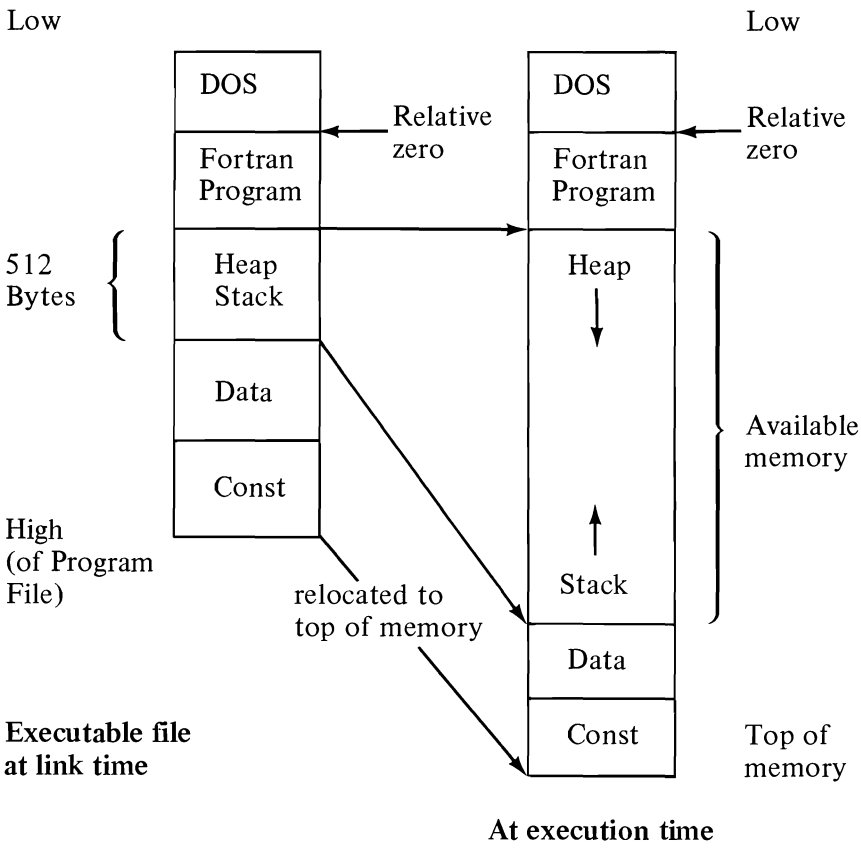
```
      55     52 FORMAT(1x)
      56        DO 55 i=1,nstd
1     57            WRITE(*,60) names(i),(scrbk(i,j),j=1,ntst)
1     58            FORMAT(15x,a5,2x,6(f6.2,1x))
***** Error 135 -- FORMAT label missing
1     59     55 CONTINUE
      60        WRITE(*,70)
      61     70 FORMAT(1h0,30x,'Student Average')
      62        WRITE(*,75)
      63     75 FORMAT(31x,'***************')
      64        WRITE(*,52)
      65        DO 80 i=1,nstd
1     66            WRITE(*,85) names(i),total(i)/ntst
1     67     85     FORMAT(32x,a5,3x,f6.2)
1     68     80 CONTINUE
      69        END
***** Error 163 -- FORMAT not found

Name    Type         Offset P Class

I       INTEGER*2      1012
J       INTEGER*2      1016
NAMES   CHAR*5           10 *
```

```
D Line# 1      7                    IBM Personal Computer FORTRAN Compiler V1.00
NSTD    INTEGER*2          2 *
NTST    INTEGER*2          6 *
SCRBK   REAL              14 *
TOTAL   REAL              18 *


      70
      71 C Report averages
      72         SUBROUTINE rptavg(nstd,ntst,scrbk,total)
      73         DIMENSION scrbk(nstd,ntst),total(nstd)
      74    110  WRITE(*,65)
      75     65  FORMAT('0',30x,'Class Average')
      76         WRITE(*,90)
      77     90  FORMAT(31x,'**************')
      78         tscrbk=0
      79         DO 40 i=1,nstd
1     80             tscrbk=tscrbk+total(i)
1     81     40  CONTINUE
      82         carg=tscrbk/(nstd*ntst)
      83         WRITE(*,95) carg
      84     95  FORMAT('0',35x,f6.2)
      85         END
```

```
Name    Type          Offset P Class

CARG    REAL            1144
I       INTEGER*2       1140
NSTD    INTEGER*2          2 *
NTST    INTEGER*2          6 *
SCRBK   REAL              10 *
TOTAL   REAL              14 *
TSCRBK  REAL            1136


Name    Type            Size    Class

GETGRD                          SUBROUTINE
GRADES                          PROGRAM
REPORT                          SUBROUTINE
RPTAVG                          SUBROUTINE
START                           SUBROUTINE
TOTIT                           SUBROUTINE


 Pass One      3 Errors Detected
              85 Source Lines
```

The Linker Map

The following diagram illustrates how the link map is translated during execution time:



Note: At execution time:

1. The code segment:offsets are correct relative to zero (load point of program just above DOS).

2. The data and constant references use the map offsets relative to the DS register, *not* the segment value displayed in the maps.

CHAPTER 3.
COMPILER METACOMMANDS

Contents

Overview

This chapter describes the compiler metacommands that process your Fortran source text.

Compiler metacommands direct the Fortran compiler to process Fortran source text in particular ways, as described below. They may be intermixed with Fortran source text within a Fortran source program; however, they are not part of the Fortran 77 language.

Any line of input to the Fortran compiler that begins with a $ in column 1 is interpreted as a compiler metacommand and must conform to one of the formats described in the following paragraphs. A compiler metacommand must fit on a single source line; continuation lines are not permitted. Also, embedded blanks are not permitted in metacommand lines except for literals.

$DEBUG
Metacommand

Purpose: $DEBUG directs that all subsequent arithmetic operations are tested for division by 0, causes source file line numbers to be reported when a runtime error occurs and causes assigned GOTO label lists to be checked.

Format: $DEBUG

Remarks: A run-time error is generated if such a condition is detected.

This metacommand can appear anywhere in a program and will be in effect until a $NODEBUG metacommand is encountered.

The default value of the $DEBUG/$NODEBUG pair of metacommands is $DEBUG.

Example:

```
$DEBUG
C This program segment contains debug code
      X=1
      Y=0
      Z=X/Y
$NODEBUG
C Now debug code is not generated
      X=2*4
      Y=0
      Z=X/Y
```

$DO66
Metacommand

Purpose: $DO66 allows DO statements to recognize Fortran 66 semantics.

Format: $DO66

Remarks: $DO66 must precede the first declaration or executable statement of the source file in which it occurs.

The Fortran 66 semantics are as follows. First, all DO statements are executed at least once. Second, extended range is permitted; that is, control may transfer in and out of the syntactic body of a DO statement.

The range of the DO statement is thereby extended to logically include any statement that may be executed between a DO statement and its ending statement. However, the transfer of control into the range of a DO statement prior to the execution of the DO statement or following the final execution of its ending statement is invalid.

If a program does not contain a DO66 metacommand, the default is to Fortran 77 semantics, as follows:

- DO statements may be executed zero times, if the initial control variable value exceeds the final control variable value (or the corresponding condition for a DO statement with negative increment).
- Extended range is invalid; that is, control may not transfer in and out of the syntactic body of a DO statement.

$DO66
Metacommand

Example:

```
$DO66
C The following loop is executed once because
C of the DO66 metacommand
      COUNT=0
      DO 10 I=2,1
         •
         •
         •
      COUNT=COUNT+1
         •
   10 Continue
      WRITE(*,20)COUNT
   20 FORMAT('  ',F10.1)
      END
```

$INCLUDE Metacommand

Purpose: $INCLUDE directs the compiler to proceed as though the specified source file were inserted in text at the point of INCLUDE. At the end of the included file, the compiler resumes processing the original source file at the line following INCLUDE.

Format: $INCLUDE:'*filespec*'

Remarks: The *filespec* entry is a valid file specification as described for the Disk Operating System (DOS).

INCLUDE metacommands can be nested as deeply as the storage on your system allows. INCLUDE metacommands are particularly useful in ensuring that several modules use the same declaration for a COMMON block.

$INCLUDE
Metacommand

Example:

```
REAL (X100),LARGE
COMMON /SHARED/ N, X, J, LARGE
```

These Fortran source statements are contained in the diskette file COMMON.FOR and may be included in the file below during compilation with $INCLUDE.

```
$INCLUDE:'COMMON.FOR'
      READ(*,100) N,(X(I),I=1,N)
100   FORMAT(I3,100(F10.5))
      CALL MAX
      WRITE(*,100) N, LARGE
110   FORMAT(1X,I3,F10.5)
      STOP
      END

      SUBROUTINE MAX
$INCLUDE:'COMMON.FOR'
      LARGE=X(1)
      J=1
      IF(N.EQ.1) GO TO 7
      DO 6 I=2,N
         IF(X(i).LE.LARGE) GO TO 6
         LARGE=X(i)
         J=I
6     CONTINUE
      RETURN
      END
```

$LINESIZE Metacommand

Purpose: $LINESIZE metacommand directs subsequent pages of the listing to be formatted *n* characters wide.

Format: $LINESIZE:*n*

Remarks: The *n* entry can be any positive integer between 40 and 132.

If a program does not contain a LINESIZE metacommand, a default line size of 80 characters is assumed.

$LIST
Metacommand

Purpose: $LIST turns on the generation of the listing file.

Format: $LIST

Remarks: This metacommand can appear anywhere in a program and will be in effect until a $NOLIST metacommand is encountered.

The default value of the $LIST/$NOLIST pair of metacommands is $LIST.

$NODEBUG Metacommand

Purpose: $NODEBUG turns off DEBUG, removing checks for division by 0, assigned GOTO label lists and causes source file line number not to be reported when a runtime error occurs.

Format: $NODEBUG

Remarks: The metacommand can appear anywhere in a program and will be in effect until a $DEBUG metacommand is encountered.

The default value of the $DEBUG/$NODEBUG pair of metacommands is $DEBUG.

$NOLIST
Metacommand

Purpose: $NOLIST turns off the generation of the listing file.

Format: $NOLIST

Remarks: This metacommand can appear anywhere in a program and will be in effect until a $LIST metacommand is encountered.

The default value of the $LIST/$NOLIST pair of metacommands is $LIST.

$PAGE
Metacommand

Purpose: $PAGE starts a new page of the listing.

Format: $PAGE

Remarks: If the first character of a line of source text is the ASCII form feed character (CONTROL-L), the form feed is treated as equivalent to the occurrence of a PAGE metacommand at that point.

$PAGESIZE
Metacommand

Purpose: $PAGESIZE metacommand directs subsequent pages of the listing to be formatted *n* lines high.

Format: $PAGESIZE:*n*

Remarks: The *n* entry can be any positive integer between 15 and 32,767.

If a program does not contain a $PAGESIZE metacommand, a default page size of 66 lines is assumed.

$STORAGE Metacommand

Purpose: $STORAGE directs that all variables declared in the source file as INTEGER or LOGICAL are allocated *n* bytes of storage. STORAGE does not affect the allocation of storage for variables declared with an explicit length specification, for example, as INTEGER*n* or LOGICAL*n*. If several files of a source program are linked together, you should be particularly careful that the various program units allocate storage consistently for variables (such as actual and formal parameters) referred to in more than one module.

Format: $STORAGE:*n*

Remarks: The *n* entry is either 2 or 4.

This metacommand must precede the first declaration or executable statement of the source file in which it occurs.

If a program does not contain a $STORAGE metacommand, a default allocation of four bytes is used.

Notes:

1. Specifying the $STORAGE:2 metacommand will increase the speed of your program. Also, it will decrease the size of your PASIBF files.

2. The default results in INTEGER, LOGICAL, and REAL variables being given the same amount of storage.

$SUBTITLE
Metacommand

Purpose: $SUBTITLE directs subsequent pages of the listing to be headed with the specified title, until overridden by another SUBTITLE metacommand.

Format: $SUBTITLE:'*subtitle*'

Remarks: The *subtitle* entry can be any valid character constant of length 0 through 40.

If a program does not contain a SUBTITLE metacommand, the null string is used as a subtitle.

This command must appear on the first line of the source file for the SUBTITLE to appear on the first page.

$TITLE
Metacommand

Purpose: $TITLE directs subsequent pages of the listing to be headed with the specified title, until overridden by another TITLE metacommand.

Format: $TITLE:'*title*'

Remarks: The *title* entry can be any valid character constant of length 0 through 40.

If a program does not contain a TITLE metacommand, the null string is used as a title.

This command must appear on the first line of the source file for the TITLE to appear on the first page.

CHAPTER 4. STATEMENTS

Contents

Control Statements

Control statements control the order of execution of statements in IBM Fortran. The control statements are as follows:

- Arithmetic IF
- Assignment
- Assigned GOTO
- Block IF
- CALL
- Computed GOTO
- CONTINUE
- DO
- ELSE
- ELSEIF
- END
- ENDIF
- Logical IF
- PAUSE

- RETURN
- STOP
- Unconditional GOTO

Block IF THEN ELSE

The Block IF, ELSEIF, ELSE, and ENDIF are described in this chapter. These statements are new to Fortran and are intended to improve the readability and structure of Fortran programs. As an overview of these subsections, the following three code examples illustrate the basic concepts.

> **Note:** Transfer of control into an IF block from outside that block is not permitted.

Example 1

Simple Block IF that skips a group of statements if the expression is false:

```
IF(I.LT.10)THEN
  •
  •  Some statements executed only if I.LT.10
  •
ENDIF
```

Example 2

Block IF with a series of ELSEIF statements:

```
IF(J.GT.1000)THEN
  •
   • Some statements executed only if J.GT.1000
    •
ELSEIF(J.GT.100)THEN
      •
       • Some statements executed only if
        • J.GT.100 and J.LE.1000
  ELSEIF(J.GT.10)THEN
          •
           • Some statements executed only if
            • J.GT.10 and J.LE.1000 and J.LE.100
       ELSE
       •
        • Some statements executed only if none of
         • above conditions are true
ENDIF
```

Example 3

Illustrates that the constructs can be nested and that an ELSE statement can follow a block IF without intervening ELSEIF statements (indentation solely to enhance readability):

```
IF(I.LT.100)THEN
  •
   • Some statements executed only if I.LT.100
    •
     IF(J.LT.10)THEN
      •
       • Some statements executed only if
        • I.LT.100 and J.LT.10
     ENDIF
•
 • Some statements executed only if I.LT.100
  •
ELSE
 •
  •Some statements executed only if I.GE.100
   •
    If (J.LT.10)THEN
     •
      • Some statements executed only if
       • I.GE.100 and J.LT.10
    ENDIF
 •
  • Some statements executed only if I.GE.100
   •
ENDIF
```

To understand in detail the block IF and associated statements, the concept of an IF-level is introduced.

For any statement, its IF-level is:

n1 - n2

The n1 entry is the number of block IF statements from the beginning of the program unit that the statement is in, up to and including that statement.

The n2 entry of a statement is the number of ENDIF statements from the beginning of the program unit, up to, but not including, that statement.

The IF-level of every statement must be greater than or equal to 0 and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than 0. Finally, the IF-level of every END statement must be 0. The IF-level defines the nesting rules for the block IF and associated statements and defines the extent of IF, ELSEIF, and ELSE blocks.

Example:

```
C THE NEXT STATEMENT HAS AN IF-LEVEL (0-0)
        CHARACTER CH1*5,CH2*6
C THE IF-LEVEL OF THE NEXT 3 STATEMENTS IS (1-0)
        IF(K.GT.P) THEN
          CH1='KERRY'
          CH2='SAYERS'
C THE IF-LEVEL OF THE NEXT 3 STATEMENTS IS (2-0)
          IF(K.LT.P) THEN
            CH1='PAUL'
            CH2='SAYERS'
          ENDIF
        ENDIF
```

Program Function and Subroutine Statements

These statements are:

- FUNCTION
- PROGRAM
- Statement Function
- SUBROUTINE

Main Program

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. In addition, it may have a PROGRAM statement as its first statement. The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program.

Subroutines

A subroutine begins with a SUBROUTINE statement and ends with the first subsequent END statement. It can contain any statement other than a PROGRAM statement, SUBROUTINE statement, or FUNCTION statement. A subroutine is a program unit that can be called from other program units by a CALL statement. When invoked, it performs the set of actions defined by its executable statements, and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via parameters or common variables.

Functions

A function is referred to in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions: external, intrinsic, and statement.

Formal Parameters

This subsection discusses the relationship between formal and actual arguments in a function or subroutine call. A formal parameter is the name by which the actual argument is known within the function or subroutine, and an actual argument is the specific variable, expression, array, and so forth, passed to the procedure in question at any specific calling location.

Actual arguments pass values by reference into and out of procedures. The number of actual arguments must be the same as formal parameters, and the corresponding types must agree.

Upon entry to a subroutine or function, the actual arguments are associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables.

This association remains in effect until execution of the subroutine or function is terminated. Thus, assigning a value to a formal parameter during execution of a subroutine or function may alter the value of the corresponding actual argument.

If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal parameter is not permitted, and can have some strange side effects. In particular, assigning a value to a formal parameter of type character, when the actual argument is a literal, can produce unpredictable results.

If an actual argument is an expression, it is evaluated immediately before the association of formal parameters and actual arguments. If an actual argument is an array element, its subscript expression is evaluated immediately before the association, and remains constant throughout the execution of the procedure, even if it contains variables that are redefined during the execution of the procedure. A formal parameter that is a variable can be associated with an actual argument that is a variable, an array element, or an expression.

A formal parameter that is an array can be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal parameter may be different than those of the actual argument, but any reference to the formal array must be within the limits of the storage sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running Fortran program, the results are unpredictable.

A formal parameter can also be the name of an external procedure, function, or intrinsic function. The actual argument must appear in an EXTERNAL or INTRINSIC statement in the program unit in which the procedure or function reference is made.

The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, REAL, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT), for choosing the largest or smallest value (MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, MIN1) and EOF must not be used as actual arguments.

I/O Statements

The I/O statements are as follows:

- BACKSPACE
- CLOSE
- ENDFILE
- OPEN
- READ
- REWIND
- WRITE

In addition, an I/O intrinsic function, EOF(u) (see Chapter 6), returns a logical value indicating whether the file associated with the unit specifier (u) passed to it is at the end of the file.

Elements of I/O Statements

The various I/O statements take certain arguments that specify sources and destinations of data transfer, as well as other functions of the I/O operation. The abbreviations used in this subsection are the unit specifier (*u*), format specifier (*f*), and input/output list (*iolist*) are as follows:

The unit specifier (*u*) can take one of these forms in an I/O statement:

- *—refers to the keyboard/display
- Integer expression—refers to an external file with a unit number equal to the value of the expression (* is unit number 0).
- Name of a character variable or character array element—refers to the internal file specified by the value of the variable or array element.

The format specifier, *f*, can take one of these forms in an I/O statement:

- Statement label—refers to a FORMAT statement labeled by that statement label.
- Integer variable name—refers to a FORMAT label assigned to that integer variable using the ASSIGN statement.
- Character expression—the format specified is the current value of the character expression provided as the format specifier.

The input/output list, *iolist*, specifies the entities whose values are transferred by READ and WRITE statements. An *iolist*, a possibly empty list, consists of input or output entities and implied DO lists, separated by commas.

Input and Output Entities

An input entity can be specified in the *iolist* of a READ statement and an output entity in the *iolist* of a WRITE statement. An entity is either a variable name, an array element name, or an array name. An array name is a means of specifying all of the elements of the array in storage sequence order.

An output entity can also be any other expression not beginning with the character "(", to distinguish implied DO lists from expressions.

