commodore **SuperPET** computer

Waterloo microFORTRAN

commodore COMPUTER

Dieses Handbuch wurde gescannt, bearbeitet und ins PDF-Format konvertiert von

Rüdiger Schuldes

schuldes@itsm.uni-stuttgart.de

Waterloo microFORTRAN

Tutorial and Reference Manual

P. H. Dirksen

J. W. Welch

Disclaimer

Waterloo Computing Systems Limited makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Waterloo Computing Systems Limited, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

PREFACE

Waterloo microFORTRAN is a dialect of FORTRAN designed to be used in educational and research environments. Because it is intended to be executed using microcomputers, it is a subset of the FORTRAN-77 standard language. It is also important to provide modern programming facilities such as good primitives for Structured Programming. Consequently, the standard language has been extended in significant ways. Because it is anticipated that the interpreter will be used widely as a debugging processor, special emphasis has been placed upon the detection and diagnosis of errors. In addition, an interactive debugging facility has been implemented.

In order to aid those familiar with the FORTRAN-77 language standard, an overview of the differences between the standard and Waterloo microFORTRAN follows:

(a) The following features are omitted: BLOCK DATA SUBPROGRAM, IMPLICIT statement, ENTRY statement, PARAMETER statement, DATA statement, arithmetic statement functions, EQUIVALENCE statement, COMMON statement, INTRINSIC statement, SAVE statement, BACKSPACE statement, INQUIRE statement, DIMENSION statement and assigned GOTO.

(b) Three data types are supported: INTEGER, REAL and CHARACTER. The implementation of the CHARACTER data type is incompatible with that of FORTRAN-77. The implemented version is considered more flexible than the FORTRAN-77 definition.

(c) A subset of the input/output capability is provided. The implementation includes sequential and direct input/output, together with a significant portion of the possible FORMAT specifications.

(d) Expressions have been generalized to be a superset of those normally allowed.

(e) An extensive collection of statements for Structured Programming have been added.

(f) The format of the program is generalized. Statements may be entered without regard to columns in a line. Continued lines are recognized by an "&" character in the first position. Syntactic units may not be continued

across lines. The length of variable names is limited only by the length of a line.

(g) Space characters within a line are significant delimiters. The language uses reserved words, while FORTRAN-77 specifies contextual recognition of names.

(h) Comment lines are recognized by an "*" character in the first position in a line.

This manual is presented in two parts. The first part is a collection of annotated examples intended to introduce the reader to many of the features of Waterloo microFORTRAN. In this way, a novice is provided with a staged introduction to the language. An experienced programmer will find examples to compare Waterloo microFORTRAN to other dialects or languages. The second part is a comprehensive language reference manual for Waterloo microFORTRAN.

Acknowledgement

All members of the Computer Systems Group at the University of Waterloo have made a significant contribution to the design of the Waterloo microFORTRAN interpreter. The design is based upon ideas evolved and proven over the past decade in other compiler projects in which the group has been involved. The actual design and programming of the processor was primarily performed by Douglas Mulholland, Jack Schueler, Glenn Waters and Jim Welch. Charlotte Ross, Sharon Malleck and Tammy Tilson were responsible for the production of the manual.

P. H. Dirksen,
J. W. Welch,

June, 1981.

Table of Contents

Waterloo microFORTRAN

Tutorial Examples

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Introduction

The following tutorial is a sequence of examples meant to introduce the reader to the "flavour" of Waterloo microFortran. They do not present a complete or rigorous treatment of any topic, as this detailed information is available in the reference manual in the latter part of this document. This tutorial could be useful in the following situations:

- Someone already familiar with FORTRAN can determine some of the major differences between Waterloo microFORTRAN and the dialect already known.

- Teachers may find the examples useful as a progressive introduction of the material to their students.

- People who already know some other language can get an appreciation for Waterloo microFORTRAN before reading the reference manual.

- Complete novices could run the various programs, and possibly learn some of the material by exploring the various language features in conjunction with the reference material.

In order that the examples be fully appreciated, it is important that they be entered into the computer and executed.

Example 1 Comments, Variables, PRINT, STOP

Function

This example sets a FORTRAN variable x equal to 12, squares this value placing the result in y, and prints both x and y.

```
* Example 1

x=12
y=x*x
print,x,y
stop
end
```

Notes:

1. The first statement contains a comment which is indicated by an asterisk (*). The computer ignores comments when executing the program.

2. The second statement is a null line containing only blanks; blank lines may appear anywhere in the program and are ignored when the program is executed.

3. x and y are *FORTRAN variables*. These variables begin with a letter and can contain letters, digits and the dollar sign. They can be of any length.

4. The Print statement contains a *print list*, in this case, x and y. The *values* contained in x and y are printed side by side on the page. Note the comma immediately following the word print.

5. Multiplication is denoted by *. The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

6. Expressions may be contained in parentheses:

   e.g. Y=(X+3.2)*6.4

7. Statements such as x=12 and y=x*x are called assignment statements.

8. Order of priority of operators is as in algebra. Equal priority proceeds left to right.

9. Both upper and lower case letters may be used in variables.

10. FORTRAN keywords such as Print, Stop, and End are always displayed in lower case, even though they may have been entered in upper case. In the Tutorial section all such words will begin with a capital letter for emphasis.

Example 2 Infinite Loops

Function

The computations of Example 1 are repeated endlessly. When you run this program, it must be stopped by depressing the *"RUN STOP"* key one or more times.

```
* Example 2

x=12
23 y=x*x
print,x,y
x=x+1
goto 23
end
```

Notes:

1. The statement x=12 gives an *initial* value to x.

2. A statement number has been inserted on the line which calculates y. Note that it is followed by a space.

3. The statement x=x+1 causes the value of x to increase by 1.

4. The statement Goto 23 causes control to transfer to the statement numbered 23.

5. The statements

```
23 y=x*x
print,x,y
x=x+1
goto 23
```

   are repeated endlessly, i.e. we have an infinite loop.

6 We learn how to stop loops automatically in the following examples.

Example 3 Exiting Loops, Relational Expressions

Function

This example prints squares of the integers from 12 to 20, automatically terminating when x=21.

```
* Example 3

x=12
23 y=x*x
print,x,y
x=x+1
if (x.le.20) goto 23
stop
end
```

Notes:

1. The If statement

   ```
   if (x.le.20) goto 23
   ```

   causes the FORTRAN processor to repeat the loop if x has a value less than or equal to 20. However, when x has the value 21 control proceeds to the next statement.

2. The x.le.20 in the If statement is called a *relational expression* and has the value *true* or *false*. When used with the logical If statement, the relational expression must be enclosed in parentheses. The characters .le., are used to represent a *relational operator*. The relational operators are as follows:

.gt.	greater than
.ge.	greater than or equal to
.lt.	less than
.le.	less than or equal to
.ne.	not equal to
.eq.	equal to

3. All variables and keywords must be preceded and followed by delimiters. The most common delimiter is a space character, but others include ( ) , . = * /. Thus, the statement

```
if (x.le.20) goto 23
```

has extra unneeded spaces after the If and before the Goto. However, the space after the Goto is necessary.

Example 4 Another Method of Constructing Loops

Function

This example performs exactly the same as the previous example using the Loop - Endloop statement.

```
* Example 4

x=12
loop
y=x*x
print,x,y
x=x+1
quitif x=21
endloop
end
```

Notes:

1. The Quitif statement

   ```
   quitif x=21
   ```

   causes the FORTRAN processor to complete the repetition of the loop if x has the value 21 when the Quitif statement is executed. Control proceeds to the statement following the Endloop.

2. The Quitif statement can appear anywhere in the loop.

3. The x=21 in the Quitif statement is also called a *relational expression* and has the value *true* or *false*. It need not be enclosed in parentheses as was the case with the logical If statement. The equal sign, =, can be used instead of .eq.. The other relational operators can be used as follows:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<>	not equal to

Example 5 Indentation, Separators

Function

This example is identical to Example 4. However, all statements in the loop are *indented* two spaces to accent those which are to be repeated.

```
* Example 5

x = 12
loop
  y = x*x
  print,x,y
  x = x+1
  quitif x = 21
endloop
end
```

Notes:

1. All statements can have as many leading blanks as are desired.

2. Blanks may appear *within* statements to make the program more readable. However, blanks may not be inserted within variable names or keywords. Thus, the Print keyword cannot be written as

```
pr int
```

Example 6 Expanding Previous Examples

Function

This example simply expands the previous examples to include the cube of the integers from 12 to 20, inclusive.

```
* Example 6

x = 12
loop
  y = x*x
  z = x*x*x
  print,x,y,z
  x = x+1
  quitif x = 21
endloop
end
```

Notes:

1. A third variable z is introduced.

2. The statement z=x*x*x could have been written as z=x*y.

3. As outlined in a previous example, the relational expression does not have to be enclosed in parentheses. However, there must be a blank between Quitif and x=21. (Without the blank, the FORTRAN processor interprets quitifx=21 as a variable followed by an equal sign followed by a constant, namely an assignment statement.)

Example 7 Character Strings

Function

This example introduces a heading for the previous example.

```
* Example 7

print,"Table of Squares and Cubes"
x = 12
loop
  y = x*x
  z = x*x*x
  print,x,y,z
  x = x+1
  quitif x = 21
endloop
end
```

Notes:

1. The "Table of Squares and Cubes" is called a *character string constant* and will print *literally* as it appears in the program. Character string constants always contain a "string" of characters between quotes.

   e.g. 'ABCDEF' or "ABCDEF"

2. Either single quotes (') or double quotes (") may be used to delimit character strings.

3. The character string can contain any legal character.

4. If a quote is required as part of the string

   e.g. IT'S

   this can be accomplished in several ways.

   e.g. "IT'S" or 'IT''S'

Example 8 Square Root Function (SQRT)

Function

This example calculates a table of square roots of x, for x having values from 1 to 30, inclusive.

```
* Example 8

x = 1
loop
  y = sqrt(x)
  print,"Square Root of ",x," is ",y
  x = x+1
  quitif x = 31
endloop
end
```

Notes:

1. SQRT is known as an *intrinsic function* and will compute the square root of the quantity in parentheses, provided the quantity is not negative.

2. Other intrinsic functions such as SIN and COS are available in FORTRAN.

3. Results are printed rounded, with at most, 8 digits.

Example 9 SINE/COSINE Functions (SIN,COS)

Function

This example calculates a table of sine and cosine values.

```
* Example 9

print,"This is a Table of Sines and Cosines"
x = 0
loop
  y = sin(x)
  z = cos(x)
  print,x,y,z
  x = x+.1
  quitif x = 2.1
endloop
end
```

Notes:

1. When you run this program, it does not stop and must be terminated using the "RUN STOP" key. This is because numbers are stored somewhat inaccurately by computers. For example, the fraction "one third" is written as .3333333 to seven figures in decimal notation, and this result is slightly incorrect. Computers work internally in *binary* notation and similar slight inaccuracies occur. Thus, 2.1 is probably stored something like

   2.09999999

   Since x=2.1 is never true, the program does not terminate. To correct this problem, replace the quitif line with

   quitif x>2.01

   and run the program again.