The expression:

```
(A+B)*(C+D)
```

must be written as:

```
+(A+B)*(C+D)
```

to distinguish it from an implied DO list. Extra code is not generated for the (+).

Implied DO Lists

Implied DO lists can be specified as items in the I/O list of READ and WRITE statements and have the form:

```
(iolist, i = e1, e2 [, e3])
```

The *iolist* entry includes implied DO lists. The *i*, *e1*, *e2*, *e3* entries are as defined for the DO statement. That is, *i* is an integer variable, *e1*, *e2*, and *e3* are integer expressions and, if *i* is INTEGER*2, then *e1*, *e2*, and *e3* must be INTEGER*2.

In a READ statement, the DO variable *i* (or an associated entity) must not appear as an input list item in the embedded *iolist*, but may have been read in the same READ statement outside of the implied DO list. The embedded *iolist* is effectively repeated for each iteration of *i* with appropriate substitution of values for the DO variable *i*. In the case of nested implied DO loops, the innermost (most deeply nested) loop is always executed fastest.

Example:

```
        INTEGER ARRAY (3,2)
C READ VALUES FOR A 3 BY 2 ARRAY FROM THE
C CONSOLE IN ROW MAJOR ORDER.
        READ (*,200) ((ARRAY (I,J),J=1,2),I=1,3)
200     FORMAT(6I)
        WRITE(*,300) ((ARRAY(I,J),J=1,2),I=1,3)
300     FORMAT(' ',6I2)
        END
```

Specification Statements

Specification statements in IBM Fortran define the attributes of user-defined variable, array, and function names. They are nonexecutable.

There are eight kinds of specification statements:

- COMMON
- DIMENSION
- EQUIVALENCE
- EXTERNAL
- IMPLICIT
- INTRINSIC
- SAVE
- TYPE

Specification statements must precede all executable statements in a program unit. All of them except IMPLICIT may appear in any order within a program unit. IMPLICIT statements must precede all other specification statements in a program unit.

Purpose: The arithmetic IF statement causes evaluation of the expression and selection of a label based on the value of the expression.

Format: IF (*e*) *s1*, *s2*, *s3*

Remarks: The *e* entry is an integer or real expression.

The *s1*, *s2*, *s3* entries are statement labels of executable statements found in the same program unit as the arithmetic IF statement.

The same statement label may appear more than once among the three labels:

- Label *s1* is selected if the value of *e* is less than 0.
- Label *s2* if the value of *e* equals 0.
- Label *s3* if the value of *e* exceeds 0.

The next statement executed is the statement labeled by the selected label. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. (A special feature, extended range DO loops, permits jumping into a DO block. See the DO66 metacommand in Chapter 3 for more information.)

Arithmetic IF

Example:

```
C EXAMPLE OF ARITHMETIC IF
      WRITE (*,90)
90    FORMAT (' ENTER TEST SCORE:')
      READ (*,100) I
100   FORMAT (I3)
      IF (I-75) 10,20,30
10    WRITE (*,500)
      STOP
20    WRITE (*,510)
      STOP
30    WRITE (*,520)
      STOP
500   FORMAT (' YOU FAILED')
510   FORMAT (' YOU JUST PASSED')
520   FORMAT (' YOU PASSED JUST FINE')
      END
```

An assignment statement assigns a value to a variable or an array element. There are two basic kinds of assignment statements: computational and label.

Computational Assignment Statement

Purpose: A computational assignment statement evaluates the expression and assigns the resulting value to the variable or array element appearing on the left.

Format: *var = expr*

Remarks: The *var* entry is a variable or array element name. The *expr* entry is an expression.

The type of the variable or array element and the expression must be compatible. They must both be either numeric, logical, or character, in which case the assignment statement is called an arithmetic, logical, or character assignment statement.

If the types of the elements of an arithmetic assignment statement are not identical, automatic conversion of the value of the expression to the type of the variable is done.

The conversion rules are given in the table below.

	Integer Expression (E)	**Real Expression (E)**
Integer Element (V)	Assign E to V	Truncate E to integer and assign to V
Real Element (V)	Append fraction (.0) to E and assign to V	Assign E to V

If the length of the expression does not match the size of the variable in a character assignment statement, it is adjusted so that it does. If the expression is shorter, it is padded with enough blanks on the right to make the sizes equal before the assignment takes place. If the expression is longer, characters on the right are truncated to make the sizes the same.

Logical expressions of any size can be assigned to logical variables of any size without effect on the value of the expression.

> **Note:** INTEGER*2 and INTEGER*4 are treated the same, except that INTEGER*4 may be assigned in accordance with its range of values. Note also that INTEGER*4 expressions may not be assigned to INTEGER*2 variables.

ASSIGN Statement

Purpose: The ASSIGN statement assigns the value of a format or statement label to an integer variable.

Format: ASSIGN *label* TO *var*

Remarks: The *label* entry is a format label or statement label. The *var* entry is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format or a statement label, and it must appear in the same program unit as the ASSIGN statement. When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an input/output statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

> **Note:** The "value" of a label is not necessarily the same as the label number.

For example, the value of LABEL in:

```
ASSIGN 400 TO LABEL
```

is not necessarily 400.

Also note that this makes the variable undefined as an integer and it cannot be used in an arithmetic expression until it has been redefined (by an assignment or READ statement) as such.

Example:

```
ASSIGN 10 TO INSERT
GOTO INSERT
•
•
•
10  PAUSE 'INSERT YOUR DISKETTE'
```

Purpose: The assigned GOTO statement causes the next statement executed to be the statement labeled by the label last assigned to i.

Format: GOTO *i* [[,] (*s* [, *s*] . . .)]

Remarks: The *i* entry is an integer variable name.

The *s* entry is a statement label of an executable statement found in the same program unit as the assigned GOTO statement.

The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, *i* must have been assigned the label of an executable statement found in the same program unit as the assigned GOTO statement. If the optional list of labels is present and $DEBUG is on, a runtime error is generated if the label last assigned to *i* is not among those listed. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. (A special feature, extended range DO loops, permits jumping into a DO block. See the $DO66 metacommand in Chapter 3 for more information.)

Assigned GOTO

Example:

```
C EXAMPLE OF ASSIGNED GOTO
      ASSIGN 10 TO CHECK
5     GOTO CHECK (10,20,40)
10    IF(IDV.NE.0) THEN
            ASSIGN 20 TO CHECK
            GOTO 5
      ELSE
            STOP 'Divisor is zero'
      ENDIF
20    WRITE(*,50) TOP/IDV
50    FORMAT(1X,' Quotient is ',F10.5)
            •
            •
            •
40    CONTINUE
```

BACKSPACE Statement

Purpose: BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record.

Format: BACKSPACE *u*

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" in this chapter). It is required and must not be an internal unit specifier.

When BACKSPACE is used with Unformatted/ Sequential files, the file is backed up by only one byte. When taking advantage of this limitation, the BINARY filemode should be used.

> **Note:** See "Concepts and Limitations" in Chapter 5 for detailed information.

If there is no preceding record, the file position is not changed.

> **Note:** If the preceding record is the end of file record, the file becomes positioned before the end of file record. If the file position is in the middle of the record, BACKSPACE positions to the start of that record.

Block IF

Purpose: The block IF statement causes the expression *e* to be evaluated. If it evaluates to true and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block.

Format: IF (*e*) THEN

Remarks: The *e* entry is a logical expression.

The IF block associated with this block IF statement consists of all the executable statements, possibly none, that appear following this statement up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement (the IF-level defines the notion of "matching" ELSEIF, ELSE, or ENDIF).

The ENDIF statement is the next statement to be executed after the last statement in the IF block (at the same IF-level as this block IF statement). If the expression in this block IF statement evaluates to true and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the block IF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement. See "Block IF THEN ELSE" under "CONTROL STATEMENTS" in this chapter.

Transfer of control into an IF block from outside that block is not permitted.

CALL Statement

Purpose: A subroutine is executed by issuing a CALL statement in another program unit which references that subroutine.

Format: CALL *sname* [(*arg* [, *arg*] ...)]

Remarks: The *sname* entry is the user-defined name of a subroutine.

The *arg* entry is an actual argument.

An actual argument may be either an expression or the name of an array. The actual arguments in the CALL statement must agree in type and number with the corresponding formal parameters specified in the SUBROUTINE statement of the referenced subroutine. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine must not have any actual arguments, but may optionally have a pair of parentheses following the name of the subroutine. Note that a formal parameter can be used as an actual argument in another subprogram call.

Execution of a CALL statement proceeds as follows. All arguments that are expressions are evaluated. All actual arguments are associated with their corresponding formal parameters, and the body of the specified subroutine is executed. Control is returned to the statement following the CALL statement upon exiting the subroutine, by executing either a RETURN statement or an END statement in that subroutine.

CALL
Statement

A subroutine can be called from any program unit. Recursive subroutine calls, however, are not permitted in IBM Fortran. That is, a subroutine cannot call itself directly, nor can it call another subroutine that results in that subroutine being called again before it returns control to its caller.

Example:

```
C EXAMPLE OF CALL STATEMENT
      IF (IERR .NE. 0) CALL ERROR(IERR)
      END
C
      SUBROUTINE ERROR(IERRNO)
      WRITE (*, 200) IERRNO
200   FORMAT(1X, 'ERROR', I5, 'DETECTED')
      END
```

CLOSE Statement

Purpose: CLOSE disconnects the unit specified and prevents subsequent I/O from being directed to that unit (unless the same unit number is reopened, possibly bound to a different file or device).

Format: CLOSE(*u*[,STATUS=*st*])

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" previously described. It is required, and must appear as the first argument. It must not be an internal unit specifier.

The *st* entry is 'KEEP' or 'DELETE', an optional argument that applies only to files opened 'NEW'. The default is 'KEEP'. This option is a character constant. Files are discarded if STATUS='DELETE' is specified. Normal termination of a Fortran program automatically closes all open files as if CLOSE with STATUS='KEEP' was specified. CLOSE for unit 0 has no effect, since the CLOSE operation is not meaningful for keyboard and display.

Example:

```
C Close the file opened in OPEN example,
C discarding the file.
      CLOSE(7,STATUS='DELETE')
```

COMMON Statement

Purpose: A COMMON statement provides a method of sharing storage between two or more program units. Such program units can share the same data without passing it as arguments.

Format: COMMON [/*cname*/]*nlist*[,/*cname*/*nlist*]. . .

Remarks: The *cname* entry is a common block name. If *cname* is omitted, then the blank common block is specified.

The *nlist* entry is a comma-separated list of variable names, array names, and array declarators. Formal parameter names and function names cannot appear in a COMMON statement.

In each COMMON statement, all variables and arrays appearing in each *nlist* following a common block name *cname* are declared to be in that common block. If the first *cname* is omitted, all elements appearing in the first *nlist* are specified to be in the blank common block.

Any common block name can appear more than once in COMMON statements in the same program unit. All elements in all *nlists* for the same common block are allocated in that common storage area in the order they appear in the COMMON statement.

All elements in a single common area must be either all or none of type character. Furthermore, if two program units refer to the same named common block containing character data, the association of character variables of different length is not permitted. Two variables are said to be associated if they refer to the same actual storage.

The size of a common block is equal to the number of bytes of storage required to hold all elements in that common block. If the same named common block is referred to by several distinct program units, the common blocks must be of the same length and the blocks are juxtaposed at their lowest address. Blank common blocks, however, can have different lengths in different program units. The maximum length may occur in any program unit.

COMMON
Statement

Example:

```
C EXAMPLE OF BLANK AND NAMED COMMONS
      PROGRAM MYPROG
      COMMON I, J, X, K(10)
      COMMON /MYCOM/ A(3)
      I=1
      CALL MYSUB
      •
      •
      •
      END

      SUBROUTINE MYSUB
      COMMON IOTHER, JOTHER, XOTHER, K(10)
      COMMON /MYCOM/ A(3)
      IF (IOTHER.NE.1) STOP 'Something''s wrong'
      •
      •
      •
      END
```

Computed GOTO

Purpose: The Computed GOTO causes the next statement executed to be the one labeled by the *i*th label in the list of labels.

Format: GOTO (*s* [, *s*] . . .) [,] *i*

Remarks: The *s* entry is a statement label of an executable statement found in the same program unit as the computed GOTO statement.

The *i* entry is an integer expression.

The same statement label may appear repeatedly in the list of labels.

The effect of the computed GOTO statement is as follows. Suppose that there are n labels in the list of labels. If *i* is out of range, *i*<1, or *i*>n, then the computed GOTO statement acts as if it were a CONTINUE statement; otherwise the next statement executed is the one labeled by the *i*th label in the list of labels. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. (A special feature, extended range DO loops, permits jumping into a DO block. See the $DO66 metacommand in Chapter 3 for more information.)

Computed GOTO

Example:

```
C EXAMPLE OF COMPUTED GOTO
      I = 1
      GOTO (10, 20, 30) I
      •
      •
      •
10    STOP 'I=1'
      •
      •
      •
20    STOP 'I=2'
      •
      •
      •
30    STOP 'I=3'
```

Purpose: The primary use for the CONTINUE statement is a convenient statement to label, particularly as the ending statement in a DO loop.

Format: CONTINUE

Remarks: The execution of a CONTINUE statement has no effect.

Example:

```
C EXAMPLE OF CONTINUE STATEMENT
      DO 10, I = 1, 10
      IARRAY(I) = 0
10    CONTINUE
```

DATA
Statement

Purpose: The DATA statement assigns initial values to variables. A DATA statement is a nonexecutable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements.

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

Remarks: The *nlist* entry is a list of variable, array element, or array names.

The *clist* entry is a list of constants, or constants preceded by an integer constant repeat factor and an asterisk, such as:

```
5*3.14159    3*'Help'    100*0
```

A repeat factor followed by a constant is the equivalent of a list of all constants of that constant's value repeated a number of times equal to the repeat constant.

You must have the same number of values in each *clist* as you have variables or array elements in the corresponding *nlist*. The appearance of an array in an *nlist* is the equivalent to a list of all elements in that array in storage sequence order. Array elements must be indexed only by constant subscripts.

The type of each noncharacter element in a *clist* must be the same as the type of the corresponding variable or array element in the accompanying *nlist*. Each character element in a *clist* must correspond to a character variable or array element in the *nlist*, and must have a length that is less than or equal to the length of that variable or array element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right.

Only local variables and array elements can appear in a DATA statement. Formal parameters, variables in common, and function names cannot be assigned initial values with a DATA statement.

Example:

```
REAL A(3),B(9,6),LIST(4)*2
DATA A/14.1,3.6,2.81/
DATA B/54*0.01/,LIST(1)/2.1/
WRITE(*,'(1X,F4.1)')B
      •
      •
      •
```

DIMENSION Statement

Purpose: A DIMENSION statement specifies the maximum values of an array variable subscripts and allocates storage accordingly.

Format: DIMENSION *name*(*d*[,*d*[,*d*]])[,*name*(*d*[,*d*[,*d*]])]...

Remarks: An array declarator is of the form:

name(*d*[,*d*[,*d*]])

The *name* entry is the user-defined name of the array. The *d* entry is a dimension declarator.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. A dimension declarator can be:

- An unsigned integer constant.
- A user name corresponding to a nonarray integer formal parameter.
- An asterisk.

A dimension declarator specifies the upper bound of the dimension. The lower bound is always one.

If a dimension declarator is an integer constant, then the array has the corresponding number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants.

If a dimension declarator is an integer argument, then that dimension is defined to be of a size equal to the initial value of the integer argument upon entry to the subprogram unit at execution time. In such a case, the array is called an adjustable-sized array.

If the dimension declarator is an asterisk, the array is an assumed-sized array and the upper bound of that dimension is not specified.

All adjustable- and assumed-sized arrays must also be formal parameters to the program unit in which they appear. Also, an assumed-size dimension declarator may only appear as the last dimension in an array declarator.

The order of array elements in storage is column-major order, that is, the leftmost subscript changes most rapidly in a storage sequential reference to all array elements.

Example:
```
DIMENSION A(10,2),X(3)
DIMENSION BIG(100,100)
```

DO
Statement

Purpose: The DO statement causes repetitive evaluation of statements following the DO through and including the ending statement.

Format: DO *s i=e1, e2* [, *e3*]

Remarks: The *s* entry is a statement label of an executable statement. The *i* entry is an integer variable. The *e1*, *e2*, *e3* entries are integer expressions.

The label must follow this DO statement and be contained in the same program unit. The statement labeled by *s* is called the ending statement of the DO loop. It must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the ending statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

A DO loop has a "range" beginning with the statement that follows the DO statement and ending with (and including) the ending statement of the DO loop.

If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop, although the loops may share the ending statement.

If a DO statement appears within an IF, ELSEIF, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block.

If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop.

The DO variable, *i*, may not be assigned in any way by the statements within the range of the DO loop associated with it. Jumping into the range of a DO loop from outside its range is not permitted. (However, there is a special feature, added for compatibility with earlier versions of Fortran, that permits "extended range" DO loops. See the $DO66 metacommand in Chapter 3 for more information.)

The execution of a DO statement causes the following steps:

1. The expressions *e1*, *e2*, and *e3* are evaluated. If *e3* is not present, it is as if *e3* evaluated to 1 (*e3* must not evaluate to 0 and *e2* should not evaluate to the largest integer value).

2. The DO variable, *i*, is set to the value of *e1*.

   Note: The prohibition on assigning INTEGER*4 values to INTEGER*2 variables applies here. Therefore, if *i* is INTEGER*2, *e1*, *e2*, and *e3* (if it exists) must also be expressions of type INTEGER*2.

3. The iteration count for the loop is:

   MAX0(((*e2*–*e1*+*e3*)/*e3*),0)

   which may be 0 if either:

   e1>*e2* and *e3*>0

   or:

   e1<*e2* and *e3*<0

   Note: However if the $DO66 metacommand is in effect, the iteration count is at least 1.

4. The iteration count is tested, and if it exceeds 0, the statements in the range of the DO loop are executed.

Following the execution of the ending statement of a DO loop, the following steps occur:

1. The value of the DO variable, *i*, is incremented by the value of *e3* that was computed when the DO statement was executed.

2. The iteration count is decremented by 1.

3. The iteration count is tested, and if it exceeds 0, the statements in the range of the DO loop are executed again.

The value of the DO variable is well-defined regardless of whether the DO loop exists because the iteration count becomes 0 or because of a transfer of control out of the DO loop or RETURN statement.

An example of the final value of a DO variable:

Example:

```
C THIS PROGRAM FRAGMENT DISPLAYS THE NUMBERS
C 1 TO 11 ON THE SCREEN

      DO 200 I=1,10
200   WRITE(*,'(I5)')I
      WRITE(*,'(I5)')I

C EXAMPLE OF DO STATEMENT
C INITIALIZE A 20-ELEMENT REAL ARRAY
      DIMENSION ARRAY(20)
      DO 1 I = 1, 20
1     ARRAY(I) = 0.0
C PERFORM A FUNCTION 11 TIMES
      DO 2 I = -30, -60, -3
      J = I/3
      J = -9 - J
      ARRAY(J) = MYFUNC(I)
2     CONTINUE
```

ELSE

Purpose: An ELSE statement is used to associate an else block with an IF or ELSEIF statement.

Format: ELSE

Remarks: The ELSE block associated with an ELSE statement consists of all of the executable statements, possibly none, that follow the ELSE statement up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The "matching" ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level.

The execution of an ELSE statement has no effect.

Transfer of control into an ELSE block from outside that block is not permitted.

Example:

```
IF    (IHRS.LE.40) THEN
       WAGES=IHRS*RATE
ELSE
      IOVT=IHRS-40
      WAGES=(40.*RATE)+(IOVT*1.5*RATE)
ENDIF
```

Purpose: The ELSEIF statement causes evaluation of the expression. If its value is true and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block.

Format: ELSEIF (*e*) THEN

Remarks: The *e* entry is a logical expression.

The ELSEIF block associated with an ELSEIF statement consists of all the executable statements, possibly none, that follow up to the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement.

Following the execution of the last statement in the ELSEIF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this ELSEIF statement. If the expression in this ELSEIF statement evaluates to true and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF-level as the ELSEIF statement. If the expression evaluates to false, the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the ELSEIF statement. See "Block IF THEN ELSE" under "CONTROL STATEMENTS" in this chapter.

END

Purpose: The END statement indicates to the compiler that it has reached the end of a program unit.

Format: END

Remarks: Unlike other statements, an END statement must wholly appear on an initial line and contain no continuation lines. No other Fortran statement, such as the ENDIF statement, may have an initial line that appears to be an END statement.

The END statement in a subprogram has the same effect as a RETURN statement. In the main program, it terminates execution of the program. The END statement must appear as the last statement in every program unit.

Example:

```
C EXAMPLE OF END STATEMENT
C END STATEMENT MUST BE LAST STATEMENT
C IN A PROGRAM
      PROGRAM MYPROG
      WRITE(*,'(22H IBM PERSONAL COMPUTER)')
      END
```

ENDFILE Statement

Purpose: ENDFILE "writes" an end of file record as the next record of the file connected to the specified unit.

Format: ENDFILE *u*

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" in this chapter). It is required and must not be an internal unit specifier.

The file is then positioned after the end of file record, so further sequential data transfer is prohibited until either a BACKSPACE or REWIND is executed.

> **Note:** BACKSPACE is not supported for direct files. If you attempt to use BACKSPACE with direct files, this statement is ignored.

ENDIF

Purpose: An ENDIF statement is required to "match" every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

Format: ENDIF

Remarks: The execution of an ENDIF statement has no effect. See "Block IF THEN ELSE" under "CONTROL STATEMENTS" in this chapter.

EQUIVALENCE Statement

Purpose: An EQUIVALENCE statement specifies that two or more variables or arrays are to share the same storage. If the shared elements are of different types, the EQUIVALENCE does not cause any kind of automatic type conversion.

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

Remarks: The (*nlist*) entry is a list of at least two elements: variable names, array names, or array element names. Formal parameters may not appear in an EQUIVALENCE statement. Subscripts must be within the bounds of the array they index.

nlist::=*element*,*element*[,*element*]. . .

An EQUIVALENCE statement specifies that the storage sequences of the elements that appear in the list *nlist* have the same first storage location. Two or more variables are said to be associated if they refer to the same actual storage. Thus, an EQUIVALENCE statement causes its list of variables to become associated. An element of type character can only be associated with another element of type character with the same length. If an array name appears in an EQUIVALENCE statement, it refers to the first element of the array.

EQUIVALENCE Statement

Restrictions on EQUIVALENCE Statements

An EQUIVALENCE statement cannot specify that the same storage location is to appear more than once, such as:

Example:

```
C THIS IS AN ERROR
      REAL R,S(10)
      EQUIVALENCE (R,S(1)),(R,S(5))
```

This forces the variable R to appear in two distinct storage locations. Furthermore, an EQUIVALENCE statement cannot specify that consecutive array elements are not stored in sequential order. For example, the following is not permitted.

Example:

```
C THIS IS ANOTHER ERROR
      REAL R(10),S(10)
      EQUIVALENCE (R(1),S(1)),(R(5),S(6))
```

When EQUIVALENCE statements and COMMON statements are used together, several further restrictions apply. An EQUIVALENCE statement cannot cause storage in two different common blocks to become equivalenced. An EQUIVALENCE statement can extend a common block by adding storage elements following the common block, but not preceding the common block. Note that extending a named common block by an EQUIVALENCE statement must not cause its length to be different from the length of the named common in other program units. For example, the following is not permitted because it extends the common block by adding storage preceding the start of the block.

Example:

```
C THIS IS A MORE SUBTLE ERROR
     COMMON /ABCDE/ R(10)
     REAL S(10)
     EQUIVALENCE (R(1),S(10))

C EXAMPLE OF EQUIVALENCE STATEMENT
     CHARACTER*10 NAME, FIRST, MIDDLE, LAST
     DIMENSION NAME(60), FIRST(20),
    1          MIDDLE(20), LAST(20)
     EQUIVALENCE (NAME(1), FIRST(1)),
    1            (NAME(21),MIDDLE(1)),
    2            (NAME(41), LAST(1))
     NAME(1)='JOE'
     IF (NAME(1).NE.FIRST(1)) STOP
       'SOMETHING''S WRONG'
```

EXTERNAL Statement

Purpose: An EXTERNAL statement identifies a user-defined name as an external subroutine or function.

Format: EXTERNAL *name* [,*name*]. . .

Remarks: The *name* entry is the name of an external subroutine or function.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, then that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user name can only appear once in an EXTERNAL statement in a given program unit.

In the *IBM Personal Computer MACRO Assembler* and the *IBM Personal Computer Pascal Compiler* the term EXTERNAL (or EXTERN) declares that an object is defined outside the current unit of compilation or assembly. This is not necessary in IBM Fortran since, in accord with the standard Fortran practice, any object referred to but not defined in a compilation unit is assumed to be defined externally. Therefore, in Fortran, EXTERNAL is only needed to specify that a particular user-defined subroutine or function is to be used as a procedural parameter.

Example:

```
C EXAMPLE OF EXTERNAL STATEMENT
      EXTERNAL MYFUNC, MYSUB
C MYFUNC AND MYSUB ARE PARAMETERS TO CALC
      CALL CALC (MYFUNC, MYSUB)
```

FUNCTION Statement

Purpose: The FUNCTION statement identifies a program unit as a function and supplies its type, name, and formal parameter.

Format: [*type*] FUNCTION *fname* ([*fparm* [,*fparm*] . . .])

Remarks: The *type* entry is INTEGER, REAL, or LOGICAL.

The *fname* entry is the user-defined name of the function.

The *fparm* entry is a formal parameter name.

The *fname* entry is a global name, and it is also local to the function it names. If *type* is not present in the FUNCTION statement, the function's type is determined by default and by any subsequent IMPLICIT or type statements that would determine the type of an ordinary variable. If *type* is present, then the function name cannot appear in any additional type statements. An external function cannot be of type CHARACTER. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Neither argument names nor *fname* can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

FUNCTION
Statement

The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or an END statement, defines the value of the function. After being defined, the value of this variable can be referenced in an expression, exactly as any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal parameters.

A function can be called from any program unit. Recursive function calls, however, are not permitted in IBM Fortran. That is, a function cannot call itself directly, nor can it call another function that results in that function being called again before it returns control to its caller.

Example:

```
C EXAMPLE OF A FUNCTION REFERENCE.
C REEDCH IS A FUNCTION THAT READS A
C CHARACTER FROM THE CONSOLE AND
C RETURNS THAT CHARACTER IN THE
C PARAMETER CH.
      CHARACTER CH
      DATA CH /'{'/
      IF(REEDCH(CH) .GT.64)
     + WRITE(*,'("  ",A)') CH
      END

      FUNCTION REEDCH(CHR)
      IMPLICIT INTEGER(R)
      CHARACTER CHR
      READ(*,'(A)') CHR
      REEDCH=ICHAR(CHR)
      END
```

IMPLICIT Statement

Purpose: An IMPLICIT statement defines the default type for user-declared names.

Format: IMPLICIT *type*(*a*[,*a*]. . .)[,*type*(*a*[,*a*]. . .)]. . .

Remarks: The *type* entry is one of the types described in Chapter 1.

The *a* entry is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range, separated by a minus sign. For a range, the letters must be in alphabetical order.

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters that appear in the specification. An IMPLICIT statement applies only to the program unit in which it appears. IMPLICIT statements do not change the type of any intrinsic functions.

Notes:

1. Either 2 or 4 bytes are used. The default is 4, but may be set explicitly to either 2 or 4 with the $STORAGE metacommand.

2. CHARACTER and CHARACTER*1 are synonyms.

3. If *n* is odd, then *n* + 1 bytes of storage are used. (See Type Statement.)

4. REAL and REAL*4 are synonyms.

IMPLICIT types can be overridden or confirmed for any specific user name by the appearance of that name in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type in question is a character type, the length of the user name is also overridden by a later type definition.

A program unit can have more than one IMPLICIT statement, but all IMPLICIT statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

Example:

```
C EXAMPLE OF IMPLICIT STATEMENT
    IMPLICIT INTEGER (A-B)
    IMPLICIT CHARACTER*10 (N)
    AGE = 11
    NAME = 'RENEE'
```

INTRINSIC Statement

Purpose: An INTRINSIC statement declares that a name is an intrinsic function.

Format: INTRINSIC *name* [,*name*]. . .

Remarks: The *name* entry is an intrinsic function name. Each name may appear once in an INTRINSIC statement. It cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Chapter 6.

The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, REAL, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT), for choosing the largest or smallest value (MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, MIN1) and EOF must not be used as actual arguments.

Example:

```
C EXAMPLE OF INTRINSIC STATEMENT
      INTRINSIC SIN, COSIN
C SIN AND COSIN ARE PARAMETERS TO CALC2
      X = CALC2 (SIN, COSIN)
```

Logical IF

Purpose: The logical IF statement causes the logical expression to be evaluated and, if the value of that expression is true, then the st statement is executed.

Format: IF (*e*) *st*

Remarks: The *e* entry is a logical expression. The *st* entry is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement. If the expression evaluates to false, the *st* statement is not executed and the execution sequence continues as if a CONTINUE statement were encountered.

Example:

```
C EXAMPLE OF LOGICAL IF
      IF (I .EQ. 0) J = 2
      IF (X .GT. 2.3) GOTO 100
100   CONTINUE
```

OPEN Statement

Purpose: The OPEN statement binds a unit number with an external device or file on an external device by specifying its filename.

Format: OPEN(*u*,FILE=*fname*[,STATUS=*st*] [,ACCESS=*ac*] [,FORM=*fm*] [,RECL=*rl*])

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" previously described). It is required, and must appear as the first argument. It must not be an internal unit specifier.

The *fname* entry is a character expression. It is required, and must appear as the second argument. If a blank is specified, the filename can be specified at runtime. See "Runtime Filename Assignment" in this section.

All arguments after *fname* are optional and can appear in any order. These options are character constants with optional trailing blanks (except RECL=).

The *st* entry is 'OLD' (the default) or 'NEW'. 'OLD' is for reading or writing existing files. 'NEW' is for writing new files.

The *ac* entry is 'SEQUENTIAL' (the default) or 'DIRECT'.

The *fm* entry is 'FORMATTED' (the default), 'UNFORMATTED', or 'BINARY'.

The *rl* entry is the record length, an integer expression. This argument to OPEN is for direct access files only, for which it is required.

Binding unit 0 to a file has no effect. Unit 0 is permanently connected to the keyboard and display. If the file is to be direct, the RECL=*rl* option specifies the length of the records in that file.

If an OPEN of a currently open unit is executed, it functions as if a CLOSE and then an OPEN is executed. If the filename in the FILE parameter is the blank filename, the file system will attempt to get the filename at runtime. The filename may also be specified on the command line.

Example: EXAMPLE PROGRAM FRAGMENT 1

```
C PROMPT USER FOR A FILE NAME.
      WRITE(*,'(A)')'SPECIFY OUTPUT FILE NAME -'
C PRESUME THAT fname IS SPECIFIED TO BE
C CHARACTER*64. READ THE FILE NAME FROM THE
C KEYBOARD.
      READ(*,'(A)') fname
C OPEN THE FILE AS FORMATTED SEQUENTIAL AS
C UNIT 7.
      OPEN(7,FILE=fname,ACCESS='SEQUENTIAL',
      STATUS='NEW')
```

EXAMPLE PROGRAM FRAGMENT 2

```
C OPEN AN EXISTING FILE
C CALLED DATA3.TEXT AS UNIT 3.
      OPEN(3,FILE='DATA3.TXT')
```

Runtime Filename Assignment

Often it is desirable to supply your application with a filename for an OPEN statement at runtime instead of coding it directly in your program. This can be done in IBM Fortran by supplying a filename of all blanks on the FILE= parameter of the OPEN statement. Example:

```
      INTEGER UNITNO
      CHARACTER*1 FNAME
      DATA FNAME/' '/
C     Open units 5 through 7.
      DO 10 I=5,7
            OPEN(I,FILE=' ')
10    CONTINUE
C     Obtain the unit number and then perform the
C     OPEN
      WRITE(*,20)
20    FORMAT(1X,'Enter the Unit number: ')
      READ(*,'(BN,I6)') UNITNO
      OPEN(UNITNO,FILE=FNAME)
      END
```

When the OPEN statement is executed, the Fortran runtime system will see the blank filename and attempt to obtain the actual name of the DOS file or device to be opened.

The Fortran runtime system will attempt to obtain the DOS filespec as follows:

- Look on the command line of the run file where the blank filename has been coded. Take a DOS filespec from the command line for each OPEN as it is executed. The filespecs on the command line must be separated by blanks if more than one appears on the line.

```
B>files user prn con info.dat
Enter the Unit number:
  300
```

- When the filespecs on the command line are exhausted, prompt the user for the filespec. The message:

```
Filename missing or invalid – try again!
```

will be displayed if the command line is empty. The prompt

```
Unit #####?
```

where ##### is the unit number that needs to be supplied with a filename. Just respond to the prompt with a DOS filespec and the OPEN will be performed on that filename.

```
B>files
File name missing or empty – try again!
UNIT 5? user
UNIT 6? prn
UNIT 7? con
```

```
Enter the Unit number:
  -12
UNIT -12? info.dat
```

- Specifying extra filespecs on the command line has no effect.

```
B>files user prn con info.dat extra1.fil extra2.fil
Enter the Unit number:
  30
```

Note: Because the filespecs are obtained as the OPENs are executed, you must know the order in which the OPENs are executed, to be able to place all the filespecs on the command line correctly.

PAUSE
Statement

Purpose: The PAUSE statement causes the program to be suspended until the Enter key is pressed on the keyboard.

Format: PAUSE [*n*]

Remarks: The *n* entry is either a character constant or a string of not more than five digits.

The argument, *n*, if present, is displayed as part of the prompt requesting input from the keyboard. If *n* is not present, then "PAUSE" and the prompt message is displayed. To continue execution of the program, press the Enter key. Execution resumes as if a CONTINUE statement was executed.

Example:

```
C EXAMPLE OF PAUSE STATEMENT
      IF (IWARN .EQ. 0) GOTO 300
      PAUSE 'WARNING: IWARN IS NONZERO'
300   CONTINUE
```

PROGRAM Statement

Purpose: The PROGRAM statement identifies a program unit as a main program and names the program unit.

Format: PROGRAM *pname*

Remarks: The *pname* entry is a user-defined name that is the name of the main program.

The *pname* entry is a global name. Therefore, it cannot be the same as that of another external procedure or common block. (It is also a local name to the main program, and must not conflict with any local name in the main program.) The PROGRAM statement may only appear as the first statement of a main program.

Example:

```
PROGRAM PAYROL
INTEGER JOEPAY,TAX,OT
DIMENSION RATE(4,69)
  •
  •
  •
END
```

READ
Statement

Purpose: The READ statement sets the items in iolist (assuming that no end of file or error condition occurs).

Format: READ(*u*[,*f*][,REC=*rn*][,END=*s1*]
[,ERR=*s*])*iolist*

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" in this chapter.) It is required, and must appear as the first argument.

The *f* entry is required for formatted read as the second argument, and must not be used for unformatted read.

The *rn* entry is specified for direct access only, otherwise an error results. It is a positive integer expression. It sets the current position to record number *rn*. If REC=*rn* is omitted for a direct access file, reading continues sequentially from the current position in the files.

The *s1* entry is an optional statement label. If it is not present, reading the end of the file results in a runtime error. If it is present, encountering an end of file condition results in the transfer to the executable statement labeled *s1*, which must be in the same program unit as the READ statement.

The *s* entry is an optional statement label. If it is not present, I/O errors result in runtime errors. If it is present, I/O errors cause control to transfer to the executable statement labeled *s*. If the read is internal, the character variable or character array element specified is the source of the input, otherwise the external unit is the source.

Example:

```
C NEED A TWO DIMENSIONAL ARRAY FOR THE
C EXAMPLE.
      DIMENSION IA(10,20)
C READ IN THE BOUNDS FOR THE ARRAY. THESE
C BOUNDS SHOULD BE LESS THAN 10 AND 20
C RESPECTIVELY. THEN READ IN THE ARRAY IN
C NESTED IMPLIED DO LISTS WITH INPUT FORMAT OF
C 8 COLUMNS OF WIDTH 5 EACH.
      READ(3,990)IL,JL,((IA(I,J),J=1,JL),I=1,IL)
990   FORMAT(2I5/,(20I5))
```

RETURN
Statement

Purpose: A RETURN statement causes return of control to the calling program unit. It may appear only in a function or subroutine.

Format: RETURN

Remarks: Execution of a RETURN statement terminates the execution of the enclosing subroutine or function. If the RETURN statement is in a function, then the value of that function is equal to the current value of the variable with the same name as the function. Execution of an END statement in a function or subroutine is equivalent to execution of a RETURN statement.

Example:

```
C EXAMPLE OF RETURN STATEMENT
C THIS SUBROUTINE LOOPS UNTIL THE USER
C TYPES 'Y' TO THE KEYBOARD
      SUBROUTINE LOOP
      CHARACTER IN
C
10    READ(*, '(A1)') IN
      IF (IN .EQ. 'Y') RETURN
      GOTO 10
      RETURN
      END
```

REWIND Statement

Purpose: Execution of a REWIND statement causes the file associated with the specified unit to be positioned at its initial point.

Format: REWIND *u*

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statements" in this chapter). It is required and must not be an internal unit specifier.

SAVE Statement

Purpose: A SAVE statement retains the definition of a common block after the return from a procedure that defines that common block.

Format: SAVE */name/* [,*/name/*]. . .

Remarks: The *name* entry is the name of a common block. Within a subroutine or function, a common block that was specified in a SAVE statement does not become undefined upon exit from the subroutine or function.

> **Note:** Because all common blocks are statically allocated in the IBM Fortran, all common blocks are automatically saved, therefore, the SAVE statement has no effect.

Example:

```
C EXAMPLE OF SAVE STATEMENT
      COMMON/MYCOM/I,J,K
      SAVE /MYCOM/
```

Purpose: A statement function identifies a user-defined function in one statement. It is similar in form to an assignment statement. A statement function may appear only after specification statements and before any executable statements in the program unit.

A statement function is not classified as an executable statement, because it is not executed until referenced. The body of a statement function serves to define the meaning of the function. It is executed, as any other function, by the execution of a function reference.

Format: *fname* (*fparm* [, *fparm*] . . .) = *expr*

Remarks: The *fname* entry is the user-defined name of the statement function.

The *fparm* entry is a formal parameter name.

The *expr* entry is an expression.

The type of the *expr* must be assignment compatible with the type of the statement function name. The list of formal parameter names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function.

Therefore, formal parameter names can be used as other user-defined names in the rest of the program unit enclosing the statement function definition. The name of the statement function, however, is local to the enclosing program unit, and must not be used otherwise (except as the name of a common block, or as the name of a formal parameter to another statement function). The type of all such uses, however, must be the same. If a formal parameter name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal parameter, never to the other usage.

Within the expression expr, references to variables, formal parameters, other functions, array elements, and constants are permitted. Statement function references, however, must refer to statement functions defined prior to the statement function in which they appear. Statement functions cannot be called recursively, either directly or indirectly.