Example 10 Declaration Statements

Function

This example is similar to Example 8 and is used to introduce reals and integers.

```
* Example 10

real y
integer x
x = 1
loop
  y = sqrt(float(x))
  print,"square root of ",x," is",y
  x = x+1
  quitif x = 31
endloop
end
```

Notes:

1. Two declaration statements have been added to the program, declaring y to be a real variable and x to be an integer variable. In the absence of any declaration, variables beginning with the letter I, J, K, L, M and N are integer and all others are real.

2. Real variables are used to hold real constants.

3. Real constants contain a decimal point and a fractional part following the decimal.

   e.g. 14.56 13.0 -.9743962 0.0

4. Real constants are also used to contain very large or very small numbers. These values are represented using the FORTRAN exponent notation; e.g.,

   3.19764*10**7 3.19764E07
   .9762583*10**-3 .9762583E-03

5. Integer variables are used to hold integer constants.

6. Integer constants do not explicitly contain a decimal point, e.g.,

```
39  46572  -65  0
```

7. While it is not important to know the various representations of real and integer constants, it is important to realize that for the same constant they are not represented in the same way. Thus the integer constant "1" is not the same as the real constant "1.".

8. The SQRT function requires a real value as its argument. The function FLOAT converts an integer to a real value.

9. Functions may have other functions as arguments, e.g.,

```
SIN (SQRT(Y))
SIN (COS(SQRT(Y)))
```

10. The output for x does not contain a decimal.

11. Real and integer variables may be used in the same program and in fact in the same expression.

Example 11 Input

Function

This program is an endless loop which requests the user to "Type in x", and it returns the value of the cube of x.

```
* Example 11

print,"Test simple input from the keyboard"
loop
  print,'Type in X'
  read,x
  y = x**3
  print,x,' cubed  = ',y
endloop
end
```

Notes:

1. The purpose of the statement

   ```
   print 'Type in x'
   ```

   is to *prompt* the user; that is, this statement reminds the user that a value must be typed into the keyboard, followed by carriage return (RETURN key).

2. The statement

   ```
   read,x
   ```

   inputs a value which may or may not contain a decimal point.

3. This program is an endless loop and must be terminated with the "RUN STOP" key.

4. The operator ** is the exponentiation operator.

Example 12 Exiting a Program due to an Input

Function

This program operates as the previous example except that when you type -999, the loop is terminated.

```
* Example 12

print,"Test simple input from the keyboard"
loop
  print,'type in X'
  read,x
  quitif x = -999
  y = x**3
  print,x,' cubed = ',y
endloop
end
```

Example 13 Nested Loops

Function

This example computes a set of tables of squares of x, with the starting value of x, increment of x, and number of entries being prompted as input from the keyboard.

```
* Example 13

loop
  print,'start x at'
  read,x
  quitif x = -999
  print,'vary x by'
  read,xvary
  print,'number of values for x is'
  read,number
  loop
    y = x*x
    print,"x=",x," y=",y
    x = x+xvary
    number = number-1
    quitif number = 0
  endloop
  print,'job finished'
endloop
end
```

Notes:

1. This example introduces a "loop within a loop". The *inside loop* is the five statements between the inside Loop - Endloop pair. The *outside loop* contains all those statements between the outside Loop - Endloop pair, which *includes* the inner loop.

2. To terminate, type -999 when the initial value of x is requested.

Example 14 IF...ELSE...ENDIF

Function

In this example, the user is asked to input a "dividend" and a "divisor". The "quotient" is printed, unless the divisor is zero, in which case an appropriate message is printed. To terminate, type -999 when the dividend is requested.

```
* Example 14

loop
  print,"input dividend"
  read,dividend
  quitif dividend = -999
  print,"Input divisor"
  read,divisor
  if (divisor = 0)
    print,"Divisor is zero"
  else
    quotient = dividend/divisor
    print,quotient
  endif
endloop
end
```

Notes:

1. The purpose of this example is to illustrate the use of the If - Else - Endif construction.

   a. The If statement always contains a relational expression which, when evaluated, is true or false.

   b. If the relational expression is true, all statements between the If and Else are executed.

   c. If the relational expression is false, all statements between the Else and Endif are executed.

2. The Else statement can be omitted. If so, all statements between the If and Endif are executed when the relational expression is true.

3. If's can be nested, if desired.

4. The condition on this type of If need not be enclosed in parentheses.

Example 15 Character Variables and Concatenation

Function

The user is prompted to type in his first name, followed by his last name. The computer "composes" the two names into a single string and prints it.

```
* Example 15

character firstname,lastname,fullname
print,"Concatenation of character strings"
loop
  print,"What is your first name?"
  read,firstname
  quitif firstname = "quit"
  print,"What is your last name?"
  read,lastname
  fullname = firstname // " " // lastname
  print,"Your full name is ",fullname
endloop
end
```

Notes:

1. Variables capable of being assigned character strings are said to be character variables which must be declared in a *character* declaration statement.

2. A declaration statement (real, integer or character statement) may contain a list of variables.

3. The two names are assigned to two character variables namely, firstname and lastname.

4. The // operator causes the two strings before and after it to be combined into one string (concatenation), with no space between them. Thus, in order to have a blank space between the two names, it is necessary to "add" three strings together, with the center one being a character string constant containing a single blank.

Example 16 Reading and Printing Various Types of Data

Function

This example shows various situations that occur when reading and printing data.

```
* Example 16

character string1,string2
read,a,b,i
print,a,b,i

read,string1,string2
print,string1,string2

read,string1,i
print,string1,i

end
```

Notes:

1. When reading data, a line is viewed as a number of zones. Two consecutive zones are separated by a comma. A zone is used for each item in the Read statement. When the zones in a line have all been used and more input items exist, the Read statement accepts another line to be processed.

2. The first Read statement asks for two real values and an integer value to be input. The values can be entered on one line separated by commas.

   e.g. 6.5,9.432, 77

3. The second Read statement asks for two character strings to be input. They could be input as two separate lines or the two strings can be placed on one line if they are separated by a comma.

   e.g. ABCD,QWERTY

4. The output from the second Print statement has the two strings concatenated.

   e.g. ABCDQWERTY

5. The third Read statement asks for a character string followed by an integer. These could be input as two lines or the string and the integer could be placed on one line with the character string and the integer separated by a comma.

6. Leading and trailing blanks are not ignored. If a comma is desired with a character string, the string should be enclosed in quotes.

7. A null string may be entered by a zone with no characters in it. This can be accomplished by entering two consecutive commas or by entering a blank line.

Example 17 Substring Operation

Function

This example assigns a value ABCDEFGHIJ to the character variable alpha, then *extracts* 4 characters from within the string, beginning at the third character and prints them as a new string from newstring.

Then, the program prints all the characters, one at a time.

```
* Example 17

character alpha,newstring
alpha = "ABCDEFGHIJ"
newstring = alpha(3:6)
print,newstring
n = 1
loop
  newstring = alpha(n:n)
  print,newstring
  n = n+1
  quitif n = 11
endloop
end
```

Notes:

1. The substring operation is defined as follows:

   ```
   character variable( M : N )
   ```

   The operation causes the Mth through the Nth characters from the character variable to form a new string. If M=N then the string consists of one character (the Mth). If M>N or M<=0, an error results.

2. One can also assign a character string to a substring. To illustrate this, try the following assignment and print the new value of newstring (newstring should be defined previously).

   ```
   newstring( 5 : 7 ) = 'eee'
   ```

Example 18 Expand on Example 17

Function

The character string, ABCDEFGHIJ, is assigned to alpha. The first print line will contain the first character, the second print line will contain the first two characters, etc., with the 10th line containing all 10 characters in the string.

```
* Example 18

character alpha,newstring
alpha = "ABCDEFGHIJ"
n=1
loop
  newstring = alpha(1:n)
  print,newstring
  n=n+1
  quitif n=11
endloop
end
```

Example 19 Length Function (LEN)

Function

The user is asked to type in his name. The program prints the name vertically, one character per line.

```
* Example 19

character name,letter
loop
  print,"What is your name?"
  read,name
  quitif name="quit"
  m = len(name)
  n=1
  loop
    letter = name(n:n)
    print,letter
    quitif n=m
    n=n+1
  endloop
endloop
end
```

Notes:

1. As each name will be of different length, we use the LEN function.

   LEN(character variable)

   returns the number of characters in the string.

Example 20 The DO Statement

Function

This example produces the same output as the previous example. It introduces the Do statement.

```
* Example 20

character name,letter
loop
  print,"What is your name?"
  read,name
  quitif name="quit"
  m = len(name)
  do n =1,m
    letter = name(n:n)
    print,letter
  enddo
endloop
end
```

Notes:

1. In previous examples, before entering the loop we have initialized the value of n to be 1 and the value of m to be the length of the string. Within the loop we have incremented the value of n by 1 and then tested if n was equal to m.

   The DO statement

   ```
   do n=1,m
   ```

   performs the same function. It initializes n to 1, tests if it is greater than or equal to m and exits the loop if this condition is true. The enddo indicates end of the loop.

2. The Do statement may be inside another loop. In fact Do's themselves may be nested.

3. The value to be incremented is assumed to be 1. However, any increment may be used by adding a third parameter to the Do statement, e.g.,

```
do i=1,99,2
do i=99,1,-2
```

4. The variable i is said to be the Do parameter, as it is used to control the execution of the loop. It is illegal to assign a value to a Do parameter while executing the associated loop.

5. The Do parameter may also be a real variable; e.g.,

```
do x = .1,2.5,.01
do x = 79.45,70.,-.01
```

The preceding examples illustrate Do statements with a real variable x.

Example 21 Reverse 3 Characters in a String

Function

This example asks the user to input a three-letter word. The program prints the word with the letters in reverse order.

```
* Example 21

character name,c1,c2,c3
loop
   print,"what is your name?"
   read,name
   quitif name="quit"
   c1=name(3:3)
   c2=name(2:2)
   c3=name(1:1)
   print,c1,c2,c3
endloop
end
```

Example 22 Reverse the Characters in any String

Function

This example asks the user to input a name. The program prints the name with the letters in reverse order.

```
* Example 22

character name,reverse
loop
  print,"what is your name?"
  read,name
  reverse = name
  quitif name="quit"
  n = len(name)
  do i=1,n
    reverse(n-i+1:n-i+1) = name(i:i)
  enddo
  print,reverse
endloop
end
```

Notes:

1. The statement

   ```
   reverse = name
   ```

   is necessary to initialize the character variable reverse to the same length as name.

2. The substring feature can be used to the left of the assignment operator.

Example 23 Simple FORMAT

Function

This example introduces format strings for the three data types.

```
* Example 23

character string
x = 12.75
print "(' ',f6.2)",x

i = 295
print "(' ',i5)",i

string = "hello"
print "(' ',a6)",string
end
```

Notes:

1. When the program is run, the output will be the three lines

   ```
   b12.75
   bb295
   HELLOb
   ```

   b represents the blank character.

2. The Print statements have been modified to include a character string immediately following the word print. This string, called a format string, contains a number of format codes separated by commas. The format codes may be enclosed in parentheses.