A statement function may be referenced only in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement which may not define that name as an array, and in a COMMON statement as the name of a common block. A statement function cannot be of type character.

Example:

```
C EXAMPLE OF STATEMENT FUNCTION STATEMENT
      DIMENSION X(10)
      ADD(J, K) = J + K
C
      DO 1, I=1, 10
      X(I) = ADD(I*10,I+2)
1     CONTINUE
```

STOP Statement

Purpose: The STOP statement causes the program to terminate. The argument, *n*, if present, is displayed upon termination.

Format: STOP [*n*]

Remarks: The *n* entry is either a character constant or a string of not more than five digits. If n is not present, the following message appears:

Stop – Program terminated.

If *n* is present, then *n* is displayed.

Example:

```
C EXAMPLE OF STOP STATEMENT
      IF (IERROR .EQ. 0) GOTO 200
      STOP 'ERROR DETECTED'
200   CONTINUE
```

SUBROUTINE
Statement

Purpose: The SUBROUTINE statement identifies a program unit as a subroutine, names the program unit, and identifies the formal parameters to that subroutine.

Format: SUBROUTINE *sname* [(*fparm* [, *fparm* . . .)]

Remarks: The *sname* entry is the user-defined name of the subroutine.

The *fparm* entry is the user-defined name of a formal parameter. *sname* is a global name (and it is also local to the subroutine it names). The list of formal parameter names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of actual arguments to that subroutine. Formal parameter names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements. A subroutine may return values by assignment to one or more of its formal parameters.

Example:

```
SUBROUTINE CONVRT (RAD,DEG)
DEG=RAD*180.0/3.14159
RETURN
END
```

Type Statement

Purpose: A type statement specifies the type of user-defined names. A type statement can confirm or override the implicit type of a name. A type statement can also specify dimension information.

Format: type *v*[,*v*]. . .

Remarks: The *type* entry is one of the data types (see "DATA TYPES" in Chapter 1).

A user name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name can have its type explicitly specified by a type statement only once. The name of a subroutine or main program cannot appear in a type statement. The *v* entry is the symbolic name of a variable, array, function, function subprogram, or an array declarator.

The following rules apply to a type declaration:

1. A type declaration statement must precede all executable statements.

2. The data type of a symbolic name can be declared only once.

3. A type declaration statement cannot be labeled.

4. A type declaration statement can be used to declare an array by appending an array declarator to an array name.

The *v* entry can be followed by a data type length specifier of the form **n*, where *n* is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If both a data-type length specifier and an array declarator are specified, the data type length specifier goes last. Examples of type declaration statements are:

Example:

```
C EXAMPLE OF TYPE STATEMENTS

      CHARACTER NAME*10, CITY*80, CH
      INTEGER COUNT, MATRIX(4,4), SUM
      REAL MAN,IABS
      LOGICAL SWITCH
      INTEGER*2 Q, M12*4, IVEC(10)*4
```

Unconditional GOTO

Purpose: The Unconditional GOTO statement causes the next statement executed to be the statement labeled *s*.

Format: GOTO *s*

Remarks: The *s* entry is a statement label of an executable statement found in the same program unit as the GOTO statement. Jumping into a DO, IF, ELSEIF, or ELSE block from outside the block is not permitted. (A special feature, extended range DO loops, permits jumping into a DO block. See the $DO66 metacommand in Chapter 3 for more information.)

Example:

```
C EXAMPLE OF UNCONDITIONAL GOTO
      COUNT=0
      DO 10 I=1,10
        COUNT=COUNT+1
        IF (COUNT.GE.10) GO TO 20
10    CONTINUE
20    WRITE(*,100) COUNT
      STOP
100   FORMAT (1X,'COUNT=',F10)
      END
```

WRITE Statement

Purpose: The WRITE statement transfers the iolist items to the unit specified.

Format: WRITE(*u*[,*f*] [,ERR=*s*] [,REC=*rn*])*iolist*

Remarks: The *u* entry is a unit specifier (see "Elements of I/O Statement" previously described). It is required, and must appear as the first argument.

The *f* entry is required for formatted write as the second argument, and must not be used for unformatted write.

The *s* entry is an optional statement label. If it is not present, I/O errors result in runtime errors. If it is present, I/O errors cause control to transfer to the executable statement labeled s.

The *rn* entry is specified for direct access only, otherwise an error results. It is a positive integer expression. It positions to record number *rn* for this WRITE. If REC=*rn* is omitted for a direct access file, writing continues from the current position in the file.

If the write is internal, the character variable or character array element specified is the destination of the output, otherwise the external unit is the destination.

Example: EXAMPLE 1 PROGRAM FRAGMENT

```
C DISPLAY MESSAGE: "ONE = 1, TWO = 2,
C THREE = 3" ON THE DISPLAY, NOT DOING
C THINGS IN THE SIMPLEST WAY
      WRITE(*,980)'ONE=',1,1+1,'ee=',+(1+1+1)
 980  FORMAT(A,I2',TWO=',1X,I1,',THR',A,I1)
```

EXAMPLE 2 PROGRAM FRAGMENT

```
C DECLARE THE DATA AREAS
      CHARACTER*15 ITEM(10)
      REAL COST(10)
      INTEGER MIN,MAX,VOLUME(10)
C OPEN THE FILES
      OPEN(6,FILE='PRN')
      OPEN(1,FILE='INVENTRY.DAT',STATUS='NEW',
       FORM='UNFORMATTED')
C READ IN THE DATA FROM THE KEYBOARD
      DO 10 I=1,10
        WRITE(*,110)I
110     FORMAT(' ENTER ITEM #',I2,': '\)
        READ(*,100) ITEM(I)
100     FORMAT(A)
        WRITE(*,210) ITEM(I)
210     FORMAT(' ENTER COST AND VOLUME FOR ',A)
        READ(*,200) COST(I),VOLUME(I)
        IF (COST(I) .EQ. 0.0) GO TO 20
200     FORMAT(BN,F10.2,I10)
C WRITE THE DATA TO AN UNFORMATTED FILE
        WRITE(1) ITEM(I),COST(I),VOLUME(I)
10    CONTINUE
20    I=I-1
C FIND THE MIN AND MAX OF THE DATA
      MIN=VOLUME(1)
      MAX=VOLUME(1)
      DO 30 J=2,I
        IF (VOLUME(J) .GT. MAX) MAX=VOLUME(J)
        IF (VOLUME(J) .LT. MIN)  MIN=VOLUME(J)
30    CONTINUE
C REPORT THE RESULTS TO THE PRINTER
      WRITE(6,220)
220   FORMAT('1ITEM',22X,'COST',9X,'VOLUME',6X,'INVENTORY')
      WRITE(6,230)
```

```
230     FORMAT('+','____',22X,'____',9X,'______',6X,'_________')
        DO 40 J=1,I
          WRITE(6,400) ITEM(J),COST(J),VOLUME(J),COST(J)*VOLUME(J)
40      CONTINUE
400     FORMAT(1X,A,5X,F10.2,5X,I10,5X,F10.2)
        WRITE(6,600) MIN,MAX
600    0FORMAT('0VOLUME MAXIMUM = ',I10,
       1        /,' VOLUME MINIMUM = ',I10)
        END
```

CHAPTER 5. I/O SYSTEM

Contents

This chapter describes the Fortran I/O system, including the basic Fortran I/O concepts, statements, and the FORMAT statement.

The major subsections of this section are:

- Overview—Provides an overview of the Fortran file system. Defines the basic concepts of I/O records, I/O units, and the various kinds of file access available.

- Concepts and Limitations—Relates the definitions made in the Overview to accomplishing various tasks using the most common forms of files and I/O statements. Gives a complete program illustrating them. There is a general discussion of I/O system limitations.

- FORMATTED I/O and the FORMAT Statement.

Overview

You need to be familiar with the terms and concepts related to the structure of the Fortran I/O system to understand the I/O statements.

Records

The building block of the Fortran file system is the record. A record is a sequence of characters or values. There are three kinds of records:

1) Formatted

2) Unformatted

3) Endfile

Formatted

Formatted records are sequences of characters terminated by the carriage return–linefeed. Formatted records are interpreted on input consistently with the way the IBM Personal Computer DOS interprets characters. They are useful when the source data is to be read by a user or used as input to another program. Formatted records must be used with the Formatted I/O statements. They are the most common type of record and are required when writing or reading to or from the display/keyboard or printer.

Unformatted

Unformatted records are sequences of values, with no system alteration or interpretation; no physical representation exists for the end of record. They are used when it is desired to store or retrieve information without the need for editing or user intervention.

Endfile

The Fortran file system simulates a virtual *endfile record* after the last record in a file, although there is no corresponding real record.

Files

A *file* is a sequence of records. Files are either external or internal.

An external file is a file on a device or a device itself. An internal file is a character variable that serves as the source or destination of some I/O action. From this point on, both internal Fortran files and the files known to DOS are referred to simply as files, with context determining meaning. (The OPEN statement provides the linkage between the two notions of files and, in most cases, the ambiguity disappears after opening a file, when the two notions coincide.)

File Properties

A Fortran file has these properties:

- Name
- Position
- Formatted, unformatted, or binary
- Sequential or direct access

File Name

A file can have a name. If present, a name is a character string identical to the name by which it is known to DOS (see the *IBM Personal Computer DOS* reference manual for more information on filename structures). The filename may also be blank which allows specifying the filename at runtime. See the OPEN statement in Chapter 4.

In addition to the IBM Personal Computer DOS filenames, the following two special filenames exist:

- USER (non-buffered display/keyboard I/O)
- LINE (non-buffered RS-232 I/O)

See "Device Identifications" in Chapter 2.

File Position

The position property of a file is usually set by the previous I/O operation. A file has an initial point, terminal point, current record, preceding record, and next record.

It is possible to be between records in a file, in which case the next record is the successor to the previous record and there is no current record.

Opening a sequential file for writing, positions the file at its beginning and discards all old data in the file. The file position after sequential writes is at the end of the file, but not beyond the endfile record. Executing the ENDFILE statement positions the file beyond the endfile record, as does a READ statement executed at the end of the file (but not beyond the endfile record). Reading an endfile record can be detected by the user using the END= option in a READ statement.

Formatted, Unformatted, and Binary Files

An external file is opened as either formatted, unformatted, or binary. All internal files are formatted. Formatted files consist entirely of formatted records. Unformatted and binary files consist entirely of unformatted records.

Sequential and Direct Access Properties

An external file is opened as either sequential or direct.

Sequential files contain records with order determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC= option which specifies a position for direct access I/O. DOS attempts to extend sequential access files if a record is written beyond the old terminating file boundary; the success of this depends on the existence of room on the physical device.

Direct access files can be read or written in any order (they are random access files).

Records are numbered sequentially, with the first record numbered 1. All records have the same length, specified when the file is opened, and each record has a unique record number, specified when the record is written.

It is possible to write the records out of order, including, for example, writing record 9, 5, and 11 in that order without writing the records in between. It is not possible to delete a record once written, but a record can be overwritten with a new value. Direct access files must reside on diskette. The values of records never written are undefined.

Internal Files

Internal files provide a mechanism for using the formatting capabilities of the I/O system to convert values to and from their external character representations, within the Fortran internal storage structures. That is, reading a character variable may be done to convert the character values into numeric, logical, or character values and writing a character variable allows values to be converted into their (external) character representation.

Special Properties

An internal file is a character variable or character array element. The file has exactly one record, which has the same length as the character variable or character array element. If less than the entire record is written, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to the I/O statement execution. Only formatted, sequential I/O is permitted to internal files and only the I/O statements READ and WRITE may specify an internal unit.

```
C     EXAMPLE OF INTERNAL FILE I/O
      CHARACTER*11 A,B
      DATA A /'1.234-86.45'/
      READ(A,2) X,Y
2     FORMAT(F5.3,F6.2)
      Z=X+Y
      WRITE(B,'(F6.2)')Z
      WRITE(*,'(1X,F6.2)')Z
      END
```

Units

A unit is a means of referring to a file. A unit specified in an I/O statement is either an external unit or an internal file specifier. An external unit specifier is either an integer expression or the character *, which stands for the display (for writing) and the keyboard (for reading). In most cases, an external unit specifier value is bound to a physical device (or file resident on the device) by name using the OPEN statement. Once this binding of value to system file name occurs, Fortran I/O statements specify the unit number as a means of referring to the appropriate external entity. Once opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE. The only exception to these binding rules is that the unit value 0 is permanently associated with the keyboard for reading and the display for writing and no explicit OPEN is necessary. The character * is interpreted by the Fortran file system as specifying unit 0. An internal file specifier is a character variable or character array element that directly specifies an internal file.

Concepts and Limitations

Fortran provides a rich combination of possible file structures. However, two kinds of files suffice for most applications: * files, and explicitly opened external, sequential, formatted files.

An asterisk (*) file represents the keyboard and display: a sequential, formatted file, also known as unit 0. This particular unit has the special properties that an entire line, terminated by the Enter key, must be entered when reading from it, and the backspace and Escape keys familiar to the IBM Personal Computer user serve their normal functions.

Explicitly Opened External, Sequential, Formatted Files

These files are bound to a system file by name in an OPEN statement.

Example

This example program uses the two kinds of files discussed above for reading and writing. The I/O statements are explained in detail in the following subsection. Copy a file with three columns of integers, each 7 columns wide, from a file whose name is input by you to another file named OUT.TEXT, reversing the positions of the first and second columns.

```
      PROGRAM COLSWP
      CHARACTER*64 FNAME
C Prompt to the display by writing to *.
      WRITE(*,900)
900   FORMAT(' INPUT FILE NAME - ')
C Read the file name from the keyboard by
C reading from *.
      READ(*,910) FNAME
910   FORMAT(A)
C Use unit 3 for input; any unit number except
C 0 will do.
      OPEN(3,FILE=FNAME)
C Use unit 4 for output; any unit number except
C 0 and 3 will do.
      OPEN(4,FILE='OUT.TXT',STATUS='NEW')
C Read and write until end of file.
100   READ(3,920,END=200)I,J,K
      WRITE(4,920)J,I,K
920   FORMAT(3I7)
      GOTO 100
200   WRITE(*,910)' Done'
      END
```

Less Commonly Used File Operations

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them follows. If the I/O is to be random access (direct), such as in maintaining a data base, direct access files are probably necessary.

If the data is to be written and read by IBM Fortran, unformatted files are perhaps more efficient in speed. The combination of direct and unformatted is ideal for a data base to be created, maintained, and accessed exclusively by IBM Fortran. If the data must be transferred without any system interpretation, especially if all 256 possible byte values are to be transferred, unformatted I/O is necessary. Internal files are not I/O in the conventional sense but rather provide certain character string operations and conversions within a standard mechanism. A file opened in IBM Fortran is either "old" or "new" but there is no concept of "opened for reading" as distinguished from "opened for writing."

Therefore, you can open "old" (existing) files and write to them, with the effect of overwriting them. Similarly, you can alternately write to and read from the same file (providing that you avoid reading beyond the end of the file, or reading unwritten records in a direct file). A write to a sequential file effectively deletes any records that existed beyond the newly written record. Normally, when a device (such as the keyboard or printer) is opened as a file, it makes no difference whether it is opened as "old" or "new." With disk files, however, opening "new" creates a new file. If that file is closed or if the program terminates without doing a CLOSE on that file, a permanent file is created with the name given when the file was opened. If a previous file existed with the same name, it is deleted.

Direct Files/Direct Device Association

There are two kinds of devices: sequential and direct. The files associated with sequential devices are streams of characters, with no explicit motion allowed except reading and/or writing. The keyboard, display, and printer are examples of sequential devices. Direct devices have the additional operation of seeking a specific location. They can be accessed either sequentially or randomly, and thus can support direct files. The Fortran I/O system does not allow direct files on sequential devices.

BACKSPACE/Sequential Device Association

The IBM Fortran I/O system does not allow backspacing a file on a sequential device such as keyboard, display or printer.

BACKSPACE/Unformatted Sequential File Association

Record boundaries are not indicated in an unformatted sequential file; therefore, BACKSPACE on such files is defined as backing up by one byte. Direct files contain records of fixed, specified length, so it is possible to backspace by records on direct unformatted files.

Functions Called in I/O Statements

During execution of any I/O statement, evaluation of an expression may cause a function to be called. That function call must not cause any I/O statement to be executed.

Partial Read/Unformatted Sequential File Association

Record boundaries are not indicated in an unformatted sequential file; therefore, a read of a record which reads only part of the record will not position the file at the beginning of the next record. The binary file should be used when taking advantage of this limitation. Binary files define BACKSPACE as backing up 1 byte.

Formatted I/O and the FORMAT Statement

Format Specifications and the FORMAT Statement

If a READ or WRITE statement specifies a format, it is considered a formatted, rather than an unformatted I/O statement. Such a format can be specified in one of three ways, as explained in "Elements of I/O Statements" in Chapter 4. There are several ways to refer to FORMAT statements and one is an immediate format in the form of a character expression containing the format itself. The following four examples are all valid and equivalent means of specifying a format:

```
1)          WRITE (*,990) I,J,K
      990   FORMAT(1X,2I5,I3)

2)          ASSIGN 990 TO IFMT
      990   FORMAT('  ',2I5,I3)
            WRITE(*,IFMT) I,J,K

3)          WRITE(*,'(I6,I5,I3)')I,J,K

4)          CHARACTER*8 FMTCH
            FMTCH = '(1H ,2I5,I3)'
            WRITE(*,FMTCH)I,J,K
```

The format specification itself must begin with "(", possibly following initial blank characters, and end with a matching ")". Characters beyond the matching ")" are ignored.

FORMAT statements must be labeled, and like all nonexecutable statements, may not be the target of a branching operation.

Between the initial "(" and terminating ")" is a list of items, separated by commas, each of which is one of:

[*r*] *ed*	repeatable edit descriptors.
ned	nonrepeatable edit descriptors.
[*r*] *fs*	a nested format specification. At most, three levels of nested parentheses are permitted within the outermost level.

The r entry is an optionally present, nonzero, unsigned, integer constant called a repeat specification.

The comma separating two list items may be omitted if the resulting format specification is still unambiguous, such as after a P edit descriptor or before or after the / edit descriptor.

Repeatable Edit Descriptors

The repeatable edit descriptors, described below, are:

I*w*
F*w*.d
E*w*.d
E*w*.dEe
L*w*
A
A*w*

The I, F, E, L, A entries indicate the manner of editing.

The *w* and *e* entries are nonzero, unsigned, integer constants.

The *d* entry is an unsigned integer constant.

Nonrepeatable Edit Descriptors

The nonrepeatable edit descriptors are as follows:

'*xxxx*'	character constants of any length (see special rules below).
*n*H*xxxx*	another means of specifying character constants (see rules below).
*n*X	denotes positional editing
/	denotes slash editing
\	denotes backslash editing
*k*P	denotes scale factor
BN	denotes blank interpretation
BZ	denotes blank interpretation

The ', H, X, /, ?, P, BN, BZ entries indicate the manner of editing.

The *x* entry is any ASCII character.

The *n* entry is a nonzero, unsigned, integer constant.

The *k* entry is an optionally signed integer constant.

Input/Output List Interaction and Format Specification

Input/Output List

If an *iolist* contains at least one item, at least one repeatable edit descriptor must exist in the format specification. In particular, the empty edit specification, (), can be used only if no items are specified in the iolist (in which case the only action caused by the I/O statement is the implicit record-skipping action associated with formats).

Each item in the iolist is associated with a repeatable edit descriptor during the I/O statement execution in turn. In contrast, the remaining format control items interact directly with the record and do not become associated with items in the iolist.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present *r* times (if omitted, *r* is treated as a repeat factor of 1). Similarly, a nested format specification is treated as if its items appeared *r* times.

The formatted I/O process proceeds as follows. The "format controller" scans the format items in the order indicated above. When a repeatable edit descriptor is encountered either:

- A corresponding item appears in the iolist, in which case the item and the edit descriptor are associated and I/O of that item proceeds under format control of the edit descriptor.

- No corresponding item appears in the iolist, in which case the "format controller" terminates I/O.

If the format controller encounters the matching final ")" of the format specification and there are no further items in the iolist, the "format controller" terminates I/O. If, however, there are further items in the iolist, the file is positioned at the beginning of the next record and the "format controller" continues by rescanning the format starting at the beginning of the format specification terminated by the last preceding right parenthesis.

If there is no such preceding right parenthesis, the "format controller" rescans the format from the beginning. Within the portion of the format rescanned, there must be at least one repeatable edit descriptor.

If the rescan of the format specification begins with a repeated nested format specification, the repeat factor indicates the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or BN or BZ blank control in effect. When the "format controller" terminates, the remaining characters of an input record are skipped or an end of record is written on output, except as noted under "Edit Descriptor" below.

Edit Descriptors

Nonrepeatable

Apostrophe Editing.

The apostrophe edit descriptor ('*xxxx*') has the form of a character constant. Embedded blanks are significant and double 'are interpreted as a single' within a character constant. Apostrophe editing cannot be used for input (READ) as it causes the character constant to be transmitted to the output unit. For an example, see "H (Hollerith Editing)" below.

H (Hollerith Editing).

The *n*H edit descriptor causes the following *n* characters, with blanks counted as significant, to be transmitted to the output unit. Hollerith editing cannot be used for input (READ).

EXAMPLES OF APOSTROPHE AND HOLLERITH EDITING

```
C Each write outputs characters between the
C slashes:  /ABC'DEF/
      WRITE (*,970)
970   FORMAT ('ABC''DEF')
      WRITE (*,'("ABC''''DEF")')
      WRITE (*,'(1X,7HABC''DEF)')
      WRITE (*,960)
960   FORMAT (8HABC'DEF)
```

X (Positional Editing).

On input (READ), the *n*X edit descriptor causes the file position to advance *n* characters, thus the next n characters are skipped. On output (WRITE), the *n*X edit descriptor causes n blanks to be written, providing that further writing to the record occurs; otherwise, the *n*X descriptor results in no operation.

/ (Slash Editing).

The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end of record is written and the file is positioned to write on the beginning of the next record.

\ (Backslash Editing).

Normally when the "format controller" terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the "format controller" is I, this automatic end of record is inhibited. This allows subsequent I/O statements to continue reading (or writing) out of (or into) the same record. The most common use for this mechanism is to prompt to the display and read a response off the same line as in:

```
WRITE (*,'(A\)') ' Input an integer → '
READ (*,'(BN,I6)') I
```

The \ edit descriptor does not inhibit the automatic end of record generated when reading from the * unit. Input from the keyboard must always be terminated by the Enter key. This permits the backspace and the Control-X keys to function properly.

P (Scale Factor Editing).

The P edit descriptor sets the scale factor for subsequent F and E edit descriptors until another *k*P edit descriptor is encountered. At the start of each I/O statement, the scale factor is initialized to 0. The scale factor affects format editing in the following ways:

- On input, with F and E editing, providing that no explicit exponent exists in the field, and F output editing, the externally represented number equals the internally represented number multiplied by 10**k.

- On input, with F and E editing, the scale factor has no effect if there is an explicit exponent in the input field.

- On output, with E editing, the real part of the quantity is output multiplied by 10**k and the exponent is reduced by k (effectively altering the column position of the decimal point but not the value output).

BN and BZ (Blank Interpretation).

These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If a BN edit descriptor is processed by the "format controller," blanks in subsequent input fields are ignored unless, and until, a BZ edit descriptor is processed. The effect of ignoring blanks is to take all the nonblank characters in the input field, and treat them as if they were right-justified in the field with the number of leading blanks equal to the number of ignored blanks. For instance, the following READ statement accepts the characters shown between the slashes as the value 123 (where <cr> indicates pressing the Enter key):

```
      READ(*,100) I
100   FORMAT (BN,I6)

      /123      <cr>/,
      /123   456<cr>/,
      /   123<cr>/.
```

The BN edit descriptor, in conjunction with the infinite blank padding at the end of formatted records, makes interactive input very convenient.

Repeatable

I, F, and E (Numeric Editing).

The I, F, and E edit descriptors are used for I/O of integer and real data. The following general rules apply to all three of them:

- On input, leading blanks are not significant. Other blanks are interpreted differently depending on the BN or BZ flag in effect, but all blank fields always become the value 0. Plus signs are optional.
- On input, with F and E editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.
- On output, the characters generated are right-justified in the field with padding by leading blanks if necessary.
- On output, if the number of characters produced exceeds the field width or the exponent exceeds its specified width, the entire field is filled with asterisks.

I (Integer Editing).

The edit descriptor I*w* must be associated with an *iolist* item of type integer. The field is *w* characters wide. On input, an optional sign may appear in the field.

F (Real Editing).

The edit descriptor F*w.d* must be associated with an iolist item of type real. The field is *w* characters wide, with a fractional part *d* digits wide. The input field begins with an optional sign followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the *d* specified in the edit descriptor; otherwise the rightmost *d* digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros if necessary). Following this is an optional exponent which is either:

- + (plus) or – (minus) followed by an integer,
- E followed by zero or more blanks followed by an optional sign followed by an integer.

The output field occupies *w* digits, *d* of which fall beyond the decimal point and the value output is controlled both by the iolist item and the current scale factor. The output value is rounded rather than truncated.

E (Real Editing).

An E edit descriptor takes either the form E*w.d* or E*w.d*E*e*. In either case the field is *w* characters wide. The *e* has no effect on input. The input field for an E edit descriptor is identical to that described by an F edit descriptor with the same *w* and *d*. The form of the output field depends on the scale factor (set by the P edit descriptor) in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent exp, of one of the following forms:

E*w.d*	-99 <= exp <= 99	E followed by plus or minus followed by the two-digit exponent.
E*w.d*	-999 <=exp <= 999	Plus or minus followed by the three-digit exponent.
E*w.d*E*e*	-((10**e) - 1) <= exp <= (10**e) -1	E followed by plus or minus followed by e digits which are the exponent with possible leading zeros.

The form Ew.d must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale factor controls the decimal normalization of the printed E field. If the scale factor, k, is in the range $-d < k <= 0$, then the output field contains exactly $-k$ leading zeros after the decimal point and d + k significant digits after this. If $0 < k < d + 2$, then the output field contains exactly k significant digits to the left of the decimal point and $d - k - 1$ places after the decimal point. Other values of k are errors.

L (Logical Editing)

The edit descriptor L*w* indicates that the field is *w* characters wide. The iolist element associated with an L edit descriptor must be of type logical. On input, the field consists of optional blanks, followed by an optional decimal point, followed by T (for TRUE) or F (for FALSE). Any further characters in the field are ignored, but accepted on input, so that TRUE and FALSE are valid inputs. On output, *w* – 1 blanks are followed by either T or F as appropriate.

A (Character Editing)

The forms of the A edit descriptor are A or A*w*, in which the former acquires an implied field width, *w*, from the number of characters in the iolist item with which it is associated. The iolist item must be of type character if it is to be associated with an A or A*w* edit descriptor. On input, if *w* exceeds or equals the number of characters in the iolist element, the rightmost characters of the input field are used as the input characters; otherwise the input characters are left-justified in the input iolist item and trailing blanks are provided. On output, if *w* exceeds the characters produced by the iolist item, leading blanks are provided; otherwise, the leftmost *w* characters of the item are output.

Carriage Control

The first character of every record transferred to the printer or display is not printed. Instead, it is interpreted as a carriage control character. The IBM Fortran I/O system recognizes certain characters as carriage control characters. These characters and their effects are as follows:

Character	Effect
space	Advances one line
0	Advances two lines
1	Advances to top of next page
+ (plus)	Does not advance (allows overprinting)

Any character other than those listed above is treated as a space and is deleted from the print line.

> **Note:** If you accidentally omit the carriage control character, the first character of the record is not printed.

CHAPTER 6. INTRINSIC FUNCTIONS

Contents

Intrinsic Functions

An intrinsic function is predefined by the Fortran compiler and available for use in a Fortran program. The table below gives the name, definition, number of parameters, and type of the intrinsic functions available in IBM Fortran 77. An IMPLICIT statement does not alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

An intrinsic function name can appear in an INTRINSIC statement, but only those intrinsic functions listed in the table below may do so. An intrinsic function name also can appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function being computed. For example, the logarithm of a negative number is mathematically undefined, and therefore not permitted.

In the table below all angles are expressed in radians. All arguments in an intrinsic function reference must be of the same type. X and Y are real, I and J integer, and C, C1, and C2 character values. Footnotes indicated by (n), listed in the definition column, refer to the notes at the end of the table.

Name	Definition	Argument	Function
INT(X)	Conversion to integer (1)	Real	Integer
IFIX(X)	Conversion to integer (1)	Real	Integer
REAL(I)	Conversion to real (2)	Integer	Real
FLOAT(I)	Conversion to real (2)	Integer	Real
ICHAR(C)	Conversion to integer (3)	Character	Integer
CHAR(I)	Conversion to character	Integer	Character
AINT(X)	Truncation to real	Real	Real
ANINT(X)	Rounding to real whole number (1)	Real	Real
NINT(X)	Rounding to integer (1)	Real	Integer
IABS(I)	Integer absolute	Integer	Integer
ABS(X)	Real absolute	Real	Real
MOD(I,J)	Integer remainder (1)	Integer	Integer
AMOD(X,Y)	Real remainder (1)	Real	Real

Name	Definition	Argument	Function
ISIGN(I,J)	Integer transfer	Integer	Integer
SIGN(X,Y)	Real transfer	Real	Real
IDIM(I,J)	Integer difference (4)	Integer	Integer
DIM(X,Y)	Real difference (4)	Real	Real
MAX0(I,J,...)	Integer maximum	Integer	Integer
AMAX1(X,Y,...)	Real maximum	Real	Real
AMAX0(I,J,...)	Real maximum	Integer	Real
MAX1(X,Y,...)	Integer maximum	Real	Integer
MIN0(I,J,...)	Integer minimum	Integer	Integer
AMIN1(X,Y,...)	Real minimum	Real	Real
AMIN0(I,J,...)	Real minimum	Integer	Real
MIN1(X,Y,...)	Integer minimum	Real	Integer
SQRT(X)	Square root	Real	Real
EXP(X)	Real e raised to power	Real	Real

Name	Definition	Argument	Function
ALOG(X)	Natural logarithm of real argument	Real	Real
ALOG10(X)	Common logarithm of real argument	Real	Real
SIN(X)	Real sine	Real	Real
COSIN(X)	Real cosine	Real	Real
TAN(X)	Real tangent	Real	Real
ASIN(X)	Real arc sine	Real	Real
ACOS(X)	Real arc cosine	Real	Real
ATAN(X)	Real arc tangent	Real	Real
ATAN2(X/Y)	Real arc tangent of X/Y	Real	Real
SINH(X)	Real hyperbolic sine	Real	Real
COSH(X)	Real hyperbolic cosine	Real	Real
TANH(X)	Real hyperbolic tangent	Real	Real

Name	Definition	Argument	Function
LGE(C1,C2)	First argument greater than or equal to second (6)	Character	Logical
LGT(C1,C2)	First argument greater than second (5)	Character	Logical
LLE(C1,C2)	First argument less than or equal to second (5)	Character	Logical
LLT(C1,C2)	First argument less than second (5)	Character	Logical
EOF(I)	Integer end of file (6)	Integer	Logical

Notes:

1. For X of type real, if X> = 0, then INT(X) is the largest integer not greater than X, and if X < 0, then INT(X) is the most negative integer not less than X. IFIX(X) is the same as INT(X).

2. For I of type integer, REAL(I) is as much precision of the significant part of I as a real value can contain. FLOAT(I) is the same as REAL(I).

3. ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 255. For any two characters, C1 and C2, (C1 .LE. C2) is true if and only if (ICHAR(C1) .LE. ICHAR(C2)) is true.

4. IDIM and DIM are defined as the actual difference if that number is positive and 0 otherwise.

5. LGE(C1,C2) returns the value true if C1 = C2 or if C1 follows C2 in the ASCII collating sequence; otherwise it returns false. LGT(C1,C2) returns true if C1 follows C2 in the ASCII collating sequence; otherwise it returns false. LLE(C1,C2) returns true if C1 = C2 or if C1 precedes C2 in the ASCII collating sequence; otherwise it returns false. LLT(C1,C2) returns true if C1 precedes C2 in the ASCII collating sequence; otherwise it returns false. The operands of LGE, LGT, LLE, and LLT must be of the same length.

6. EOF(I) returns the value true if the unit specified by its argument is at or past the end of file record; otherwise it returns false. The value of I must correspond to an open file, or to 0 which indicates the display or keyboard device.

APPENDIXES

Contents

APPENDIX A. MESSAGES

Error conditions may be undetected, detected by the compiler, or detected by the runtime system. This appendix gives error messages and codes for errors detected by the compiler and runtime system. Compiler errors are divided into front end (FOR1) errors and back end (FOR2) errors; runtime errors are divided into file system errors and all other errors.

Compile-Time Error Messages

Front End Errors

Front end error messages include a number as well as a message. The front end recovers from most errors, but a few are called "panic" errors. These panic errors also give the message:

```
? ERROR: error message
```

Where error message can be:

- Filename error in file *fname* – Syntax error in filename
- Device full error in file *fname* – Diskette full
- File not found error in file *fname* – Cannot find file
- Stack Overflow – Compiler out of storage
- No room in heap – Compiler out of storage
- Out of memory
- Real Math Overflow

  Note: The *fname* entry is the name of the file in error.

In these cases it may be difficult to locate the error because line numbers are not given. The last line compiled can be located by directing the listing file to USER (see "Device Identification" in Chapter 2).

The error message **"Internal error"** refers to an internal consistency check which failed; no matter what source program is compiled, there should not be a way to get one of these messages.

1 Fatal error reading source block.
2 Nonnumeric characters in label field.
3 Too many continuation lines.
No more than nine continuation lines allowed for each initial line.
4 Fatal end of file encountered.
Unexpected end of file encountered while reading source file, for example: END statement missing, extra line(s) after END.
5 Labeled continuation line.
6 Missing field on $ compiler metacommand.
7 Cannot open file.
8 Unrecognizable metacommand.
9 Input file invalid format.
10 Too many nested include files.
11 Integer constant overflow.
12 Real constant error.
Incorrect representation of real constant.
13 Too many digits in constant.
14 Identifier too long.
15 Character constant not closed.
Closing apostrophe not found for character constant, for example: missing closing apostrophe, character constant extends past column 72.
16 Zero length character constant.
Zero length character constants not allowed.
17 Invalid character in input.
Character is not acceptable outside of Hollerith string or character constant.
18 Integer constant expected.
19 Label expected.
20 Label error.
21 Type expected, for example: in IMPLICIT statement.
22 Integer constant expected.
23 Extra characters at end of statement.
Characters encountered after expected end of line.
24 "(" expected.
25 Letter already used in IMPLICIT.
26 ")" expected.
27 Letter expected.
28 Identifier expected.
29 Dimension(s) expected.

30 Array already dimensioned.
31 Too many dimensions.
32 Incompatible arguments.
33 Identifier already has type.
34 Identifier already declared.
35 INTRINSIC FUNCTION not allowed here.
This INTRINSIC FUNCTION not allowed as an argument.
36 Identifier must be a variable.
37 Identifier must be a variable or the current FUNCTION.
38 "/" expected.
39 Named COMMON block already saved.
40 Variable already appears in a COMMON.
41 Variables in two different COMMON blocks.
42 Number of subscripts conflicts with declaration.
43 Subscript out of range.
44 Forces two calls to the same location.
One name cannot reference more than one call.
45 Forces location in negative direction.
EQUIVALENCE extends COMMON block in negative direction.
46 Forces location conflict.
47 Statement number expected.
48 CHARACTER and numeric items in same COMMON block.
49 CHARACTER and noncharacter item conflict.
50 Invalid symbol in expression.
51 SUBROUTINE name in expression.
52 INTEGER or REAL expected.
53 INTEGER, REAL or CHARACTER expected.
54 Types not compatible.
55 LOGICAL expression expected.
56 Too many subscripts.
57 Too few subscripts.
58 Variable expected.
59 "=" expected.
60 Size of CHARACTER items must agree.
61 Assignment types do not match.
62 SUBROUTINE name expected.
63 Dummy parameter not allowed.
Formal parameter not allowed in COMMON statement.

64	Dummy parameter not allowed. Formal parameter not allowed in EQUIVALENCE statement.
65	Assumed size declarations only for dummy arrays.
66	Adjustable size declarations only for dummy arrays.
67	Assumed size must be last dimension.
68	Adjustable bound must be parameter or in COMMON.
69	Adjustable bound must be simple integer variable.
70	More than one main program.
71	Size of named COMMON must agree.
72	Dummy arguments not allowed. Formal parameters not allowed in DATA statement.
73	COMMON variables not allowed. COMMON variables not allowed in DATA statement.
74	SUBROUTINE, FUNCTION, or INTRINSIC names not allowed.
75	Subscript out of range.
76	Repeat count must be >=1.
77	Constant expected.
78	Type conflict.
79	Number of variables does not match.
80	Label not allowed.
81	No such INTRINSIC FUNCTION.
82	INTRINSIC FUNCTION type conflict.
83	Letter expected.
84	FUNCTION type conflict with previous call.
85	SUBROUTINE/FUNCTION already defined.
87	Argument type conflict.
88	SUBROUTINE/FUNCTION conflict with previous use.
89	Unrecognizable statement.
90	CHARACTER FUNCTION not allowed.
91	Missing END statement.
93	Fewer actual arguments than dummy arguments in call.
94	More actual arguments than dummy arguments in call.
95	Argument type conflict.
96	SUBROUTINE/FUNCTION not defined.
98	CHARACTER size invalid.

100	Statement order. Statement out of order.
101	Unrecognizable statement.
102	Jump into block not allowed. Jump into IF, ELSEIF, or ELSE block not allowed. Jump into DO block not allowed without $DO66 metacommand specified.
103	Label already used for FORMAT.
104	Label already defined.
105	Jump to format not allowed.
106	DO statement not allowed here.
107	DO label must follow DO statement.
108	ENDIF not allowed here.
109	Matching IF missing.
110	Improperly nested DO block in IF block.
111	ELSEIF not allowed here.
112	Matching IF missing.
113	Improperly nested DO or ELSE block.
114	"(" expected.
115	")" expected.
116	THEN expected.
117	Logical expression expected.
118	ELSE not allowed here.
119	Matching IF missing.
120	GOTO not allowed here.
121	GOTO not allowed here.
122	Block IF not allowed here.
123	Logical IF not allowed here.
124	Arithmetic IF not allowed here.
125	"," expected.
126	Expression of wrong type.
127	RETURN not allowed here.
128	STOP not allowed here.
129	END not allowed here.
131	Label not defined.
132	DO or IF block not terminated.
133	FORMAT not allowed here.
134	FORMAT label already referenced.
135	FORMAT label missing.
136	Identifier expected.
137	Integer variable expected.
138	TO expected.
139	Integer expression expected.
140	ASSIGN statement missing.
141	Unrecognizable character constant.
142	Character constant expected.