3. Format strings will let us print (and read) data according to our own rules as compared to rules set up by the FORTRAN processor. Thus in the above example we can print x as b12.75 instead of 12.750000.

4. The string 'b', is used as a special control character for vertical spacing of the output. If the character is blank, output is single-spaced, if it is 0 the output is double-spaced and if it is - the output is triple-spaced. The character 1 causes the output to display at the top of the screen. While Print statements

do not need to have a control character, it is usually advisable that they be included.

5. The format code F6.2 is used to print real constants. The value is printed using the first 6 print positions, with 2 decimal places, and the constant is right-justified within the 6 position field and is padded on the left with blanks.

6. The format code I5 is used to print integer constants. The value is printed using the first 5 positions and the constant is right-justified in the 5 position field and is padded on the left with blanks.

7. The format code A6 is used to print character strings. The value is printed using the first six positions and the string is left-justified in the 6 position field and is padded on the right with blanks.

Example 24 E and F FORMAT

Function

This example illustrates the use of format strings by printing real quantities using F and E formats.

```
* Example 24

x=26.458
print,x
print "(' ',f6.3)",x
print "(' ',f8.4)",x
print "(' ',f5.2)",x
print "(' ',f4.3)",x

print "(' ',e14.8)",x
print "(' ',e14.3)",x
print "(' ',e7.5)",x

end
```

Notes:

1. F format is used to print real values in a "normal" notation.

2. E format is used to print real numbers using the E-type or scientific notation.

3. F8.4 causes an extra zero to be added to the fractional portion.

4. F5.2 specifies only 2 digits following the decimal causing the value 26.46 to be printed. Note that rounding has occurred.

5. F4.3 asks that a 5 digit field plus a decimal be printed in a 4 position field. Since this is not possible, an error is indicated by filling the field with *'s.

6. E14.8 causes the value 0.2645800E+02 to be printed. E format causes the values to be printed in normalized form: the most significant digit is immediately to the right of the decimal.

7. E14.3 specifies that only 3 significant digits are to be printed. Again the value is rounded.

8. E7.5 does not allow sufficient positions to print the value, the 'E', and the exponent. Thus, *'s are placed in the field.

Example 25 I and A FORMAT

Function

This example illustrates the use of format strings by printing integer constants and characters strings.

```
* Example 25

character string

n=32
print,n
print "(' ',i5)",n
print "(' ',i1)",n

string="junk"
print,string
print "(' ',a5)",string
print "(' ',a3)",string
end
```

Notes:

1. I5 causes the value 32 to be printed with 3 blanks on the left.

2. I1 does not allow enough space for the constant and *'s are printed to indicate this condition.

3. A4 causes the string "junk" to be printed with 1 blank on the right.

4. A3 does not allow enough space for the string. However, in this case the left-most 3 characters of the string are printed.

Example 26 Mixed FORMAT

Function

This example shows use of a number of format codes.

```
* Example 26

character string

n=32
string="hello"
x=26.458
print,n,string,x
print "(' ',i5,a8,f9.3,e14.8)",n,string,x,x
print "(' ','n=',i5,'x=',f6.3)",n,x

end
```

Notes:

1. Format codes of different types may be used in the same format string.

2. Character strings inserted in the format string are printed.

Example 27 Test REAL FORMAT Output

Function

This example permits the user to enter a particular real number and to display it using various format statements which are requested as input.

```
* Example 27

character form
loop
  print,"Give me a real number"
  read,value
  quitif value=-999
  loop
    print,"Give me a format for a real number"
    read,form
    quitif form="quit"
    print form,value
  endloop
endloop
end
```

Note:

1. This example can be used to provide "drill" with respect to the rules of format.

2. The format string can be stored in a character variable.

Example 28 Test CHARACTER FORMAT Output

Function

This example operates identically to Example 27 except that the variables involved are *character* variables.

```
* Example 28

character form,string
loop
  print,"Give me a character string"
  read,string
  quitif string="zzz"
  loop
    print,"Give a format for a character string"
    read,form
    quitif form="quit"
    print form,string
  endloop
endloop
end
```

Example 29 Arrays, Printing a Blank Line

Function

This example requests the user to enter exactly 10 names. It then prints them in reverse sequence.

```
* Example 29

character names(10)
character name
do i=1,10
  print,"name please"
  read,name
  names(i) = name
enddo

print
print,"The names in reverse order are"
do i=10,1,-1
  print,names(i)
enddo
end
```

Notes:

1. The names are stored in an *array* which is specified using the declaration statement. This statement creates a ten element array:

   names(1),names(2), ... ,names(10)

2. Each element of the array is able to hold a character string, with each string being of a different length, if desired.

3. Numeric arrays can be set up using the real or integer declaration statements.

4. Arrays can have up to seven *dimensions*.

5. The print statement with no list causes a blank line to be printed.

Example 30 Select Names from an Array

Function

The user enters 10 names. Then he is asked for an integer between 1 and 10. If he enters 5, the 5th name is printed, etc.

```
* Example 30

character names(10),name

do i=1,10
  print,"Name please"
  read,name
  names(i) = name
enddo

loop
  print,"Enter a number between 1 and 10"
  read,i
  quitif i=-1
  print,"The i'th name is ",names(i)
endloop
end
```

Note:

1. The declaration statement may be used to declare the table names as well as the string name. The variables are separated by a comma.

Example 31 Printing an Array

Function

The program asks the user to input 10 names. These names are then printed in the same order.

```
* Example 31

character names(10),name

do i=1,10
   print,"Name please"
   read,name
   names(i) = name
enddo

print,names
end
```

Note:

1. The second print statement contains the name of the array. The entire array names is printed, i.e. all ten elements are printed. As many elements are printed in one line as the line can hold.

2. The output line(s) may be difficult to read since the strings are concatenated. The next example resolves this problem.

Example 32 Printing an Array with FORMAT

Function

This program functions as in the last example except the output is printed with format control.

```
* Example 32

character names(10),name

do i=1,10
   print,"name please"
   read,name
   names(i) = name
enddo

print "(' ',2a20)",names
end
```

Note:

1. The format code a20 is preceded by a field count namely 2 indicating that the format code is to be used twice. This causes 2 elements to print in each line; the format string is reused as often as necessary. Each time it is reused a new line is signalled.

2. If any name contains more than 20 characters, the left-most 20 are printed.

Example 33 More I/O with Arrays

Function

The example reads 10 values, stores them in an array, and then prints the values.

```
* Example 33

character nametable(10)

print,"enter 10 names - 1 per line"
read,nametable

print "(' ',3a20)",nametable
end
```

Note:

1. If we wish to read or print all the elements of an array, we can use the variable nametable without subscripts.

2. The names are printed 3 per line.

Example 34 Even more I/O with Arrays

Function

The example reads a number of values which partially fill an array and then prints the values.

```
* Example 34

character nametable(100)

print,"enter the number of names - max 100"
read,number
print,"enter",number,' names - 1 per line"'
read,(nametable(i),i=1,number)

print
print,"The",number," names are"
print "(' ',2a10)",(nametable(i),i=1,number)
end
```

Notes:

1. Both the read and print use a do loop as part of the statement.

2. The input data may contain more than one name per line. This is particularly useful when entering numeric constants.

3. The output names can be printed more than one per line.

Example 35 REMOTE BLOCKS

Function

This example invites the user to submit 10 names which are stored in an array and then are printed in reverse order.

```
* Example 35

character nametable(10),name
execute readdata
print,"Names in reverse order"
execute printdata
stop

remote block readdata
do i=1,10
  print,"Name please"
  read,name
  nametable(i) = name
enddo
endblock

remote block printdata
do i=10,1,-1
  print,nametable(i)
enddo
endblock

end
```

Notes:

1. The main purpose of this example is to introduce remote blocks. We define two remote blocks namely, readdata and printdata.

2. Remote blocks are defined by the Remote Block statement which includes the name of the block.

3. A remote block is terminated by an Endblock statement.

4. A remote block is *called* by including the name in an execute statement.

   e.g. execute readdata

5. When a block is called by the Execute, control passes to the first statement of the block. The statements are executed in sequence until the Endblock statement is encountered. Control then passes to the statement immediately following the Execute.

6. All variables in the calling program are available to the remote block when it is executing. Similarly, all variables assigned values in a remote block have the updated values after execution of the block is complete.

7. This type of use of a call permits one to *modularize* the program and is a matter of programming *style*.

8. Remote blocks cannot be nested. In other words, a remote block cannot be placed within another remote block.

9. A remote block may be called using an Execute statement which is within a remote block.

Example 36 SUBROUTINE Subprograms

Function

Calculate the average of a set of 15 numbers. The subroutine subprogram is introduced.

```
* Example 36

real marks(15)
print,"Input 15 numbers"
read,(marks(i),i=1,15)
call average(marks,15)
end

subroutine average(marks,number)
real marks(15)
sum=0.0
do i=1,number
  sum=sum+marks(i)
enddo
answer=sum/number
print,"The average of",number," numbers is",answer
return
end
```

Notes:

1. The marks can be input more than 1 per line separated by commas.

2. The average is calculated in a special portion of the program called a subroutine subprogram.

3. The subroutine is "called" using the statement

```
call average(marks,15)
```

4. The statement

```
subroutine average(marks,number)
```

gives the subprogram a name, average, and indicates which data is required in this case to calculate the average.

5. After the average is calculated and printed, control "returns" to the statement following the call.

6. Variables in the calling program are not available to a subroutine unless their values are passed as arguments. Thus, if variables in the calling program and the subroutine have the same name, they are treated as separate variables. The assignment of a value to a variable in the subroutine would have no effect on a variable with the same name in the calling program.

Example 37 Another SUBROUTINE Example

Function

Use the subroutine of the previous example to calculate the average of the first five numbers, first ten numbers, and first fifteen numbers.

```
* Example 37

real marks(15)
print,"Input 15 numbers"
read,(marks(i),i=1,15)
call average(marks,5)
call average(marks,10)
call average(marks,15)
end

subroutine average(marks,number)
real marks(15)
sum=0.0
do i=1,number
  sum=sum+marks(i)
enddo
answer=sum/number
print,"The average of",number," numbers is",answer
return
end
```

Note:

1. Three successive calls are made to the subroutine with the values 5, 10, and 15 respectively.

Example 38 FUNCTION Subprograms

Function

Do exactly as the previous example but also print the largest of the three averages.

```
* Example 38

real marks(15)
print,"Input 15 numbers"
read,(marks(i),i=1,15)
a1 = average(marks,5)
a2 = average(marks,10)
a3 = average(marks,15)
big = amax1(a1,a2,a3)
print,"The largest average is",big
end

function average(marks,number)
real marks(15)
sum=0.0
do i=1,number
  sum=sum+marks(i)
enddo
answer=sum/number
print,"The average of",number," numbers is",answer
average=answer
return
end
```

Notes:

1. Since we wish to calculate and print the largest average, we require some means of remembering each of the averages.

2. One method is to convert the subroutine subprogram to a function subprogram.

3. The function subprogram is called by the statement

```
a1=average(marks,5)
```

4. The statement

```
function average(marks,number)
```

gives the function a name and indicates the data required to calculate the average.

5. After calculating the average, it is assigned in the statement

```
average=answer
```

to the function name. Control then returns to the statement which called the function.

6. The value assigned to the function name can now enter into any arithmetic operation. In this case, it is assigned to a1, a2 and a3 respectively.

7. The intrinsic function amax1 is used to find the largest average.

Example 39 File Definition, OPEN, CLOSE, End of File

Function

This program creates a file on disk called namefile. The program requests the user to type names at the terminal. These names are printed as records on the disk file. When the name "quit" is input, the program halts.

```
* Example 39

integer status
character name
open (unit=2,file="namefile")

loop
  print,"name?"
  read,name
  quitif name="quit"
  write(unit=2) name
endloop
close (unit=2)

open (unit=2,file="namefile")
loop
  read(unit=2,iostat=status)name
  quitif status<>0
  print,name
endloop
close (unit=2)
end
```

Notes:

1. The Open statement is used to tell the system that we wish to use a file. If there is no such file as namefile, a new file is automatically created. Since the file will be used as output and if this is not a new file, all old information will be destroyed. The 2 in the open statement is a unit number assigned to the file *for this program only*. All other references to the file in this program are done using this number rather than the file name.

2. The file name (e.g., namefile) may be a character string of 16 or fewer characters.

3. When we want to write data into the file, we use a write statement. This causes a "record" containing the single field name to be written onto unit 2.

4. When the file has been written, it must be *closed*. A special "end of file indicator" is written at the end of the file.

5. When the file is reopened following the close, it is automatically positioned at the *beginning*.

6. As the records are read, one name at a time is assigned to the variable name.

7. When the read statement does not successfully complete, the variable status is assigned a non-zero value.

8. The file is closed before exiting the program.

Example 40 Multiple-field File Records

Function

This program creates two fields in each record in the file, namefile.

```
* Example 40

integer age,status
character name
open (unit=2,file="namefile")

loop
  print,"name?"
  read,name
  quitif name="quit"
  print,"age?"
  read,age
  write(unit=2) name, ",", age
endloop
close (unit=2)

open (unit=2,file="namefile")
loop
  read(unit=2,iostat=status)name,age
  quitif status<>0
  print,age,name
endloop
close (unit=2)
end
```

Note:

1. Each time the write statement is executed both name and age are written onto the file. Since we wish to read the file at some later point, a comma is placed between the name and age.

2. When the data is read we use *exactly* the same number and type of fields as when the data was written.

3. Note that we print age to the left of name to show that this ordering is independent of the order of the fields in the record.

Example 41 Reading the Entire Record

Function

Here we create a file, namefile, and subsequently read each record as a character string.

```
* Example 41

integer age,status
character name,form,record
form="(a20,'--',i3)"
open (unit=1,file="namefile")

loop
  print,"name?"
  read,name
  quitif name="quit"
  print,"age?"
  read,age
  write(unit=1,fmt=form)  name,age
endloop
close (unit=1)

open (unit=1,file="namefile")
loop
  read(unit=1,iostat=status)record
  quitif status<>0
  print,record
endloop
close (unit=1)
end
```

Notes:

1. Each record will be 25 characters in length. Name will be left-justified in the first 20 positions, the constant "--" will appear in positions 21 to 22 in each record, followed by the age right-justified in positions 23 to 25.

2. The entire record will be read and assigned to the character variable line. When it is printed, we see the format of each record as described above.

Notes:

Waterloo microFORTRAN

Reference Manual

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter A

Fundamental Concepts

A-1 Statements, Data Types and Expressions

A FORTRAN program is a series of statements as shown in the following example:

```
value = 71
cube = value * value * value
print, value, cube
end
```

There are four statements in the program. The first two statements are assignment statements which cause values to be assigned to variables when executed. The third statement, when executed, causes the current values of the two variables, "value" and "cube", to be displayed at the terminal. The fourth marks the end of the program.

Once the program has been entered into the computer (see the description of the Waterloo microEditor in the System Overview manual), the program may be executed by entering the RUN command. This will cause execution of the statements

starting with the first one in the program. In the preceding sample program, four statements would be executed in the following sequence:

VALUE = 71

The value 71 is assigned to variable named VALUE.

*CUBE = VALUE * VALUE * VALUE*

The value (71) of the variable VALUE is multiplied together twice to produce a result (357911). This resultant value is assigned to the variable CUBE.

PRINT, VALUE, CUBE

The execution of this statement causes the current value of VALUE (71) and CUBE (357911) to be displayed at the terminal.

END Statement

This statement causes execution of the program to stop.

The following output will be displayed on the terminal:

```
Executing...
 71.000000 357911.00
...Stop
```

The first message signals that program execution has started. The last message indicates that the execution of the program has completed.

It should be noted that the sample program performed a simple manipulation (computation of a cube) of numeric data. In subsequent sections we will discuss in detail numeric data and numeric variables.

Additional Rules About Programs:

(1) *Use of Spaces*

Spaces should be used within statements to clarify the program. Spaces cannot occur inside variable names, numeric constants, operators which are composed of more than one character, keywords, or statement numbers.

(2) *Comment Statements*

When the first character in a statement is an asterisk (*), the statement is treated only as documentation and is ignored during the execution of the program.

(3) *Null Lines*

Lines containing only space characters may be entered anywhere in a program. These lines are ignored during the execution of a program. These null lines may be used to increase the readability of programs.

(4) *END Statement*

The END statement is used to mark the end of a program unit.

(5) *Statements*

Each statement starts on a new line.

(6) *Statement Numbers*

Most statements may be preceded by an integer called a statement number. FORMAT statements must have a statement number. PROGRAM, SUBROUTINE and FUNCTION statements may not have statement numbers.

(7) *Continued Statements*

Occasionally, a statement contains too many characters to be conveniently placed on a single line. A statement may be continued by placing an ampersand character (&) as the first character on the continued line, thus

```
A =
& B
& + C
```

is equivalent to

```
A = B + C
```

Only the first line in a continued statement may have a statement number.

Statement elements may not be continued across separate lines. It is illegal to split across a continued line variable names, keywords, quoted strings, constants or compound operators.

A-2 Assignment Statement

Syntax: variable = expression

Examples: X = 493 * 4 + 2
TITLE = 'ABC' // PAGE

The assignment statement specifies an expression (to the right of the assignment operator (=)) which is used to calculate a resultant value to be assigned the variable (to the left of the assignment operator (=)). In succeeding sections we will discuss in detail the rules for specifying variables and expressions.

The expression is evaluated and assignment occurs when the statement is executed. This means that a different result could be assigned each time the statement is executed since the values of variables used in the expression to the right of the assignment operator (=) could have different values at each execution of the statement.

CHARACTER expressions (see CHARACTER data) may only be assigned to CHARACTER variables. Numeric data (see Numeric data) may only be assigned to numeric variables. When a REAL value is assigned to an INTEGER variable, the value to be assigned is converted to an integer by truncating the fractional part from the value.

A-3 Variable Names

A variable name is specified as a sequence of alphabetic (A-Z, a-z) characters, digits (0-9), underscore (_) characters, and dollar-sign ($) characters. The first character in the name must be alphabetic. The name may have as many characters as desired as long as it is entered on a single line. It is a good idea to use names which clearly indicate the use of the variable. Examples of variables are as follows:

SumofSquares
PageNumber

LineCounter
StudentAverage
TitlePage
CustomerName
DataRecord

Note that the names can be used to clearly define the usage of the variable.

At any time there is one value associated with a variable. These values may be assigned to variables by the execution of FORTRAN statements such as assignment statements. When a program begins execution, none of the variables have been assigned values and all are said to be undefined variables. An error will be detected if one of these variables is used in an expression before a value has been assigned to it.

Certain names, such as PRINT and END, are reserved for use as FORTRAN keywords. These names (see RESERVED WORDS) cannot be used as variable names. These names may be entered with any combination of upper and lower case letters. They will, however, be displayed in the program as if they had been entered entirely using lower case letters.

A-4 Data Types in General

There are three different types of data which may be manipulated in FORTRAN programs:

(1) *INTEGER Data*

These data elements are integers or whole numbers (no fractional part). When expressions involving only integers are evaluated, the result is an integer, even if the algebraic result would involve a fraction.

(2) *REAL Data*

These data elements may contain fractions. The precision of these numbers is limited to a number of significant digits (say eight) depending upon the type of the computer used. Consequently, the results of computations are not necessarily absolutely accurate.

(3) *CHARACTER Data*

These data elements are sequences of zero or more characters.

Successive sections will discuss each of these data types in detail.

Each variable is defined to contain values of one of the three data types. This may be accomplished by specifying the variable in INTEGER, REAL, or CHARACTER statements located at the start of the program. When a variable is used in a program and is not explicitly mentioned in one of those statements, it is assumed to be a REAL variable unless the first character of the name is one of the letters I,J,K,L,M,N,i,j,k,l,m or n. In the latter situation, the variable is an INTEGER variable.

A-5 Type Statements

Syntax: INTEGER variable,variable,....,variable
REAL variable,variable,....,variable
CHARACTER variable,variable,....,variable

Examples: integer PageCount
real Cost,ProfitMargin
character StudentName,PrintLine

The type statements are used to declare the type of variables (and of functions and arrays as discussed later). A variable that is not declared in this way is assumed to be either REAL or INTEGER (see DEFAULT TYPE) as established by the first letter in the name. It is an error to declare the same variable more than once in a program unit.

A-6 Numeric Data

Numeric data can be represented as either INTEGER data or REAL data. INTEGER data represents numbers which have no fractional part while REAL data is used to represent numbers which may have fractional parts.

The magnitude of numbers which can be represented is limited by the computer hardware. Typically much larger numbers can be represented by REAL data (see SYSTEM DEPENDENCIES) than by INTEGER data. For example, in the Motorola 6809 computer the range of INTEGER values is -32767 to 32767 and the largest REAL value is approximately ten to the 35th power (the smallest is the negative value of this magnitude).

The precision of REAL numbers is limited to several digits (the implementation for the Motorola 6809 limits the precision to approximately 9 digits). Consequently, each REAL number should be viewed as an approximation to an actual value. An INTEGER value is always precise since it represents a whole number.

The smallest absolute value (apart from zero) which can be represented is also limited. For example, in the implementation for the Motorola 6809 the smallest positive value is approximately one divided by 10 to the 35th power.

Because REAL values are to be viewed as approximations, the results of computations involving these values are also approximations. It is well known that the potential error increases as more calculations involving these values are performed. In most practical problems, however, the amount of error is not significant compared to the result. The discipline of Numeric Analysis studies this problem. It is beyond the scope of this manual to discuss the problem in detail.

INTEGER constants are written as a sequence of digits:

```
43    -17    2974
```

The preceding examples illustrate valid INTEGER constants. It is erroneous to include commas (,) or decimal points (.) in the numbers.

```
4,327    69.    2,231.
```

The preceding are all illegal INTEGER constants.