143 Integer expression expected.
144 STATUS option expected.
145 Character expression not allowed.
Expression of wrong type, character expression expected.
146 FILE= missing.
147 RECL= already defined.
148 Integer expression expected.
149 Unrecognizable option.
150 RECL= missing.
151 Adjustable arrays not allowed here.
Adjustable arrays not allowed as I/O list elements.
152 End of statement encountered in implied DO, expressions beginning with "(" not allowed as I/O list elements.
153 Variable required as control for implied DO.
154 Expressions not allowed in I/O list.
Expressions not allowed in I/O list of a READ statement.
155 REC= option already defined.
156 Integer expression expected.
157 END= not allowed here.
158 END= already defined.
159 Unrecognizable I/O unit.
160 Unrecognizable format in I/O.
161 Options expected after ",".
162 Unrecognizable I/O list element.
163 FORMAT not found.
164 ASSIGN missing.
165 Label already used as FORMAT.
166 Integer variable expected.
167 Label defined more than once as FORMAT.
203 CHARACTER FUNCTION not allowed.
406 Unit zero must be formatted and sequential.
407 ERR= already defined.
408 Too many labels.
Too many labels specified in arithmetic IF.
409 Invalid size for this type.
411 Integer type conflict.
415 DIMENSION too big.
420 Invalid FUNCTION call.

421 Invalid INTRINSIC FUNCTION.
The names of intrinsic functions for type conversion, lexical relationship, and for choosing the largest or smallest value must not be used as actual arguments.

501 Unrecognizable character.

502 Blank not allowed in metacommand.

503 Metacommand not allowed here.

504 Size already defined.

601 Out of range.

701 CHARACTER type expected.

703 Internal error.

705 Internal error.

706 Internal error.

708 Internal error.

709 CHARACTER type not expected.

710 Internal error.

711 Internal error.

713 Long integer conversion error.

Back End Errors

There are two kinds of errors given by the back end (optimizer and code generator): user errors and internal errors. There are very few user errors; all are concerned with limitations that cannot be detected by the front end.

A large number of internal consistency checks are done in the back end, but naturally these should always be correct and never give an internal error. Both user and internal back end errors cause an immediate stop. Both give an error number and approximate listing line number.

Back End User Errors

1. Attempt to divide by zero. For example: A DIV 0.

2. Overflow during integer constant folding. For example: (Maximum + A + Integer).

3. Expression too complex/Too many internal labels. Try breaking up expression with intermediate value assigns.

Back End Internal Errors

These errors have the format:

*** Internal Error NNN

NNN is the internal error number, which ranges from 1 to 999. There is little that can be done when an internal error occurs, except report it to your authorized IBM Personal Computer dealer. Perhaps try changes to the program near the line where the error occurred.

File System Errors

Errors caught at runtime can be divided into file system errors and all other errors. File system errors will be described first. File system error codes range from 1000 to 1999. Codes from 1000 to 1299 are for IBM Fortran file system errors. These errors are given below:

File system errors all have the format:

error type error in file *filename*

followed by the error code.

The *error type* field is one of the following:

Operation
Filename
Device full
File not found
File not open
Data format
Line too long

File System Error Codes

1000	Write error when writing end of file.
1002	Filename extension with more than 3 characters.
1003	Error during creation of new file (disk/directory full).
1004	Error during open of existing file (file not found).
1005	Filename with zero or more than 8 characters.
1007	Total filename length over 21 characters.
1008	Write error when advancing to next record.
1009	File too big (over 65,535 logical sectors).
1010	Write error when seeking to direct record.
1011	Attempt to open a random file to a non-disk device.
1012	Forward space or backspace on a non-disk device.
1013	Disk or directory full error during forward space or back space.
1200	Format missing final ")".
1201	Sign not expected in input.
1202	Sign not followed by digit in input.
1203	Digit expected in input.
1204	Missing N or Z after B in format.
1205	Unexpected character in format.
1206	Zero repetition factor in format not allowed.
1207	Integer expected for w field in format.
1208	Positive integer required for w field in format.
1209	"." expected in format.
1210	Integer expected for d field in format.
1211	Integer expected for e field in format.
1212	Positive integer required for e field in format.
1213	Positive integer required for w field in A format.
1214	Hollerith field in format must not appear for reading.
1215	Hollerith field in format requires repetition factor.
1216	X field in format requires repetition factor.
1217	P field in format requires repetition factor.
1218	Integer appears before + or – in format.
1219	Integer expected after + or – in format.
1220	P format expected after signed repetition factor in format.
1221	Maximum nesting level for formats exceeded.
1222	")" has repetition factor in format.
1223	Integer followed by "," invalid in format.

1224 "." is invalid format control character.
1225 Character constant must not appear in format for reading.
1226 Character constant in format must not be repeated.
1227 "/" in format must not be repeated.
1228 "?" in format must not be repeated.
1229 BN or BZ format control must not be repeated.
1230 Attempt to perform I/O on unknown unit number, for example, OPEN statement missing or never executed.
1231 Formatted I/O attempted on file opened as unformatted.
1232 Format fails to begin with "(".
1233 I format expected for integer read.
1234 F or E format expected for real read.
1235 Two "." characters in formatted real read.
1236 Digit expected in formatted real read.
1237 L format expected for logical read.
1238 Blank logical field.
1239 T or F expected in logical read.
1240 A format expected for character read.
1241 I format expected for integer write.
1242 w field in F format not greater than d field + 1.
1243 Scale factor out of range of d field in E format.
1244 E or F format expected for real write.
1245 L format expected for logical write.
1246 A format expected for character write.
1247 Attempt to do unformatted I/O to a unit opened as formatted.
1251 Integer overflow on input.
1252 Not enough input to satisfy IOlist/format, for example, specifying an i10 format and only inputting 5 characters.
1253 Too many bytes written to direct access unit record.
1255 Attempt to do external I/O on a unit beyond end of file record.
1256 Attempt to position a unit for direct access on a nonpositive record number.
1257 Attempt to do direct access to a unit opened as sequential.
1258 Unable to seek to file position.
1260 Attempt to backspace unit connected to printer or keyboard/display.

1264 Attempt to do unformatted I/O to internal unit.

1265 Attempt to put more than one record into internal unit.

1266 Attempt to write more characters to internal unit than its length.

1267 EOF called on unknown unit.

1268 Dynamic file allocation limit exceeded.

1270 Console I/O error.

1271 EOF function called on printer or keyboard/display.

1272 File operation attempted after error encountered on previous operation.

1273 Keyboard buffer overflow: too many bytes written to keyboard input record (must be less than 132).

1274 Error while reading long integer. Specifying $STORAGE:2 will cause a more specific error to occur.

1275 Error while writing long integer. Specifying $STORAGE:2 will cause a more specific error to occur.

1297 Integer variable not currently assigned a format label.

1298 End of file encountered on read with no END= option.

1299 Integer variable not ASSIGNed a label used in assigned GOTO.

Other Runtime Errors

Non-file system error codes range from 2000 to 2999. In some cases metacommands control whether errors are checked; in other cases they are always checked. The metacommand controlling a check, if any, is given in the list below:

2000-2049 Memory Errors

The heap is the storage area that the IBM Fortran system uses to allocate storage dynamically. Since the stack and the heap grow toward each other, these errors are all related, for example, a stack overflow can cause a "Heap Is Invalid" error.

2000 Overflow.
While calling a procedure or function, the stack ran out of storage space.

2001 No Room In Heap.
While attempting to get dynamic storage, not enough room was found in the heap for a new variable. Always caught.

2002 Heap Is Invalid.
While attempting to get dynamic storage, the allocation algorithm discovered the heap structure is wrong. Always caught.

2050-2099 Integer Arithmetic

2052 Signed Divide By Zero.
INTEGER value divided by zero; check if $DEBUG.

2054 Signed Math Overflow.
INTEGER result outside maximum range; check if $DEBUG.

2084 Integer zero to Negative Power; always checked.

2100-2149 Type REAL Arithmetic

2100 REAL Divide By Zero.
REAL value divided by zero; always checked.

2101 REAL Math Overflow.
REAL value too large for representation; always checked.

2102 SIN Or COS Argument Range.
SIN or COS function argument is too large to get a meaningful result.

2103 EXP Argument Range.
EXP function in which argument is too large for result to fit in representation.

2104 SQRT Of Negative Argument.
Square root function on argument<zero; always caught.

2105 LN of Non-Positive Argument.
Natural log function on argument<=zero; always caught.

2106 TRUNC/ROUND Argument Range.
Converting a REAL outside the range of INTEGER; always caught.

2131 Tangent Argument Too Small.
Tangent function argument so small result invalid; always caught.

2132 ARCSIN Or ARCCOS Of REAL>1.0.
ARCSIN or ARCCOS argument greater than one; always caught.

2133 Negative Real To Real Power.
Attempt to raise a negative real to a real power; always caught.

2134 Real Zero to Negative Power; always checked.

2200-2249 Long Integer Arithmetic

2200 Long Integer Divided by Zero; always checked.

2201 Long Integer Math Overflow; always checked.

2234 Long Integer Zero to Negative Power; always checked.

2250-2999 Other Errors

2451 Assigned GOTO Label Not In List.

APPENDIX B. DIFFERENCES BETWEEN IBM FORTRAN AND ANSI FORTRAN 77

This appendix describes how IBM Fortran differs from the standard subset language. The standard defines two levels, full Fortran and subset Fortran. IBM Fortran is a superset of the latter. The differences between IBM Fortran and the standard subset Fortran fall into two general categories: full-language features, and extensions to standard.

Full-Language Features

Several features from the full-language are included in this implementation. In all cases, a program written to comply with the subset restrictions compiles and executes properly, since the full-language properly includes the subset constructs.

Subscript Expressions

The subset does not allow function calls or array element references in subscript expressions, but the full-language and this implementation do.

DO Variable Expressions

The subset restricts expressions that define the limits of a DO statement, but the full-language does not. IBM Fortran also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

Unit I/O Number

IBM Fortran allows an I/O unit to be specified by an integer expression, as does the full-language.

Expressions in Input/Output List (iolist)

The subset does not allow expressions to appear in an I/O list whereas the full-language does allow expressions in the I/O list of WRITE statements. IBM Fortran allows expressions in the I/O list of a WRITE statement, provided that they do not begin with an initial left parenthesis.

> **Note:** The expression (A+B)*(C+D) can be specified in an output list as +(A+B)*(C+D). Doing so does not generate any extra code to evaluate the leading +.

Expression in Computed GOTO

IBM Fortran allows an expression for the value of a computed GOTO, consistent with the full, rather than the subset, language.

Generalized I/O

IBM Fortran allows both sequential and direct access files to be either formatted or unformatted. The subset language restricts direct access files to be unformatted and sequential to be formatted. IBM Fortran also contains an augmented OPEN statement that takes additional parameters not included in the subset. There is also a form of the CLOSE statement, which is not included in the subset. I/O is described in more detail in Chapters 4 and 5. The READ and WRITE statements allow the optional ERR parameter.

Extensions to Standard

The implemented language has several minor extensions to the full-language standard. These are compiler metacommands, backslash edit control, and end of file intrinsic function.

Compiler Metacommands

Compiler metacommands were added to allow the programmer to communicate certain information to the compiler. An additional kind of line, called a compiler metacommand line, has been added. It is characterized by a dollar sign, $, appearing in column 1. A compiler metacommand line may appear any place that a comment line can appear, although certain metacommands are restricted to appear in certain places. A compiler metacommand line is used to convey certain compile-time information to the IBM Fortran system about the nature of the current compilation.

Backslash Edit Control

The edit control character can be used in formats to inhibit the normal advancement to the next record associated with the completion of a READ or a WRITE statement. This is particularly useful when prompting to an interactive device, such as the display, so that a response can be on the same line as the prompt.

End of File Intrinsic Function

An intrinsic function, EOF, is provided. The function accepts a unit specifier as an argument.

APPENDIX C. THE LINKER (LINK) PROGRAM

Introduction

The Linker (LINK) program is a program that:

- Combines separately produced object modules.
- Searches library files for definitions of unresolved external references.
- Resolves external cross-references.
- Produces a printable listing that shows the resolution of external references and error messages.
- Produces a relocatable load module.

You will learn how to start LINK at the end of this appendix. You should read all of this appendix before you start LINK.

Files

The linker processes the following input, output, and temporary files:

Input Files

Type	Default .ext	Override .ext	Produced by
Object	.OBJ	Yes	Compiler (1) or MACRO Assembler
Library	.LIB	Yes	Compiler
Automatic Response	(none)	N/A (2)	User

Notes:

1. One of the optional compiler packages available for use with the IBM Personal Computer DOS.

2. N/A – Not applicable.

Output Files

Type	Default .ext	Override .ext	Used by
Listing	.MAP	Yes	User
Run	.EXE	No	Relocatable loader (COMMAND.COM)

VM.TMP (Temporary File)

LINK uses as much storage as is available to hold the data that defines the load module being created. If the module is too large to be processed with the available amount of storage, the linker will need additional storage space. If this happens, a temporary diskette file called VM.TMP is created on the DOS default drive.

A message is displayed to indicate when the overflow to diskette has begun. Once this temporary file is created, you should not remove the diskette until LINK ends. When LINK ends, the VM.TMP file is deleted.

If the DOS default drive already has a file by the name of VM.TMP, it will be deleted by LINK and a new file will be allocated. The contents of the previous file is destroyed; therefore, you should avoid using VM.TMP as one of your own filenames.

Definitions

Segment, *group*, and *class* are terms that appear in this appendix and in some of the messages at the end of this appendix. These terms describe the underlying function of LINK. An understanding of the concepts that define these terms provides a basic understanding of the way LINK works.

Segment

A *segment* is a contiguous area of storage up to 64K bytes in length. A segment may be located anywhere in storage on a *paragraph* (16-byte) boundary. Each of the four segment registers defines a segment. The segments can overlap. Each 16-bit address is an offset from the beginning of a segment. The contents of a segment are addressed by a segment register/offset pair.

The contents of various portions of the segment are determined when machine language is generated.

Neither size nor location is necessarily fixed by the machine language generator, because this portion of the segment may be combined at link time with other portions forming a single segment.

A program's ultimate location in storage is determined at load time by the relocation loader facility provided in COMMAND.COM, based on your response to the Load Low parameter. The Load Low parameter will be discussed in this appendix.

Group

A *group* is a collection of segments that fit together within a 64K-byte segment of storage. The segments are named to the group by the assembler or compiler. A program may consist of one or more groups.

The group is used for addressing segments in storage. The various portions of segments within the group are addressed by a segment base pointer plus an offset. The linker checks that the object modules of a group meet the 64K-byte constraint.

Class

A *class* is a collection of segments. The naming of segments to a class affects the order and relative placement of segments in storage. The class name is specified by the assembler or compiler. All portions assigned to the same class name are loaded into storage contiguously.

The segments are ordered within a class in the order that the linker encounters the segments in the object files. One class precedes another in storage only if a segment for the first class precedes all segments for the second class in the input to LINK. Classes are not restricted in size. The classes will be divided into groups for addressing.

Command Prompts

After you start the linker session, you receive a series of four prompts. You can respond to these prompts from the keyboard, respond to these prompts on the command line, or you can use a special diskette file that is called an *automatic response file* to respond to the prompts. An example of an automatic response file is provided in this appendix. Refer to "How to Start the Linker Program" in this appendix for information on how to start the linker session.

LINK prompts you for the names of object, run, list, and library files. When the session is finished, LINK returns to DOS. The DOS prompt is displayed when LINK has finished. If the LINK is unsuccessful, LINK will display a message.

The prompts are described in their order of appearance on the display. The default is shown in square brackets ([]), in the response column.

Prompt	Responses
Object Modules[.OBJ]:	*filespec1*[+*filespec2*...]
Run File [*filename1*.EXE]:	*filespec*
List File [NUL.MAP]:	[*filespec*]
Libraries [.LIB]:	[*filespec1*[+*filespec2*...]]

Notes:

1. If you enter a filespec without specifying the drive, the default drive is assumed. The library prompt has a variation to this.

2. You can end the linker session prior to its normal end by pressing Ctrl-Break.

Detailed Descriptions of the Command Prompts

The following detailed descriptions contain information about the responses that you can enter to the prompts.

Object Modules [.OBJ]:

Enter one or more filespecs for the object modules to be linked. If the extension is omitted, LINK assumes the filename extension .OBJ. If an object module has another filename extension, the extension must also be specified. Object filenames may not begin with the @ symbol.

Filespecs must be separated by single plus (+) signs or blanks.

LINK loads segments into classes in the order encountered.

If you specify an object module, but LINK cannot locate the file, a prompt requests you to insert the diskette containing the specific module. This permits .OBJ files from several diskettes to be included.

On a single-drive system, diskette exchanging can be done safely *only* if VM.TMP has *not* been opened. A message will indicate if VM.TMP has been opened. The VM.TMP file is discussed in this appendix.

IMPORTANT: If a VM.TMP file has been opened, you should *not* remove the diskette containing the VM.TMP file. Remember, once a VM.TMP file is opened, the diskette it resides on cannot be removed.

If a VM.TMP file has been opened and the linker is unable to locate an object module on the same drive as VM.TMP has been allocated, the linker session will end.

Run File [*filename1*.EXE]:

The filespec you enter is created to store the Run (executable) file that results from the LINK session. All Run files receive the filename extension .EXE, even if you specify an extension. If you specify an extension, your specified extension is ignored.

The default filename for the runfile prompt is the first filename specified on the object module prompt.

List File [NUL.MAP]:

The List file will not be created unless you specifically request it. This can be done by overriding the default with a filespec or a drive ID. If the linker is unable to locate an object module on the same drive as the list file has been allocated, the linker session will end.

The List file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the Run file.

The DOS reserved filename NUL with the default extension .MAP is used if you do not enter a filespec.

> **Note:** If the List file is allocated to a diskette, it must not be removed until the LINK has ended.
>
> To avoid generating the .MAP file on a diskette, you can specify the display as the List file device. For example
>
> ```
> List File [NUL.MAP]: CON
> ```
>
> If you direct the output to your display, you can also print a copy of the output by pressing the Ctrl-PrtSc keys.

Libraries [.LIB]:

The valid responses are either listing a combination of library filespecs and drive identification, or pressing the Enter key. If you just press the Enter key, LINK defaults to the library provided as part of the Compiler package. The Compiler package also provides the location of the library. For linking objects from just the MACRO Assembler, there is no automatic default library search.

When LINK attempts to reference a library file and cannot find it, a prompt requests you to enter the drive identifier containing the library.

If you answer the library prompt, you may specify a list of drive IDs and filespecs separated by plus signs (+) or spaces. A drive ID tells the linker where to look for all subsequent libraries on the library prompt. The automatically searched library filespecs are conceptually placed at the end of the response to the library prompt.

When linking an object module produced by the IBM Personal Computer Fortran Compiler which looks for the library FORTRAN.LIB on drive A, the following library prompt responses may be used:

```
Libraries [.LIB]:B:
```

Look for FORTRAN.LIB on drive B.

```
Libraries [.LIB]:B:USERLIB
```

Look for USERLIB.LIB on drive B and FORTRAN.LIB on drive A.

```
Libraries [.LIB]:A:+USERLIB1+USERLIB2+B:+USERLIB3+A:
```

Look for USERLIB1.LIB and USERLIB2.LIB on drive A, USERLIB3.LIB on drive B, and FORTRAN.LIB on drive A.

You can enter from 1-8 library filespecs. The filespecs must be separated by plus signs or spaces.

LINK searches the library files in the order they are listed to resolve external references. When it finds the module that defines the external symbol, the module is processed as another object module.

If two or more libraries have the same filename and filename extension regardless of the location, only the first library in the search order is searched.