REAL constants may be written with a decimal point (.):

```
47.63    .97    263.    -27.4
```

The preceding are all valid REAL constants. Commas (,) may not be used in REAL constants. In order to compactly represent very large or small numbers, scientific notation may also be used to represent REAL values.

Scientific Notation	*Value*
43.E4	430000
43.E-7	.0000043
-16.2E+5	-1620000

In this form, an integer or decimal value is followed by an E and a second integer value. The actual value to be represented is determined by taking the number

preceding the E and multiplying it by 10 raised to the power of the integer following the E.

There are no rules which specify where scientific notation should or should not be used. A programmer should use whichever form makes the program easier to understand by another person.

A-7 CHARACTER Data

CHARACTER data is a sequence of 0 to 32767 characters. CHARACTER constants may be used to specify CHARACTER data:

```
'HI THERE'
"Report Title"
"John Smith, President"
"I can't leave now"
''
""
```

The preceding examples illustrate valid CHARACTER constants. The last two constants are character strings of length zero, called null strings. Each constant is enclosed by a pair of either single (') or double (") quotation characters. The same character must be used to start and terminate the constant. Thus, to include one of the quotation characters in the constant, the other quotation character can be used to mark the start and the end of the constant.

Alternatively, a quotation character may be included as part of a string by including the quotation character (which delineates the string) twice in successive positions.

```
'I''m programming'
""""Howdy""""
```

The preceding two examples represent the following character strings:

```
I'm programming
"Howdy"
```

A-8 Expressions

Expressions are combinations of operators, variables, constants, and function references which specify how a computation is to be performed. The order by which this computation occurs is determined by parentheses in the expression and the priority of the operators involved. The following table gives the priority of the operators:

Priority	*Operator*
7	enclosed in parentheses
6	**
5	*, /
4	+, -
3	comparisons
2	//, .AND.
1	.OR.

Operations enclosed in parentheses are performed before any other operations. Operations with a higher priority are performed before those with a lower priority. When two operations have the same priority, the leftmost is performed first. In the evaluation, the current values for variables specified in the expression are used in the computation.

The evaluation of an expression can be viewed as a number of successive reductions of sub-expressions, according to the priorities, until only a single value (the result) remains. Consider the following expression:

```
-A**((2+B)/C)*2.5*C
```

where the REAL variables A,B, and C have values 4., 6., and 2., respectively. The first step is to substitute the values for the variables used in the expression:

```
-4.**((2.+6.)/2.)*2.5*2.
```

The evaluation then proceeds according to the priority of the operations.

Expression	*Operator Applied*
−4.**((2.+6.)/2.)*2.5*2.	(Start)
−4.**({8.} /2.)*2.5*2.	+
−4.** {4.} *2.5*2.	/
−{256.} *2.5*2.	**
− {640.} *2.	*
− {1280.}	*
{−1280.}	−

In the preceding example, each evaluation result is shown in braces ({}). The second column shows the operation performed at each step. In the next section, precise definitions of the operators are specified.

A-9 Operators

In this section, each of the operators is defined. Each operation involves one or two operands which are indicated by NUMB for an INTEGER or REAL datum and by CHAR for a CHARACTER datum.

Exponentiation: NUMB ** NUMB

The first value is raised to the power of the second value. When both operands are INTEGER, the result is an INTEGER value. Otherwise, the result is a REAL value. It is an error if the first number is negative and the second number is not an INTEGER value. It is also illegal to attempt to raise zero to the zero power.

Multiplication: NUMB * NUMB

The result is the product of the two operands. When both values are INTEGER, the result is INTEGER; otherwise, the result is REAL.

Division: NUMB / NUMB

The result is the quotient of the two operands. When both values are INTEGER, the result is INTEGER, otherwise, the result is REAL. The second operand is divided into the first. It should be noted that the INTEGER result of this operation is obtained by removing the fractional part of the algebraic result. Thus, 8/3 will result in 2. The result is not rounded. An error will be detected when the second value is zero.

Addition: NUMB + NUMB

The result is the sum of the two operands. When both values are INTEGER, the result is INTEGER; otherwise, the result is REAL.

Subtraction: NUMB - NUMB

The result is the difference of the two operands. When both values are INTEGER, the result is INTEGER; otherwise, the result is REAL. The second operand is subtracted from the first.

Unary Minus: - NUMB

The result has the same size as the operand and is given the opposite sign.

Unary Plus: + NUMB

The result is identical to the numeric operand. This operation has no effect and is included only to aid in writing readable programs.

Numeric Comparison

NUMB = NUMB or NUMB .eq. NUMB (equal)
NUMB > NUMB or NUMB .gt. NUMB (greater)
NUMB < NUMB or NUMB .lt. NUMB (less)
NUMB >= NUMB or NUMB .ge. NUMB (greater,equal)
NUMB <= NUMB or NUMB .le. NUMB (less,equal)
NUMB <> NUMB or NUMB .ne. NUMB (not equal)

The two numeric operands are compared to determine a result of one (relationship is true) or zero (relationship is false). The relationship to be tested is indicated by the comparison operator.

Care should be exercised when comparing two REAL numbers for equality or for non-equality. Because the numeric representation in a computer is accurate, at most, to a finite number of significant digits, two values which are algebraically equal may be computed as unequal.

Character Comparison

CHAR = CHAR or CHAR .eq. CHAR (equal)
CHAR > CHAR or CHAR .gt. CHAR (greater)
CHAR < CHAR or CHAR .lt. CHAR (less)
CHAR >= CHAR or CHAR .ge. CHAR (greater, equal)
CHAR <= CHAR or CHAR .le. CHAR (less, equal)
CHAR <> CHAR or CHAR .ne. CHAR (not equal)

The two character operands are compared to determine a numeric result of one (relationship is true) or zero (relationship is false). The relationship to be tested is indicated by the comparison operator.

When one string is shorter than the other, it is padded with space characters to be the length of the larger and then the comparison is performed.

Concatenation: CHAR // CHAR

The result of this operation is a character string composed of the contents of the first operand followed by the contents of the second operand; i.e.,

'WATERLOO' // ' FORTRAN'

results in a character string

'WATERLOO FORTRAN'

This operation is used to build up larger character strings from smaller character strings.

Logical AND: NUMB .AND. NUMB

The result of this operation is an INTEGER value of zero or one. If both the operands are non-zero, the result is one; otherwise, the result is zero.

Logical OR: NUMB .OR. NUMB

The result of this operation is an INTEGER value of zero or one. If either operand is non-zero, the result is one; otherwise, the result is zero.

Logical NOT: .NOT. NUMB

The result of this operation is an INTEGER value of zero or one. If the operand is zero, the result is one; otherwise, the result is zero.

A-10 Substrings

Syntax: variable(first:last)
or variable(first:)
or variable(:last)

---

where "variable" is a CHARACTER variable and "first" and "last" are numeric expressions.

A substring is used to represent part of a CHARACTER variable. The two numeric expressions, if present, are evaluated as integer values to be used as the first and last positions of the subsection of the CHARACTER variable. When the first number is missing, the value 1 is used. When the second number is missing the length of the character string is used as the second number. This is illustrated by the following table:

specification	*value*
C	ABCDEFG
C(3:5)	CDE
C(:3)	ABC
C(3:)	CDEFG

It is an error to specify a substring for an undefined CHARACTER variable, for a null string, or for a part of the substring beyond the bounds of a defined string.

A substring may be used in expressions wherever a CHARACTER value is appropriate. A substring may also be assigned a value, in which case the CHARACTER value is either truncated to be the size of the substring or padded with space characters to be the size of the substring. For example,

```
(1)  character S
(2)  S = 'ABCDEFG'
(3)  S(3:5) = 'XXXXX'
(4)  print, S
(5)  end
```

would produce the following output:

```
ABXXXFG
```

A-11 Examples of Expressions

In this section, several examples of expressions are given in order to illustrate the FORTRAN operations. Suppose that the following assignment statements have been executed in order to assign values to variables:

```
A  = 2
B  = 3
C  = 3
D  = 4
AS ='AAA'
BS ='BBBB'
CS ='CCCCC'
```

These preceding values will be used in the following examples. In each case the successive reductions to the final result are illustrated.

Example: (A+B)*-(D+C**2)

```
(2+3)*-(4+3**2)
 {5} *-(4+3**2)
 {5} *-(4+{9})
 {5} *- {13}
 {5} * {-13}
   {-65}
```

Example: AS // BS // CS

```
'AAA' // 'BBBB' // 'CCCCC
 {'AAABBBB'} 'CCCCC'
    {'AAABBBBCCCCC'}
```

Example: A=B .OR. C<>D

```
2=3 .OR. 3<>4
{0} .OR. 3<>4
{0} .OR. {1}
     {1}
```

Example: .NOT. (A+C=B+D)

```
.NOT. (2+3 = 3+4)
.NOT. (5   = 3+4)
.NOT. (5   = 7))
.NOT. 0
     {1}
```

A-12 Interrupting Programs

A program may be interrupted while it is executing by entering the "STOP" key found on the keyboard. Thus, a program in an "infinite loop" can be interrupted.

Chapter B

Structured Control Statements

B-1 What is Meant by Control

"Control" means the manner in which statements are selected for execution. Normally, statements are executed one at a time in the order in which they occur in the program. Thus, after one statement is executed, the next statement to be executed is normally the statement following the one just executed.

Certain statements cause the normal sequential progression of control to be altered. These statements may cause control to be altered from the next statement to another statement in the program. Consider the following program to print the squares of integers from 10 to 20.

```
(1)   integer Value
(2)   Value = 10
(3)   while Value < = 20
(4)         print, Value, Value * Value
(5)         Value = Value + 1
(6)   endloop
(7)   end
```

The parenthesized numbers to the left of the program are not part of the program and are used only for reference purposes in this manual. When the program is executed, lines (4) and (5) will be repeated 11 times. For each repetition, the variable VALUE will have a different value, starting with 10 and increasing by 1. Thus, when the statement on line (4) is executed each time, the current value of Value and the square of that value are printed.

The sample program illustrates a loop. The bounds of the loop are defined by the statements at line (3) and at line (6) respectively. The WHILE statement at line (3) specifies that the body of the loop (lines (4) and (5)) is to be executed while the value of Value is less than or equal to 20. As long as this condition is evaluated as true, control will pass to the statement following the WHILE statement. When the condition is evaluated as false, control will pass to the next statement following the ENDLOOP statement. Whenever the ENDLOOP statement is executed, control passes to the WHILE statement in order to test if the loop is to be repeated again.

The preceding example was intended to illustrate how the normal sequential flow of execution can be altered by certain FORTRAN statements. The remainder of this chapter describes some of the statements which can be used in this way. We have classified these statements as structured control statements.

It has been theoretically proven that the only necessary constructs required to write programs are the normal sequential flow of control, a way to program loops and a way to select different sequences of statements for execution. The first two mechanisms have already been illustrated in part. Programs that are written using only these mechanisms are usually called "Structured Programs". Consequently, we have termed the statements which are used to control program execution in this manner, "Structured Control Statements".

There also exist, primarily for historical reasons, other methods of altering control from the next sequential statement. Although they are not strictly required in the FORTRAN language, they have been included so that existing FORTRAN programs may be executed (see MISCELLANEOUS STATEMENTS).