Parameters

At the end of any of the four linker prompts, you may specify one or more *parameters* that instructs the linker to do something differently. Only the / and first letter of any parameter is required.

/DSALLOCATION

The */DSALLOCATION* parameter directs LINK to load all data defined to be in DGROUP at the *high-end* of the group. If the /HIGH parameter is specified (module loaded high), this allows any available storage below the specifically allocated area within DGROUP to be allocated dynamically by your application and still be addressable by the same data space pointer.

> **Note:** The maximum amount of storage which can be dynamically allocated by the application will be 64K (or the amount actually available) minus the allocated portion of DGROUP.

If the */DSALLOCATION* parameter is not specified, LINK will load all data defined to be in the group whose group name is DGROUP, at the *low-end* of the group, beginning at an offset of 0. The only storage thus referenced by the data space pointer should be only that specifically defined as residing in the group.

All other segments of any type in any GROUP other than DGROUP will be loaded at the low-end of their respective groups, as if the /DSALLOCATION parameter is not specified.

For certain compiler packages, DSALLOCATION is automatically used.

/HIGH

The */HIGH* parameter directs LINK to cause the loader to place the Run image as high as possible in storage. If you specify the */HIGH* parameter, this directs the linker to cause the loader to place the Run file as high as possible without overlaying the transient portion of **COMMAND.COM**, which occupies the highest area of storage when loaded. If you do not specify the */HIGH* parameter, the linker will direct the loader to place the Run file as low in memory as possible.

The */HIGH* parameter is used with the */DSALLOCATION* parameter.

/LINE

For certain IBM Personal Computer language processors, the */LINE* parameter directs LINK to include the line numbers and addresses of the source statements in the input modules in the List file.

/MAP

The */MAP* parameter directs LINK to list all public (global) symbols defined in the input modules. For each symbol, LINK lists its value and segment-offset location in the Run file. The symbols are listed at the end of the List file.

/PAUSE

The */PAUSE* parameter tells LINK to display a message to you. This message will request you to insert the diskette that is to receive the Run file.

/STACK:*size*

The *size* entry is any positive decimal value up to 65536 bytes. If you do not use the /STACK, you specify that the original stack size provided by the assembler or compiler is to be used.

If you specify a value greater than 0 but less than 512, the value 512 is used. This value is used to override the size of the stack that the assembler or compiler has provided for the load module being created.

If the size of the stack is too small, the results of executing the resulting load module are unpredictable.

At least one input (object) module must contain a stack allocation statement. This is automatically provided by compilers. For the assembler, the source must contain a SEGMENT command that has the combine type of STACK. If a stack allocation statement was not provided, LINK returns the following message:

Warning: No Stack statement

How to Start the Linker Program

Before You Begin

- Make sure the files you will be using for the LINK are on the appropriate diskettes.
- Make sure you have enough free space on your diskettes to contain your files and any generated data.

You can start the linker program by using one of two options:

Option 1 – Console Responses

From your keyboard, enter:

```
LINK
```

The linker is loaded into storage and displays a series of four prompts, one at a time, to which you must enter the requested responses. (Detailed descriptions of the responses that you can make to the prompts are discussed in this appendix in the section called "Command prompts.")

If you enter an erroneous response, such as the wrong filespec or an incorrectly spelled filespec, you must press Ctrl-Break to exit LINK, then you must restart LINK. If the response in error has been typed but not entered, you may delete the erroneous characters, for that line only.

An example of a linker session, using the display response option, is provided in this appendix in the section called "Example Linker Session."

As soon as you have entered the last filename, the linker will begin to run. If the linker finds any errors, it will display the errors as well as in the listing file.

> **Note:** After any of these responses, before pressing Enter, you may continue the response with a comma and the answer to what would be the next prompt without having to wait for that prompt. If you end any with the semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

Option 2 – Command Line

From your keyboard, enter:

```
LINK objlist,runfile,mapfile,liblist/parms;
```

Your linker is loaded and immediately performs the tasks indicated by the command field as shown in the above example.

When you use this command line, the prompts described in Option 1 will not be displayed if an entry for all four files are specified or if the command line ends with a semicolon.

If an incomplete list is given and no semicolon is used, the linker will prompt for the remaining unspecified files. The */parms* will never be prompted for, but may be added to the end of the command line or to any file specification given in response to a prompt. Each prompt will display its default which may be accepted by pressing the Enter key, or overridden with an explicit filename or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files will be defaulted without further prompting.

Certain other variations of this command line are permitted.

Examples:

1) **LINK module**

Object module is module.OBJ. A prompt is given, showing the default of module.EXE. After the response is entered, a prompt is given showing the default of NUL.MAP. After the response is given, a prompt is displayed showing the default of .LIB.

2) **LINK module;**

If the semicolon is added, no further prompts are displayed. The object module of module.OBJ is linked, the runfile will be put into module.EXE; no listfile is produced.

3) **LINK module,,;**

This is similar to the above example, except the listfile is produced in module.MAP.

4) **LINK module,,**

Using the same example, but without the semicolon, module.OBJ is linked, the runfile is produced in module.EXE, but a prompt is given with the default of module.MAP.

5) **LINK module,,NUL;**

No listfile is produced. The runfile is in module.EXE. No further prompts are displayed.

Option 3 – Automatic Responses

From your keyboard, enter:

LINK @filespec

It is often convenient to save responses to the linker for use at a later time. This is very convenient when long lists of object modules need to be specified.

For this option, you enter a filespec preceded by an @ symbol in place of a prompt response or part of a prompt response. The prompt is answered by the contents of the diskette file. The filespec may not be a reserved DOS filename.

Before using this option, you must create the Automatic Response File. It contains several lines of text, each of which is the response to a linker prompt. These responses must be in the same order as the linker prompts that were discussed earlier in this appendix. If desired, a long response to the object module or libraries prompt may be contained across several lines by using a plus sign (+) to continue the same response onto the next line.

Use of the filename extension is optional and may be any name. There is no default extension.

Use of this option permits the command that starts LINK to be entered from the keyboard or within a batch file without requiring any response from you.

Automatic Response File – Resp1

```
MODA+MODB+MODC

MODD+MODE+MODF
```

Automatic Response File – Resp2

```
Runfile/P
Printout
```

Command line

```
LINK @Resp1+mymod,@Resp2:
```

Notes:

1. In this example, the use of the plus sign causes the modules listed in the first two lines and any module entered by the user on the command line in response to the object module prompts to be considered as the input object modules.

2. Each of the above lines ends when you press the Enter key.

Example Linker Session

This example shows you the type of information that is displayed during a linker session.

Once you enter

```
A>b:link
```

the system responds with the following messages:

```
IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982
Object Modules [.OBJ]: example
Run File [EXAMPLE.EXE]: example/map
List File [NUL.MAP]: prn/line
Libraries [.LIB]:
```

Notes:

1. By responding **prn** to the List file prompt, we sent our output to the printer.

2. By just pressing Enter in response to the Libraries prompt, an automatic library search is performed.

3. By specifying the /MAP parameter, we get both an alphabetic listing and a chronological listing of public symbols.

4. By specifying the /LINE parameter, LINK gives us a listing of all line numbers for all modules. (The /LINE parameter can generate a large amount of output.)

If LINK cannot locate a library on the specified drive, the following message is displayed:

```
cannot find library A:FORTRAN.LIB
enter new drive letter:
```

The drive that the indicated library is located on must be entered.

Once LINK locates all libraries, the Linker MAP displays a list of segments in the relative order of their appearance within the load module. The list looks like this:

```
Start     Stop      Length  Name            Class

00000H    00028H    0029H   MAINQQ          CODE
00030H    000F6H    00C7H   ENTXQQ          CODE
00100H    00100H    0000H   INIXQQ          CODE
00100H    038D3H    37D4H   FILVQQ_CODE     CODE
038D4H    04921H    104EH   FILUQQ_CODE     CODE
•
•
•
074A0H    074A0H    0000H   HEAP            MEMORY
074A0H    074A0H    0000H   MEMORY          MEMORY
074A0H    0759FH    0100H   STACK           STACK
075A0H    07925H    0386H   DATA            DATA
07930H    082A9H    097AH   CONST           CONST
```

The information in the **Start** and **Stop** columns shows a 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module. The addresses displayed are not the absolute addresses of where these segments are loaded. To find the absolute address of where a segment is actually loaded, you must determine where the segment listed as being at relative zero is actually loaded; then add the absolute address to the relative address shown in the .MAP listing. The procedure you use to determine where relative zero is actually located is discussed in this appendix, in the section called "How to Determine the Absolute Address of a Segment."

Now, because we specified the /MAP parameter, the public symbols are displayed by name and by value. For example:

```
Address          Publics by Name

0492:0003H       ABSNQQ
06CD:029FH       ABSRQQ
0492:00A3H       ADDNQQ
06CD:0087H       ADDRQQ
0602:000FH       ALLHQQ
•
•
•
0010:1BCEH       WT4VQQ
0010:1D7EH       WTFVQQ
0010:1887H       WTIVQQ
0010:19E2H       WTNVQQ
0010:11B2H       WTRVQQ

Address          Publics by Value

0000:0001H       MAIN
0000:0010H       ENTGQQ
0000:0010H       MAINQQ
0003:0000H       BEGXQQ
0003:0095H       ENDXQQ
•
•
•
F82B:F31CH       CRCXQQ
F82B:F31EH       CRDXQQ
F82B:F322H       CESXQQ
F82B:F5B8H       FNSUQQ
F82B:F5E0H       OUTUQQ
```

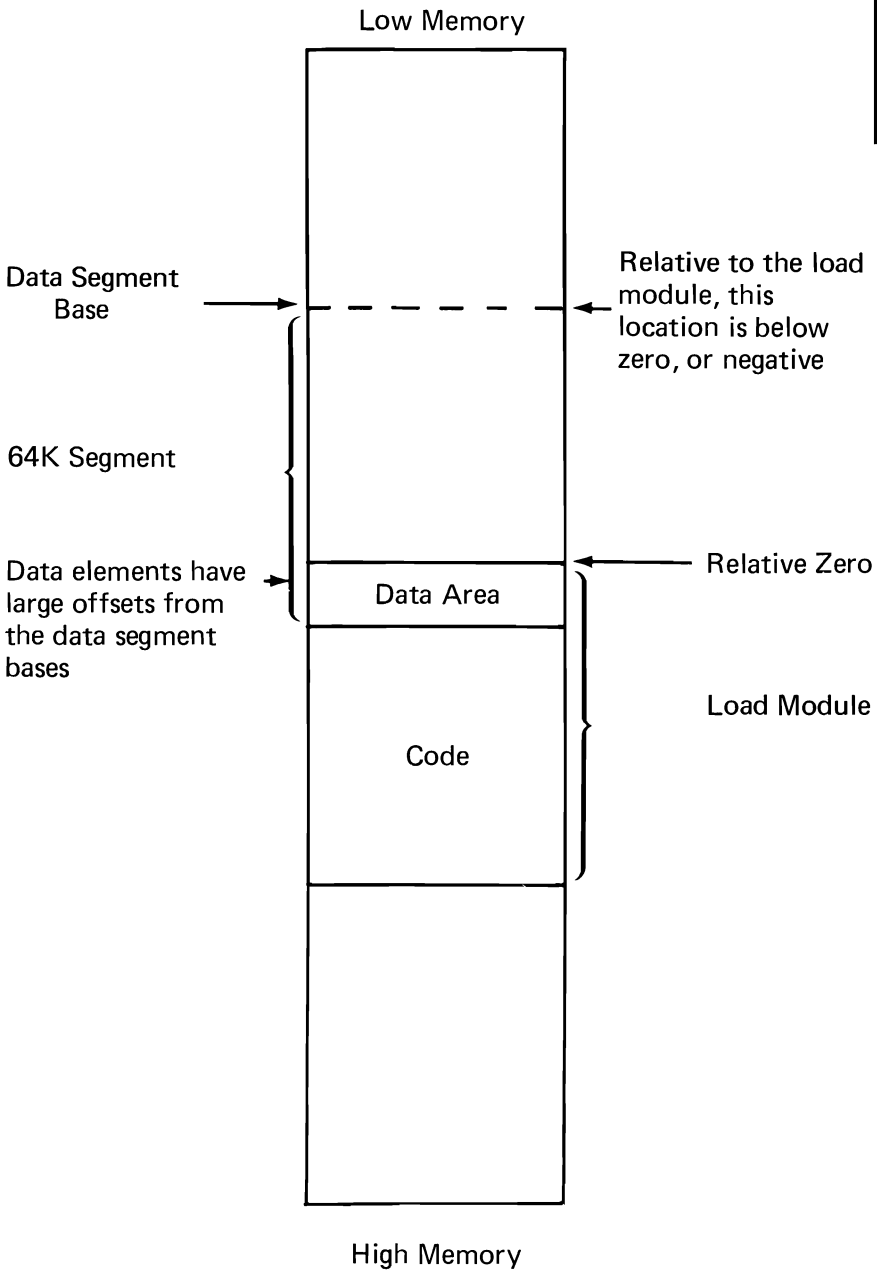
The addresses of the public symbols are also in the *segment offset* format, showing the location relative to zero as the beginning of the load module. In some cases, an entry may look like this:

```
F8CC:EBE2H
```

This entry appears to be the address of a load module that is almost one megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative. See the chart at the end of this appendix. When LINK has completed, the following message is displayed:

Program entry point at 0003:0000

Load Module Storage Map



How to Determine the Absolute Address of a Segment

The Linker MAP displays a list of segments in the relative order of their appearance within the load module. The information displayed shows a 20-bit hex address of each segment relative to location zero. The addresses that are displayed are not the absolute addresses of where these segments are actually located. To determine where relative zero is actually located, we must use $DEBUG. $DEBUG is discussed in Chapter 6.

Using $DEBUG,

1. Load the application.

   Note the segment value in CS and the offset within that segment to the entry point as shown in IP. The last line of the Linker MAP also describes this entry point, but uses relative values, not the absolute values shown by CS and IP.

2. Subtract the relative entry as shown at the end of the .MAP listing from the CS IP value. For example, let's say CS is at 05DC and IP is at zero.

   The Linker MAP shows the entry point at 0100 0000. (0100 is a segment ID or paragraph number; 0000 is the offset into that segment.)

   In this example, relative zero is located at 04DC 0000, which is 04DC0 absolute.

   If a program is loaded low, the relative zero location is located at the end of the Program Segment Prefix, or in the value in DS plus 100H.

Messages

All messages, except for the warning messages, cause the LINK session to end. Therefore, after you locate and correct a problem, you must rerun LINK.

Messages will appear both in the list file and on the display unless you have directed the list file to CON in which case the display messages are suppressed.

A complete list of messages are as follows:

About to generate .EXE file

Change diskettes, press any key.

An internal failure has occurred

Report this problem to your authorized IBM Personal Computer Dealer.

Attempt to access data outside of segment bounds

The object module is probably bad.

Bad numeric parameter

An invalid number was found on the /STACK parameter.

Cannot find file *filename*

Change diskettes, press any key.

This error is unrecoverable if either VM.TMP or the list file has been opened to the diskette where the object cannot be located.

Cannot find library *library name*

Enter new drive letter.

Cannot open overlay

Cannot open temporary file

The directory is full.

DUP record too complex

A problem exists in an object module created from an assembler source program. A single DUP requires 1024 bytes before expansion.

Fixup offset exceeds field width

A machine language processor instruction refers to an address with a NEAR attribute instead of a FAR attribute.

Invalid format file

A library is in error.

Invalid object module

Object module(s) incorrectly formed or incomplete (as when the language processor was stopped in the middle).

Invalid switch

The linker found an invalid parameter on the Command line or on a prompt.

Out of space on list file

Out of space on run file

Out of space on VM.TMP

No more diskette space remains to expand the VM.TMP file.

Program size exceeds capacity of linker

The load module is too big for processing.

Segment size exceeds 64K

Attempted to combine identically named segments which resulted in segment requirement of greater than 64K. 64K-bytes is the addressing limit.

Stack size exceeds 64K

A number greater than 65536 was found on the /STACK parameter.

Symbol defined more than once

The linker found two or more modules that define a single symbol name.

Symbol table capacity exceeded

The limit is about 30K. Use shorter and/or fewer names.

There was/were *number* errors detected

Too many libraries specified

The limit is 8 libraries.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 10 including DGROUP.

Too many public symbols in one module

The limit is 1024 public symbols.

Too many segments or classes

The limit is 256 (segments and classes taken together).

Too many overlays

The limit is 64.

Unexpected end-of-file on library

Unexpected end-of-file on VM.TMP

The diskette containing VM.TMP has been removed.

VM.TMP is an illegal file name and has been ignored

VM.TMP cannot be used for object file name.

APPENDIX D. LINKING OBJECT MODULES

You may find it useful to compile parts of a Fortran program separately. This may be necessary when not enough diskette space or storage is available to the compiler. An IBM Fortran compilation consists of one or more program units. Thus, you may place parts of your source program in separate files, compile them separately, and link them together using the IBM Personal Computer Linker.

Object modules produced by the IBM Personal Computer Fortran Compiler may be linked with those produced by the IBM Personal Computer Pascal Compiler and the IBM Personal Computer MACRO Assembler. This appendix describes how to link:

- An IBM Personal Computer Pascal subroutine with an IBM Personal Computer Fortran main program.
- An IBM Personal Computer Fortran subroutine with an IBM Personal Computer Pascal main program.
- An assembler subroutine with an IBM Personal Computer Fortran main program.

IBM Fortran passes all parameters by reference. That is, the address of an actual argument is passed to the subroutine not the value. When a call is made to a subroutine from IBM Fortran, the segment base and the offset of the actual arguments are pushed on the stack in a left to right order. A long call is then done to the subroutine. This type of parameter passing is done for all types of parameters and the subroutine must know exactly what the address on the stack points to in order to handle it correctly. When complete arrays are passed, the start address of the array is passed. When externals or intrinsics (procedural parameters) are passed, a zero (0) and a DS offset are pushed and point to a 4-byte code segment base and offset pair of the item passed.

Linking with Pascal

It is easy to link IBM Fortran with the IBM Personal Computer Pascal. IBM Fortran pass-by-reference parameters are known to the IBM Personal Computer Pascal as VARS parameters. The following relationship exists between the data types for the two languages:

IBM Fortran	IBM Pascal
INTEGER*2	INTEGER
INTEGER*4	NONE
LOGICAL*2	NONE (See Note 1)
LOGICAL*4	NONE
CHARACTER*n	STRING(n)
REAL	REAL
REAL*4	REAL

Notes:

1. IBM Fortran LOGICAL*2 variables may be referenced in IBM Personal Computer Pascal with the INTEGER type.

2. When I/O is to be done in a subroutine which is not in the same compiler language as the main program, a call must be made to the file system initialization routine of the language in which the subroutine was written before any I/O in the subroutine can be done. These routines reside in the file system units of the respective language. Normally a call is automatically done, but when a main program for that language does not exist, the call to the routine is not made.