B-2 Conditions

Several of the structured control statements to be discussed cause control to be altered depending upon the evaluation of a numeric expression called a condition. The condition is said to be false when the evaluation of the expression produces a result of zero; otherwise, the condition is said to be true. Several examples of conditions are:

```
X > 4
(PageCount > 55) .or. (FirstTime)
(SEX = 'F') .and. (SALARY > 5000.00)
```

In the syntactic descriptions to follow, "condition" is used to indicate a condition.

B-3 Loops with WHILE or LOOP

```
Syntax:    LOOP
     or    WHILE condition
             ....(body of loop)
           ENDLOOP
     or    ENDWHILE
     or    UNTIL
```

Loops have three parts: a statement (WHILE or LOOP) to mark the start of the loop; a number of FORTRAN statements to form the body of the loop; and a statement (ENDLOOP, ENDWHILE or UNTIL) to mark the end of the loop. The body of the loop is repeated while the numeric expression is non-zero at the start of the loop (WHILE statement) and/or the numeric expression is zero at the end of the loop (UNTIL statement). When the loop begins with a WHILE statement, an ENDWHILE statement may be used to mark the end of the loop, as an alternative to the ENDLOOP statement.

In the preceding section we have already seen an example of a loop using a WHILE statement to start a loop and a ENDLOOP statement to terminate the loop. When a WHILE statement is used to start a loop, an ENDWHILE statement may be used as an alternative for the ENDLOOP statement to mark the end of the loop. An equivalent program is as follows:

```
(1)  integer  Value
(2)  Value = 10
(3)  loop
(4)       print, Value, Value * Value
(5)       Value = Value + 1
(6)  until  Value > 20
(7)  end
```

In this case the loop was programmed with a LOOP statement to start the loop and an UNTIL statement to terminate a loop.

Depending on the data involved in the expression associated with a WHILE statement, a WHILE-ENDLOOP loop may have the body of the loop executed zero times. This will be the situation when the expression associated with the WHILE statement is initially false (zero). A LOOP-UNTIL loop, however, will always execute the beginning of the body of the loop at least once. This is because the WHILE statement is executed at the start of each iteration of a loop and because the UNTIL statement is executed at the completion of each iteration of a loop.

It is also possible to program WHILE-UNTIL loops. These loops are controlled by both the WHILE statement at the start of the loop and the UNTIL statement at the end of the loop. Consider the following program:

```
(1)  integer  Value
(2)  real Sum
(3)  Value = 0
(4)  Sum  = 0
(5)  while Sum < 1000
(6)       Value = Value + 1
(7)       Sum = Sum + Value
(8)  until Value > = 99
(9)  print, Value, Sum
(10) end
```

The preceding program may be used to determine the first two-digit integer for which the sum of the positive integers to that number exceeds one thousand. If no such integer exists, the sum of the integers from one to ninety-nine is printed.

It is also possible to program LOOP-ENDLOOP loops. In this case a statement in the body of the loop must be used to terminate the loop (see QUIT statement).

The body of a loop may itself contain a loop. In this case the inner loop is said to be nested inside the outer loop. Consider the following program:

```
(1)   integer  Year, Month
(2)   real Rate, Principal
(3)   Rate = .015
(4)   Principal = 1.00
(5)   Year = 1
(6)   while  Year < 2
(7)        print, 'Year = ', Year
(8)        Month = 1
(9)        while Month <= 12
(10)            Principal = Principal * (1. + Rate)
(11)            Print, Month, Principal
(12)            Month = Month + 1
(13)       endloop
(14)       Year = Year + 1
(15)  endwhile
(16)  end
```

The preceding program shows how a principal of one dollar appreciates in value at a rate of 1.5% computed monthly over a two-year period. Note the inner loop, lines (9) to (13), nested inside the outer loop, lines (6) to (15).

It is considered good programming practice to indent the body of loops. In this way, the program becomes more readable as the repeated sequences of statements are clearly marked by the indentation level.

B-4 Structured DO Loop

Syntax: DO var = first, last
or DO var = first, last, increment
.... (body of loop)
ENDDO

A structured DO loop specifies a loop which is repeated with a control variable "var" assigned a number of values starting with the value of the expression "first". At the end of each iteration of the loop, the value of the expression "increment" is added to the variable "var".

Before each iteration of the loop, the control variable is tested to determine if the body of the loop is executed another time. The repetition terminates, and control continues at the statement following the ENDDO when:

- if the increment is positive, when the value of "var" exceeds the value of "last"
- if the increment is negative, when the value of "var" precedes the value of "last"
- It is an error for "increment" to have a value of zero.

It should be noted that a DO loop may be executed zero times.

The values of "first", "last", and "increment" are calculated immediately before any repetitions of the loop are performed and are not recalculated at each repetition. If an expression for "increment" is not specified, then a value of 1 is used as the increment. Thus, the number of iterations of the loop is established before the loop is executed and any change to variables involved in the calculation of "first", "last", or "increment" will not alter this number of iterations.

The program in the preceding section can be rewritten as follows:

```
(1)   integer Year, Month
(2)   real Rate, Principal
(3)   Rate = .015
(4)   Principal = 1.00
(5)   do Year = 1, 2
(6)       print, "Year = ", Year
(7)       do Month = 1, 12
(8)           Principal = Principal * (1 + Rate)
(9)           print, Month, Principal
(10)      enddo
(11)  enddo
(12)  end
```

Note the convenience of DO loops for the loops in question.

When the control variable is an INTEGER variable, the expressions are treated as integer expressions by truncating any fractional parts of their values when necessary.

It is an error to assign a value to the control variable while executing the body of a DO loop using that variable.

B-5 IF, ELSEIF, ELSE, ENDIF Statements

```
Syntax:   IF          condition
              ....
          ELSEIF      condition    (optional)
              ....                 (optional)
          ELSEIF      condition    (optional)
              ....                 (optional)
          ELSE                     (optional)
              ....                 (optional)
          ENDIF
```

The IF statement and its associated statements are used to execute different sequences of statements, depending upon the alternatives in the data. Consider the following sequence of FORTRAN statements:

```
(1)  if ColourCode = 19
(2)       Colour = 'red'
(3)  elseif ColourCode = 23
(4)       Colour = 'blue'
(5)  else
(6)       Colour = 'unknown colour'
(7)  endif
```

The execution of this sequence will cause a CHARACTER variable Colour to be assigned a value of 'red', 'blue', or 'unknown colour' depending on the value of the variable ColourCode. Thus, the example illustrates a three-way choice, based upon the current value of the variable ColourCode.

The general form of an IF-group is shown by the syntax at the start of this section. An IF-group must start with an IF statement and must be terminated with an ENDIF statement. The simplest form is illustrated in the following example:

```
(1)  if SexCode = "male"
(2)     MaleCount = MaleCount + 1
(3)     MaleWages = MaleWages + Salary
(4)  endif
```

In this case, the expression associated with the IF is evaluated. If the result is true (non-zero), then the statements following the IF are executed. Otherwise, control passes to the statement following the ENDIF statement.

A second form of the IF may be used to distinguish between two alternatives.

```
(1)  if Salary > 5000
(2)     Type = 'executive'
(3)     ManagementCount = ManagementCount + 1
(4)  else
(5)     Type = 'worker'
(6)     WorkerCount = WorkerCount + 1
(7)  endif
```

In this situation one of two sequences of statements is selected. When the expression

in the IF statement evaluates as true (non-zero), the first sequence (2-3) is selected and then control passes to the statement following the ENDIF statement. When the expression in the IF statement is false (zero), the second sequence of code (5-6) is selected and then control passes to the statement following the ENDIF statement.

When several alternatives are possible, the ELSEIF statement can be used to select one of a number of alternatives.

```
(1)   if Code = 'add'
        ....statements for add processing
(2)   elseif Code = 'chg'
        ....statements for change processing
(3)   elseif Code = 'dlt'
        ....statements for deletion processing
(4)   else
        ....statements for error processing
(5)   endif
```

The preceding example illustrates a selection of one of four alternatives. The actual statements to do the processing for each of the alternatives have been left out for clarity. The value of the CHARACTER variable Code is used to select which of the alternatives is to be executed:

- when the expression in statement (1) is true (non-zero), the statements between statements (1) and (2) are executed and then control passes to the statement following the ENDIF statement.

- otherwise, if the expression in statement (2) is true (non-zero), the statements between statements (2) and (3) are executed and then control passes to the statement following the ENDIF statement.

- otherwise, if the expression in statement (3) is true (non-zero), the statements between (3) and (4) are executed and then control passes to the statement following the ENDIF statement.

- otherwise, the statements between (4) and (5) are executed and then control passes to the statement following the ENDIF statement.

As many ELSEIF statements as required can be associated with an IF statement. The ELSE statement, if present, must follow the ELSEIF statements for an IF group. When the ELSE statement is not included and all the expressions in the IF statement

and all ELSEIF statements are false, then none of the alternative sequences of statements are executed and control continues at the statement following the ENDIF statement.

It is a good idea to use the same indentation for each of the IF, ELSEIF, ELSE and ENDIF statements and to indent the alternative sequences of code. In this way the program becomes more readable.

B-6 Nesting Loops and IF-Groups

A program in Waterloo FORTRAN may be viewed as a number of "blocks" of statements. A block can be defined to be a program, the body of a loop, the alternatives in an IF-group, the body of a function or subroutine definition (see FUNCTIONS), or the alternatives in a GUESS/ADMIT).

A block can be nested inside another block, but cannot overlap only part of another block. Thus, loops can be nested inside loops, an IF group can be nested inside a loop, and a loop can exist inside one of the alternatives of an IF group. Consider the following example:

		Nest-level
(1)	while	1
	....	2
(2)	if	2
	....	3
(3)	else	2
	....	3
(4)	loop	3
	....	4
(5)	until	3
	.....	3
(6)	endif	2
	.....	2
(7)	while	2
	....	3
(8)	endloop	2
	....	2
(9)	endloop	1

The example illustrates a loop (1-9) which contains an IF group (2-6) and another loop (7-8). One of the alternatives (3-6) of the IF group contains a loop (4-5).

The following example illustrates an illegal nesting of blocks:

```
(1)  if
       .....
(2)    loop
       .....
(3)  else
       .....
(4)    until
       .....
(5)  endif
```

The loop (2-4) neither contains nor is contained in the two alternatives (1-3, 3-5) of the IF group (1-5). In such situations, Waterloo FORTRAN will display an error message diagnosing the error.

B-7 QUIT and QUITIF Statements

```
Syntax:   QUIT
          QUITIF condition
```

It is possible to exit from an IF, LOOP, WHILE, or DO block by using the QUIT or QUITIF statements. Consider the following example:

```
(1)  integer i, Sum
(2)  Sum = 0
(3)  do i = 1, 100
(4)      Sum = Sum + i
(5)      quitif Sum > 500
(6)      print, i, Sum
(7)  enddo
(8)  end
```

The example will print the sums of the integers 1,2,... until the sum exceeds 500. The QUITIF statement (line 5) causes control to continue following the end of the block (line 7) when the associated expression evaluates as true.