Initialization routine names

The following routines do not have parameters and the call is made in the main program or the subroutine:

- IBM Fortran – INIVQQ
- IBM Pascal – INIFQQ

The following main program and subroutine demonstrates combining IBM Fortran and IBM Pascal:

An IBM Fortran Main Program:

```
$STORAGE:2
      CHARACTER*8 ts
      INTRINSIC sin
      EXTERNAL sub1

C INITIALIZE THE IBM PASCAL FILE SYSTEM
      CALL INIFQQ

      CALL PASCAL (1,ts,result,.TRUE.,sin)
      WRITE(*,100) ts,result
      CALL PASCAL (2,ts,result,.FALSE.,sub1)
      WRITE(*,110) ts,result
100   FORMAT(1x,'Time is: ',a8,' Sin(.5) = ',f10.5)
110   FORMAT(1x,'date is: ',a8,' sub1(.5) = ',f10.5)
      END

      REAL FUNCTION sub1(parm)
      REAL parm
      sub1 = cos(parm)+5
      END
```

An IBM Personal Computer Pascal Subroutine

```
MODULE pasmodule;
PROCEDURE date(VAR s:STRING); EXTERN;
PROCEDURE time(VAR s:STRING); EXTERN;

TYPE
     string8=STRING(8);

PROCEDURE pascal (VARS code:INTEGER;VARS
     line:STRING8;VARS num:REAL;
     VARS switch:INTEGER;FUNCTION
     func(VARS i:REAL):REAL);

VAR s:string8;
     parm:REAL;

BEGIN
     {Get the code and perform the requested function}
     IF code=1 then
       BEGIN
         time(s);
         line:=s;
       END
     ELSE
       IF code=2 THEN
         BEGIN
           date(s);
           line:=s;
         END
       ELSE
         writeln('Invalid code');
     {Make the call to the passed in procedure}
     parm:=0.5;
     num:=func(parm);
     {Examine the logical variable}
     IF switch=1 THEN
          WRITELN('true')
```

```
    ELSE
        IF switch=0 THEN
            WRITELN('false')
        ELSE
            writeln('Error in logical variable')
END;
END.
```

When the two programs are linked together using the IBM Personal Computer Linker, the final run file will call the procedure PASCAL twice, returning the results as indicated in the logic of the program.

> **Note:** When linking IBM Personal Computer Pascal V1.00 with IBM Personal Computer Fortran V1.00, it is important that the linker search the IBM Fortran library first. This can be done by placing all object modules produced by IBM Fortran before those produced by IBM Personal Computer Pascal in response to the "Object module" prompt of the linker.
>
> You may also explicitly supply the names of the libraries in the linker "Libraries" and place the IBM Fortran library first. In addition, the linker will report PPMUQQ as being defined more than once when IBM Fortran and IBM Pascal are linked successfully. (Ignore the message.)

Linking with the MACRO Assembler

It is useful to call an assembler language routine compiled with the IBM Personal Computer MACRO Assembler to communicate with the IBM Personal Computer DOS or the Basic Input Output System (BIOS). Other machine level facilities may also be accessed in this way.

When a call is made to a subroutine, the area on the stack used to communicate between the calling program and the subroutine is termed the frame. The stack is growing down so as parameters are pushed the stack pointer (SP) decreases.

The layout of the frame is as follows:

Low address

Optionally, this area can be used for temporary data
A saved copy of the callers BP
A 4-byte return address pushed by the CALL instruction
Addresses of the parameters pushed by the caller. These are segment base/ offset pairs.

High address

Note: The stack grows from the high address to the low address.

When your assembler language subroutine receives control from the IBM Fortran, the parameters and the return address are on the stack and the SP points to the first byte of the 4-byte return address. At this point, you should push BP so that it can later be restored for the caller. If you are using the stack as a temporary data area, you should adjust the SP downward as to leave room for this data. You at this point set BP to SP so your BP points to the first byte at the bottom of the frame. This is very useful in accessing the information on the stack.

The following points should be noted:

- The far procedure must be contained within a code segment.

- The name of the far procedure must be identified as public.

- A group statement must be included to group the DATA segment with the group DGROUP.

- You should assume CS points to your code segment and DS, ES, and SS point to DGROUP.

- You must save and restore the callers BP and SP. The callers CS and IP are restored by the RET instruction. If you alter the caller's DS you must restore it also.

The following IBM Fortran program calls an IBM Personal Computer MACRO Assembler subroutine which obtains the time from the IBM Personal Computer DOS.

The IBM Fortran main program is as follows:

```
$STORAGE:2
      INTEGER A,B,HOURS,MINS,SECS,HSECS
      CALL TIMER(A,B)
      HOURS=A/256
      MINS=MOD(A,256)
      SECS=B/256
      HSECS=MOD(B,256)
      WRITE(*,10)HOURS,MINS,SECS,HSECS
10    FORMAT(1x,'THE TIME IS: ',i2,':',i2':',i2,'.',i2)
      END
```

The IBM Personal Computer MACRO Assembler subroutine is as follows:

```
FRAME   STRUC
SAVEBP  DW      ?     ;SAVE AREA FOR CALLER'S BP
SAVERET DD      ?     ;RETURN ADDRESS FOR FAR RETURN
B       DD      ?     ;ADDRESS OF WORD WHERE HOUR
;                      AND MINUTE ARE TO BE PLACED
A       DD      ?     ;ADDRESS OF WORD WHERE SECS
;                      AND HUNDREDS ARE TO BE PLACED
FRAME   ENDS
;
CSEG    SEGMENT 'CODE'
DGROUP  GROUP   DATA
        ASSUME  CS:CSEG,DS:DGROUP,ES:DGROUP,SS:DGROUP
TIMER   PROC    FAR
        PUBLIC  TIMER
        PUSH    BP          ;SAVE CALLER'S BP
        MOV     BP,SP       ;SET FRAME BASE
        MOV     AH,2CH      ;REQUEST THE TIME FUNCTION
        INT     21H         ;CALL (INTERRUPT) DOS
        LES     BX,[BP]+A   ;GET THE ADDR OF 1ST PARM
        MOV     ES:[BX],CX  ;PUT HOURS & MIN INTO 1ST
;                           PARM
        LES     BX,[BP]+B   ;GET THE ADDR OF 2ND PARM
        MOV     ES:[BX],DX  ;PUT SECS AND HUNDREDS
;                           INTO 2ND PARM
        POP     BP          ;RESTORE CALLER'S BP
        RET     8           ;FAR RETURN, FREE 8 BYTES
;                           OR PARMS, RESTORE THE SP
TIMER   ENDP
CSEG    ENDS
        END
```

APPENDIX E. A SAMPLE SESSION

```
A>REM -- The following sample session illusrates the use

A>REM -- of the IBM Personal Computer FORTRAN Compiler to compile, debug and

A>REM -- execute an application program.  The intermixed REMs and

A>REM -- PAUSEs are for documentation purposes only.

A>
A>REM -- The DOS diskette is in drive A

A>REM -- Make the DOS default drive B

A>B:

B>PAUSE -- Insert into drive B, an unformatted diskette, the scratch diskette
Strike a key when ready . . . d
B>REM -- Format the new diskette

B>a:format
Insert new diskette for drive B:
and strike any key when ready
```

```
Formatting...Format complete


Format another (Y/N)?n


B>REM -- Make a copy of Edlin on the scratch diskette

B>copy a:edlin.com b:
          1 File(s) copied
B>REM -- Invoke Edlin to input the program

B>edlin grades.for
New file
*i
 1:*$title:'Student grades management program'
 2: $storage:2
 3:       PROGRAM grades
 4:
 5: C Define arrays and variables
 6:       DIMENSION scores(14,10),total(14)
 7:       CHARACTER*5 names(14)
 8:       INTEGER s
 9:       CALL start(nostd,notst)
10:       CALL getgrd(nostd,notst,names,scores)
11:       CALL totit(nostd,notst,scores,total)
```

```
      CALL report(nostd,notst,names,scores,total)
      CALL rptavg(nostd,notst,scores,total)
      END
C Read in program parameters
      SUBROUTINE start(nstd,ntst)
      WRITE(*,5)
    5 FORMAT('0',10x,'Student Grades Management')
      WRITE(*,6)
    6 FORMAT(11x,'******* ****** **********')
      WRITE(*,8)
    8 FORMAT('0Input #Students and #Grades-Use 2I2')
      READ (*,10) nstd,ntst
   10 FORMAT(2i2)
      END
C Read in student grades
      SUBROUTINE getgrd(nstd,ntst,names,scrbk)
      CHARACTER*5 names(nstd)
      DIMENSION   scrbk(nstd,ntst)
      DO 15 a=1,nstd
           WRITE(*,9) ntst
    9      FORMAT('0Name-A5  Grade-',I2,'F5.0')
           READ(*,20) names(a),(scrbk(a,j),j=1,ntst)
   20      FORMAT(a5,6f5.0)
   15 CONTINUE
      END
```

```
C Make the totals
      SUBROUTINE totit(nstd,ntst,scrbk,total)
      DIMENSION scrbk(nstd,ntst),total(nstd)
      DO 10 i=1,nstd
         total(i)=0
         DO 10 j=1,ntst
            total(i)=total(i)+scrbk(i,j)
10    CONTINUE
      END
C Report student scores
      SUBROUTINE report(nstd,ntst,names,scrbk.total)
      CHARACTER*5 names(nstd)
      DIMENSION scrbk(nstd,ntst),total(nstd)
      WRITE(*,45)
   45 FORMAT(13x,'Student  Test 1 Test 2 Test 3 Test 4 Test 5 Test 6')
      WRITE(*,50)
   50 FORMAT(13x,'*******  ****** ****** ****** ****** ****** ******')
      WRITE(*,52)
   52 FORMAT(1x)
      DO 55 i=1,nstd
          WRITE(*,60) names(i),(scrbk(i,j),j=1,ntst)
          FORMAT(15x,a5,2x,6(f6.2,1x))
   55 CONTINUE
      WRITE(*,70)
   70 FORMAT(1h0,30x,'Student Average')
```

```
      WRITE(*,75)
   75 FORMAT(31x,'***************')
      WRITE(*,52)
      DO 80 i=1,nstd
          WRITE(*,85) names(i),total(i)/ntst
   85     FORMAT(32x,a5,3x,f6.2)
   80 CONTINUE
      END

C Report averages
      SUBROUTINE rptavg(nstd,ntst,scrbk,total)
      DIMENSION scrbk(nstd,ntst),total(nstd)
  110 WRITE(*,65)
   65 FORMAT('0',30x,'Class Average')
      WRITE(*,90)
   90 FORMAT(31x,'*************')
      tscrbk=0
      DO 40 i=1,nstd
          tscrbk=tscrbk+total(i)
   40 CONTINUE
      carg=tscrbk/(nstd*ntst)
      WRITE(*,95) carg
   95 FORMAT('0',35x,f6.2)
      END
*\
```

```
*e

B>REM -- Look at the directory

B>dir
EDLIN      COM         2392   08-04-81
GRADES     FOR         2496   11-09-81

B>REM -- Run FOR1 (pass one), sending the listing to the printer

B>PAUSE -- Insert the FOR1 diskette in drive A
Strike a key when ready . . . a
B>a:for1
IBM Personal Computer FORTRAN Compiler
Version 1.00 (C)Copyright IBM Corp 1982
Source filename [.FOR]: grades
Object filename [GRADES.OBJ]:
Source listing  [NUL.LST]: prn
Object listing  [NUL.COD]:

***** Error 137,line 30 -- integer variable expected

***** Error 135,line 58 -- FORMAT label missing

***** Error 163,line 69 -- FORMAT not found
```

```
B>REM -- Got three errors, go back to Edlin to fix them

B>edlin grades.for
End of input file
*30
    30:*       DO 15 a=1,nstd
    30:*       DO 15 i=1,nstd
*33
    33:*           READ(*,20) names(a),(scrbk(a,j),j=1,ntst)
    33:*           READ(*,20) names(i),(scrbk(i,j),j=1,ntst)
*58
    58:*           FORMAT(15x,a5,2x,6(f6.2,1x))
    58:*   60      FORMAT(15x,a5,2x,6(f6.2,1x))
*e

B>REM -- Rerun FOR1

B>a:for1
IBM Personal Computer FORTRAN Compiler
Version 1.00 (C)Copyright IBM Corp 1982
Source filename [.FOR]: grades
Object filename [GRADES.OBJ]:
Source listing  [NUL.LST]:
Object listing  [NUL.COD]:
```

```
B>REM -- Ok, that worked, now execute FOR2 (pass two)

B>REM -- Look at the directory

B>dir
EDLIN      COM         2392   08-04-81
GRADES     BAK         2496   11-09-81
GRADES     FOR         2496   11-09-81
PASIBF     SYM         4968   11-09-81
PASIBF     BIN        11232   11-09-81

B>PAUSE -- Insert the FOR2 diskette in the drive A
Strike a key when ready . . . h
B>a:for2

 Code Area Size = #0644  (1604)
 Cons Area Size = #0006    (6)
 Data Area Size = #04A0   (1184)

 Pass Two    No Errors Detected.

B>REM -- Look at the directory
```

```
B>dir
EDLIN      COM         2392   08-04-81
GRADES     BAK         2496   11-09-81
GRADES     FOR         2496   11-09-81
GRADES     OBJ         3592   11-09-81

B>REM -- Now its Link time

B>PAUSE -- Insert the LIBRARY diskette in drive A
Strike a key when ready . . .
B>a:link

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp 1982

Object Modules [.OBJ] : grades
Run File [GRADES.EXE] :
List File [NUL.MAP] :
Libraries [.LIB] :
```

```
B>REM -- Look at the directory

B>dir
EDLIN      COM          2392   08-04-81
GRADES     BAK          2496   11-09-81
GRADES     FOR          2496   11-09-81
GRADES     EXE         36864   11-09-81
GRADES     OBJ          3592   11-09-81

B>REM -- Ok, great it compiled, now run the program

B>grades


           Student Grades Management
           ******* ****** **********

Input #Students and #Grades-Use 2I2
 5 3


Name-A5  Grade- 3F5.0
Mez   100.0 100.   100
```

```
Name-A5  Grade- 3F5.0
Danny   78.   85.    99

Name-A5  Grade- 3F5.0
Steve    42    67    38

Name-A5  Grade- 3F5.0
Janin   99.  100.   93.

Name-A5  Grade- 3F5.0
Joe      95    83    93
```

Student	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Mez	100.00	100.00	100.00			
Danny	78.00	85.00	99.00			
Steve	42.00	67.00	38.00			
Janin	99.00	100.00	93.00			
Joe	95.00	83.00	93.00			

```
Student Average
***************

 Mez       100.00
 Danny      87.33
 Steve      49.00
 Janin      97.33
 Joe        90.33

Class Average
*************

      84.80

B>
```

GLOSSARY

actual argument: The information passed to a procedure by a caller.

allocate: To assign a resource for use in performing a specific task.

ASCII: American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of seven-bit coded characters (8 bits including the parity bit). The ASCII set consists of control and graphic characters.

ASM: A filename extension that identifies a source file for the IBM Personal Computer MACRO Assembler.

AUX: The device identification for the asynchronous communication adapter port (serial port).

basic real constant: A real number (sign optional) that does not have an exponent part.

byte: Eight bits, enough storage for one character.

COD: A filename extension that identifies a disassembled object listing file produced by a compiler.

code: A set of rules specifying the manner in which data may be represented in a discrete form. A set of instructions that specifies the manner in which data is to be manipulated.

COM1: The device identification for the asynchronous communication adapter port (serial port).

CON: The device identification for the display/keyboard.

constant: A data item of non-varying fixed value.

debug: To detect, trace, and correct errors in a computer program.

default: A value, attribute, or option that is assumed when none has been specified by the user.

diskette drive A, B: The device identification for the diskette drives. Used in front of any filename specification.

DOS: Disk Operating System.

dummy argument: The name that is given to the information received by a procedure from a caller. (See **formal parameter**.)

edit: To create or modify the contents of a source program or document. For example, to insert, delete, change, rearrange, or copy lines.

editor: A computer program that processes commands to enter lines into a source program or document or to modify them.

EDLIN: The text editor of DOS which can be used to create, modify, and display files.

endfile record: The Fortran file system simulates a virtual endfile record after the last record in a file, although there is no corresponding real record.

EXE: A filename extension that identifies a relocatable executable file.

expression: A source language combination of one or more arithmetic or logical operations and operands often represented by a combination of terms possibly within paired parentheses.

filename: The filename consists of one-to-eight characters. It may be immediately followed by a filename extension.

filename extension: The optional filename extension consists of a period (.) and from one-to-three characters. It is used to identify the type of the file.

filespec: A 3-part file specification consisting of an optional diskette drive identification, a required filename, and an optional filename extension.

file control block: A table which resides in the heap containing updated information pertaining to an opened file, which the file system uses for accurate and correct handling of that file.

FOR: A filename extension that identifies a Fortran source file.

formal parameter: The name that is given to the information received by a procedure from a caller. (See **dummy argument**.)

formatted record: Formatted records are sequences of characters terminated by the carriage return–linefeed. Formatted records are interpreted on input consistently with the way the IBM Personal Computer DOS interprets characters. They are useful when the source data is to be read by a user or used as input to another program. Formatted records must be used with the formatted I/O statements. They are the most common type of record and are required when writing or reading to or from the display/keyboard or printer.

Fortran Compiler, IBM Personal Computer: A computer program that translates a source program written in the Fortran language to object code.

heap: An area of storage used for dynamic space allocation such as file control blocks.

LINE: The device identification for the asynchronous communication adapter port (serial port).

Linker, IBM Personal Computer: That part of the Disk Operating System used to create one load module from one or more independently translated object modules by resolving cross references among the modules.

listing file: A file, which can be printed or displayed, which lists the source code statements and errors.

LST: A filename extension that identifies a printable listing file produced by a compiler or assembler.

LPT1: The device identification for the line printer.

MAP: A filename extension that identifies the printable listing file produced by the IBM Personal Computer Linker.

MACRO Assembler, IBM Personal Computer: A computer program that translates a *source* program written in assembler language into *object* code.

NUL: The device identification for a non-existent device. As an input device, immediate end-of-file is generated. As an output device, write operations are simulated but data is not written.

OBJ: A filename extension that identifies the relocatable object file.

object code: Output from a language translator, which is itself executable machine code or is suitable for processing to produce executable machine code.

object module: A program unit, which is the output of a language translator, and which is suitable for input to the Linker.

PAS: A filename extension that identifies a Pascal source file.

PASIBF.SYM/PASIBF.BIN: The intermediate binary files passed between the front end (FOR1) and the back end (FOR2) of the IBM Fortran Compiler.

Pascal Compiler, IBM Personal Computer: A computer program that translates a source program written in the Pascal language to object code.

PRN: The device identification for the line printer.

procedure: A function, statement function or subroutine.

source module: The source statements that constitute the input to a language translator.

stack: An area in storage that stores temporary register information and returns addresses of subroutines.

stack pointer (SP): A register that provides the current location of the stack.

symbol: A method of referring to a resource of the computer; a representation of something by relationship, association, or convention.

truncate: To delete or omit a trailing portion of a string of items.

unformatted record: An unformatted record is a sequence of values, with no system alteration or interpretation; no physical representation exists for the end of record. They are used when it is desired to store or retrieve information without the need for editing or user intervention.

USER: The device identification for the non-buffered display/keyboard.

user defined name: Any name that the user defines or redefines. Names that may be redefined are names such as intrinsic functions, statement functions, variable names; and so forth.

user name: Any name that the user references.

variable: A data item that can assume any of a given set of values.

word: Two bytes.

Glossary 6

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

The Personal Computer
Computer Language Series

Product Comment Form

FORTRAN Compiler
by Microsoft 6172284

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

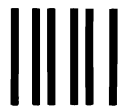
If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____



NO P
NEC
IF M
IN
UNITE

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

The Personal Computer
Computer Language Series

Product Comment Form

FORTRAN Compiler
by Microsoft 6172284

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

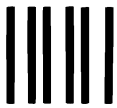
If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____



NO P
NECI
IF M
IN
UNITE

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

The Personal Computer
Computer Language Series

Product Comment Form

FORTRAN Compiler
by Microsoft 6172284

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

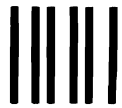
If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____



NO P
NECI
IF M
IN
UNITE

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6172284

Printed in United States of America