In general, the QUIT or QUITIF statements may cause control to continue following the statement which makes the end of the associated block. The QUIT statement causes this to occur unconditionally; the QUITIF causes control to continue in this way only when the associated condition is true (non-zero). When used with block identifiers (described in the next section), the QUIT and QUITIF statements may be used to exit outer blocks when the statements are contained in nested blocks.

B-8 Block Identifiers

Syntax: : name

It is often useful to name a block. This provides not only a documentation aid, but also a means of exiting from within nested blocks.

```
(1)  integer i, Sum
(2)  Sum = 0
(3)  do i = 1, 100              : SumLoop
(4)      Sum = Sum + i
(5)      quitif Sum > 500 : SumLoop
(6)      print, i, Sum
(7)  enddo                      : SumLoop
(8)  end
```

The preceding example is identical to the one in the preceding section except that the loop has been given a block identifier "SumLoop". This name is also referenced in the QUITIF statement (line 5). Names of blocks are formed according to the same rules as apply to variable names (see VARIABLE NAME). A block should not have the same name as a variable within the same program unit.

A block identifier, if specified on the first statement in a block, may be specified on any of the other statements which mark boundaries in that block. Of course, the same name must be used on each of these statements. In this way, IF-groups, loops, structured DO loops, and GUESS-groups may be named.

Another advantage of block identifiers is their use in conjunction with QUIT or QUITIF statements. Consider the following example:

```
(1) integer Year, Month
(2) real Principal, Rate
(3) Rate = .015
(4) Year = 1
(5) Principal = 1.00
(6) loop                                    :YearLoop
(7)    do Month = 1, 12
(8)       Principal = Principal * (1. + Rate)
(9)       quitif Principal > 2.0000 :YearLoop
(10)   enddo
(11)   Year = Year + 1
(12) endloop                          :YearLoop
(13) print, year, month
(14) end
```

The program determines the year and month when a single dollar will double in value as it appreciates in value at a rate of 1.5% each month. The QUITIF statement (line 9) causes control to continue following the outer loop (lines 6-12) when the value of Principal exceeds 2. If the block identifiers had not been used, control would continue after the ENDDO, which is not the desired effect.

B-9 GUESS and ADMIT Statements

```
Syntax:    GUESS              [: block identifier]
             .....
           ADMIT              [: block identifier]
             .....
           ADMIT              [: block identifier]
             .....
           ENDGUESS           [: block identifier]
```

It is sometimes convenient to have a more advanced control structure than that supplied by IF-groups or loops. A GUESS-group defines a number of blocks of statements. Each block is separated from the next one by an ADMIT statement. This structure is useful because a QUIT or QUITIF statement can cause control to continue

at the next block (not following the ENDGUESS unless the QUIT or QUITIF is in the last block). Consider the following sequence of statements:

```
(1)   real Start, Ending, Increment, x
      .....
(2)   guess
(3)      read, Start
(4)      read, Ending
(5)      quitif Start >= Ending
(6)      read, Increment
(7)      quitif Increment <= 0
(8)      quitif Increment > (Ending - Start)
(9)      do x = Start, Ending, Increment
(10)        print, x, x ** 2
(11)     enddo
(12)  admit
(13)     print, "table parameters in error"
(14)  endguess
```

The example will read (lines 3, 4, 6) three values from the terminal to be used to display a table of squares. The first two describe the starting value and ending value for the table, and the last specifies the steps used to create the table. The sequence of statements will detect three errors:

- starting value greater than ending value
- increment non-positive
- increment larger than table

In each case, a QUITIF statement (lines 5, 7, 8) is used to exit from the first block in the GUESS group. When the condition associated with these statements is TRUE, control continues at the start of the next block (line 13).

The terms "guess" and "admit" are used to emphasize the structure of the program. In the example, the first block of statements represents the "guess" that the values to be read are error-free and the table will be displayed. When an error is detected, the second block must be executed ("admit" that the "guess" was not correct for the data in question).

In general, a GUESS group consists of one or more blocks, separated by ADMIT statements, and ending with an ENDGUESS statement. The execution of a QUIT statement or a QUITIF statement in which the condition is TRUE causes control to continue at the start of the next block in the GUESS group. When these statements are found in the last block in the GUESS group, control continues following the ENDGUESS statement. Similarly, after the last statement in a block has been executed, control continues following the ENDGUESS statement.

Chapter C

Remote Blocks

C-1 Introduction

In any non-trivial program it is imperative to organize the program into sections or modules, each of which performs a distinct, well-defined activity. Remote blocks may be used to effect this organization. Consider the following program:

```
(1)   integer Year, Month
(2)   real Rate, Principal
(3)   Rate = 0.015
(4)   Principal = 1.00
(5)   do Year = 1, 2
(6)       execute ComputeYearBalances
(7)   enddo
(8)
(9)   remote block ComputeYearBalances
(10)      print, 'Year = ", Year
(11)      do Month = 1, 12
(12)          Principal = Principal * (1 + Rate)
(13)          print, Month, Principal
(14)      enddo
(15)  endblock
(16)
(17)  end
```

The program is a variation of one introduced in the preceding chapter. It displays how one dollar appreciates in value over a two-year period when interest is calculated at 1.5% monthly.

The remote block "ComputeYearBalances" specifies a sequence of statements to be executed whenever an EXECUTE statement which references that block is encountered. Control automatically continues following the EXECUTE statement that invoked the remote block. Consequently, the execution of the program proceeds as if the body of the remote block was substituted for the EXECUTE statement indicating that block.

C-2 Execute Statement

Syntax: EXECUTE remoteblock

The EXECUTE statement causes a remote block to be invoked. Once the statements in the remote block have completed execution, control continues at the next statement following the EXECUTE statement.

EXECUTE statements may be placed within remote blocks as illustrated in the following example:

```
(1)  integer Year, Month
(2)  real Rate, Principal
(3)  Rate = 0.015
(4)  Principal = 1.00
(5)  do Year = 1, 2
(6)      execute ComputeYearBalances
(7)  enddo
(8)
(9)  remote block ComputeYearBalances
(10)     print, "Year = ", Year
(11)     do Month = 1, 12
(12)         execute ComputeMonthBalance
(13)     enddo
(14) endblock
(15)
(16) remote block ComputeMonthBalance
(17)     Principal = Principal * (1 + Rate)
(18)     print, Month, Principal
(19) endblock
(20)
(21) end
```

Note the EXECUTE statement (line 12) within the remote block "ComputeYearBalances".

C-3 Remote Blocks

```
Syntax:   REMOTE BLOCK name
              .... (body of remote block)
          ENDBLOCK
```

A remote block defines a group of statements to be executed when an EXECUTE statement, which references the name of the remote block, is executed. When a remote block is invoked, the system "remembers" the EXECUTE statement which caused the block to be activated so control can continue following that EXECUTE

statement when the execution of a remote block is complete.

When the remote block begins execution, the first statement to be executed is the one following the REMOTE BLOCK statement. Control continues in the normal way within the remote block until the ENDBLOCK statement is encountered. At this point, the execution of the remote block is complete and control continues following the EXECUTE statement which invoked the remote block in question.

Additional Rules About Remote Blocks:

(1) The names of remote blocks are formed according to the same rules as apply to variable names (see Variable Names). An error will be detected if a remote block name is also used as a variable name.

(2) Remote blocks must be defined in the same program unit (main program, subroutine, or function) as the EXECUTE statements which reference them.

(3) All variables used in a remote block may be used elsewhere in the program unit (main program, subroutine, or function). Similarly, all variables used elsewhere in a program unit may be used in a remote block.

(4) The definition of a remote block cannot occur as part of the body of another remote block definition, although a remote block may include EXECUTE statements.

(5) The definition of a remote block cannot occur within a loop, IF-group, or GUESS/ADMIT group. Any such groups defined in a remote block must be completely defined in the remote block.

(6) It is an error to use a GOTO statement to branch into the body of a remote block or to branch out of the body of a remote block.

(7) When a remote block is encountered as the next sequential statement during the execution of a program, then control continues at the statement following the ENDBLOCK statement for the remote block. In other words, the statements in a remote block are skipped if a remote block is encountered without execution of an EXECUTE statement.

(8) It is permissible to recursively activate remote blocks.

Notes

Chapter D

Arrays

D-1 Introduction

An array is a collection of numbers or character strings which can be referenced and manipulated either individually or jointly. The statement

```
INTEGER  Product(100)
```

may be used to define an array of 100 INTEGER values. These values could be referenced in a program as Product(1), Product(2),..., Product(100) and used anywhere in a program that a simple INTEGER variable could be used.

Consider the following program which displays, in reverse order, the sums of the positive integer values, up to 30.

```
(1)   integer SumOfInteger(30)
(2)   SumOfInteger(1) = 1
(3)   do i = 2, 30
(4)      SumOfInteger(i) = i + SumOfInteger(i-1)
(5)   enddo
(6)   do i = 30, 1, -1
(7)      print, i, SumOfInteger(i)
(8)   enddo
(9)   end
```

In line (1), any array SumOfInteger is defined to contain 30 INTEGER elements. The first of these numbers is assigned a value of 1 in line (2). The loop, lines (3) to (5), will cause the remainder of the array to be assigned values in such a way that the i-th element contains the sum 1+2+...+(i-1)+i. The loop, lines (6) to (8), will cause the array values to be displayed, starting with the 30-th and proceeding to the first.

The program illustrates the use of a subscript to reference an element in an array. This subscript is a numeric expression, enclosed in parentheses, following the array name. The particular element to be referenced is located by evaluating the subscript expression. Thus, when i has a value 17, SumOfInteger(i-1) refers to the 16-th element in the array SumOfInteger.

D-2 Defining Arrays

```
Syntax:   INTEGER array(dim,dim,...),array(dim,dim,...)
          REAL array(dim,dim,...),array(dim,dim,...)
          CHARACTER array(dim,dim,...),array(dim,dim,...)
```

Arrays are defined using one of the type statements INTEGER, REAL, or CHARACTER. The type statement specifies the type of all elements in the array. The number of elements in the array is determined by the list of dimensions (INTEGER constants) enclosed in parentheses and separated by commas. The product of these dimensions specifies the number of elements in the array.

```
CHARACTER Name(100,3),Label(5,4,2)
```

In the preceding example, Name is declared to have 300 elements and Label is declared to consist of 40 elements.

Additional Rules:

(1) Array names are specified in the same manner as other variable names (see Variable Names).

(2) A maximum of seven dimensions may be specified. Discretion should be used when specifying multiple dimensions. Since the memory in a computer is finite, there may not be sufficient space to store all the elements of a large array. In this circumstance, an error message diagnosing "memory overflow" will be issued.

(3) Arrays may be passed as parameters to subroutines or functions. Additional rules (see Array Parameters) apply for the declaration of these arrays in the invoked subroutine or function.

(4) As with simple variables, all elements in an array are initially said to have "undefined values". An error will be detected should an array element be referenced in an expression before it is assigned a value.

D-3 Subscripts

As has been introduced, subscripts are used to reference individual elements in arrays. In general, the same number of subscript expressions must be specified as there are dimensions in the array. The subscript expressions are enclosed in parentheses and separated by commas, following the array name. Consider the following array specifications:

```
real  Sales(10,20,5)
character  Name(100)
```

The following example illustrates several valid subscripted variables for the preceding array definitions:

```
Sales(region-100, salesman, product)
Name(43)
Name(salesman)
```

As the program executes, the subscript expressions are evaluated and the resultant values are used in order to calculate the actual element to be referenced. It is illegal for subscripts to be non-positive or to exceed the associated dimension specification.

A subscripted variable may be used wherever a simple variable is allowed. For example, a subscripted variable may be used in expressions, assigned values, or displayed with a PRINT statement.

When a subscript expression produces a REAL result, the value is truncated to become an integral value. Thus, ARR(7.9999) refers to the 7-th element of the array ARR.

D-4 Substring with Character Arrays

A substring of a character array element is referenced by specifying the substring information following the array subscript.

```
character name (20,3)
....
name(7,2)(4:6) = 'xxx'
```

In the example, 'xxx' is assigned to positions 4 to 6 in the element name(7,2).

D-5 Storage Order of Arrays

In many circumstances it is necessary to know the order of storage of the elements of an array. For example, a PRINT statement will display the elements of an array in the order in which they are stored.

Elements are stored by varying the first subscripts before the subscripts which follow them. For an array A(3,2), the elements would be stored as follows:

A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2)

Similarly, an array A(2,3,2) would be stored in the following order:

A(1,1,1), A(2,1,1), A(1,2,1),
A(2,2,1), A(1,3,1), A(2,3,1),
A(1,1,2), A(2,1,2), A(1,2,2),
A(2,2,2), A(1,3,2), A(2,3,2)

Notes

Chapter E

Functions and Subroutines

E-1 Program Units

A FORTRAN program consists of a number of program units: main program, functions, and subroutines. Each program must have exactly one main program and may have any number of functions or subroutines.

When a program begins execution, the main program receives control. A subroutine is invoked using a CALL statement. This causes that subroutine to receive control. When a RETURN or END statement is encountered in the subroutine, control returns to the invoking program unit, immediately following the CALL statement used to invoke the subroutine.

A function is invoked when the function name is encountered during the evaluation of an expression. Control is then passed to the function. When an END or RETURN statement is encountered, the return value computed by the function is then used to continue evaluation of the original expression. Thus, functions and subroutines are similar, with the major differences being the manner in which they are invoked and the capability to return a value from a function.

Each program unit is considered to be self-contained in that the definitions of FORTRAN entities are only locally known in each unit. Thus, a variable X, declared in one unit, is not able to be directly referenced in another program unit. If a second program unit declared a variable X, this second variable is considered to be a distinct variable from the first. Consequently, it may be given a different type and usage in the second program unit. Other FORTRAN entities having this local scope include names of remote blocks and statement numbers. Each variable declared in a program unit has an undefined value when the program unit is invoked.

Values are communicated between program units by the use of parameters. As is described in following sections, parameters may be used to pass values to an invoked program unit and, in the right circumstances, to return values to the invoking program unit.

As part of the FORTRAN system, there are a number of supplied or intrinsic functions to perform well-known calculations (e.g., to compute trigonometric SINE). These functions are invoked in the same way as functions located in the program.

E-2 Main Program

When a program is placed into execution, the main program receives control. Consequently, each FORTRAN program must contain exactly one main program.

The main program has the following format:

(a) an optional PROGRAM statement
(b) the FORTRAN statements to be executed
(c) an END statement

The program statement has the following syntax:

PROGRAM name

The name should be a meaningful name for the program and must not be identical to a name of a subroutine or function used in the program.

E-3 Parameters in General

In general, parameters are used to pass values between program units. These items are termed arguments in the invoking routine and are symbolically represented by parameters (which look like simple variables) in the invoked routine. Because of the flexibility with which parameters may be passed, several sections deal with the various aspects of parameters. These sections are found following the sections dealing with subroutines and functions, and will apply to both these kinds of program units.

E-4 Subroutines

Subroutines are invoked using a call statement:

```
CALL  name(argument, argument, ...)
```

Arguments and parameters (described subsequently) may be optionally used to communicate between the invoking program unit and the invoked subroutine.

The format of a subroutine is as follows:

(a) a SUBROUTINE statement
(b) the FORTRAN statements to be executed
(c) an END statement

A SUBROUTINE statement has the following syntactic format:

```
    SUBROUTINE  name(parameter, parameter, ...)
or SUBROUTINE  name()
or SUBROUTINE  name
```

The statement is used to specify the start of the subroutine. The name of this subroutine must be distinct from the name in the optional PROGRAM statement and from the names of other functions or subroutines used in the program. It is also an error to use the subroutine name as a variable within that subroutine.

The following program illustrates a simple use of subroutines:

```
(1)  program Illustrate
(2)    call Greatest(5,7,3)
(3)    call Greatest(6,2,9)
(4)  end
(5)
(6)  subroutine Greatest(a,b,c)
(7)    integer a,b,c,g
(8)    g = a
(9)    if g < b
(10)     g = b
(11)   endif
(12)   if g < c
(13)     g = c
(14)   endif
(15)   print, g
(16) end
```

The subroutine Greatest is passed three INTEGER arguments. The main program, Illustrate, causes the subroutine to be invoked twice and the following to be displayed:

```
7
9
```

Within the subroutine, three parameters (a,b,c) are used, in the same manner as simple variables, with the argument values appropriately assigned to them.

E-5 Functions

A function is used to compute a value to be returned. The function is invoked by referencing it in an expression. The returned value is subsequently used to calculate the value of the expression. Consider the following program:

```
(1)   program Illustrate
(2)       integer Greatest
(3)       print, Greatest(5,7,3)
(4)       print, Greatest(6,2,9)
(5)   end
(6)
(7)   integer function Greatest(a,b,c)
(8)       integer a,b,c,g
(9)       g = a
(10)      if g < b
(11)          g = b
(12)      endif
(13)      if g < c
(14)          g = c
(15)      endif
(16)      Greatest = g
(17)  end
```

The function Greatest returns an INTEGER value equal to the largest of the three arguments passed to it. This is accomplished by assigning to the function name the greatest of the three parameters.

The general format of a function is as follows:

(a) a FUNCTION statement
(b) the FORTRAN statements to be executed
(c) an END statement

The FUNCTION statement has the following syntactic definition:

```
    type FUNCTION  name
or type FUNCTION  name()
or type FUNCTION  name(parameter, parameter, ...)
or FUNCTION  name
or FUNCTION  name()
or FUNCTION  name(parameter, parameter, ...)
```

where "type" is one of INTEGER, REAL, or CHARACTER and, if specified, determines the type of value to be returned.

When the type of function is not specified in the FUNCTION statement, the type of value to be returned is determined by the type of the function name, when used as a variable in the program unit. In this case, the type is determined either by explicitly declaring the type using an INTEGER, REAL or CHARACTER statement, or by the default rules according to the first letter in the function name.

Each time a function is invoked, the return value must be assigned to the function name. Otherwise, an undefined value will be returned causing an error to be detected.

E-6 Argument: simple variable

When a simple (unsubscripted) variable is passed as an argument, the corresponding parameter in the invoked program unit must be declared as a simple variable of the same type. If the original argument has been assigned a value, then the corresponding parameter is given that value; otherwise, the parameter has an "undefined value".

The corresponding parameter may be used in the invoked program unit in the same way as a simple variable. When the execution of the program unit completes, the value of the corresponding parameter is assigned to the original simple variable passed as an argument.

```
(1)     program SimpleVariable
(2)         x = 5
(3)         call Sub(x)
(4)         print, x
(5)     end

(6)     subroutine Sub(y)
(7)         print, y
(8)         y = y * 2
(9)     end
```

The program would cause the following output to be displayed:

```
 5.0000000
 10.000000
```

E-7 Argument: expression

When an expression is passed as an argument, the expression is evaluated and the resultant value is assigned to the corresponding parameter. The parameter must be a simple variable of the same type as the expression. The parameter may be used anywhere in the invoked program unit as a simple variable.

The passing of an expression is illustrated by the following program:

```
(1)     program Expression
(2)         call sub(18)
(3)     end

(4)     subroutine Sub(y)
(5)         print, y
(6)     end
```

The following output would be displayed:

```
18.000000
```

E-8 Argument: substring of simple variable

When the argument passed to a program unit is a substring of a simple CHARACTER variable, the corresponding parameter in the invoked program unit must be declared as a CHARACTER variable. The substring argument must be defined and that value is assigned to the corresponding parameter. The parameter may be used anywhere in the invoked program unit as a simple variable. When the execution of the invoked program unit is complete, the value of the parameter is assigned to the substring argument according to the same rules as substring assignment.

The passing of a substring argument is illustrated by the following program:

```
(1)   program Substring
(2)       character s
(3)       s = 'abcdefgh'
(4)       call sub(s(4:7))
(5)       print, s
(6)   end

(7)   subroutine sub(t)
(8)       character t
(9)       print, t
(10)      t = '123456'
(11)  end
```

The following output would be displayed:

```
defg
abc1234h
```

E-9 Argument: array

When an array is passed as a parameter, it must be declared in the invoked program unit as an array of the same type (INTEGER, REAL or CHARACTER). This is illustrated in the following program:

```
(1)   real ARR(10)
(2)   do i = 1, 10
(3)       ARR(i) = i
(4)   enddo
(5)   call PrintArr(ARR)
(6)   end
(7)
(8)   subroutine PrintArr( Array )
(9)   real Array(10)
(10)  do i = 1,10
(11)      print, Array(i)
(12)  enddo
(13)  end
```

The subroutine PrintArr causes the array to be printed.

The array declared in the invoked program unit need not be dimensioned exactly as the original argument. The only restriction is that the number of elements of the corresponding parameter must not exceed the number of elements in the original array. Consider the following program:

```
(1)   real ARR(10)
(2)   do i = 1, 10
(3)       ARR(i) = i
(4)   enddo
(5)   call PrintArr(ARR)
(6)   end
(7)
(8)   subroutine PrintArr(Array)
(9)   real Array(4,2)
(10)  do j = 1,2
(11)      do i = 1,4
(12)          print,Array(i,j)
(13)      enddo
(14)  enddo
(15)  end
```

The subroutine PrintArr causes the first 8 elements of the array to be printed.

More flexibility is possible, if the array dimensions are passed as arguments. This is illustrated in the following program.

```
(1)   real ARR(9,2)
(2)   do i = 1,9
(3)      do j = 1,2
(4)         ARR(i,j) = i * 10 + j
(5)      enddo
(6)   enddo
(7)   call PrintArr(ARR,5)
(8)   end
(9)
(10)  subroutine PrintArr(Array,n)
(11)  real Array(n)
(12)  do i = 1,n
(13)     print,Array(i)
(14)  enddo
(15)  end
```

Execution of the preceding program will produce the following output:

```
11.000000
21.000000
31.000000
41.000000
51.000000
```

It should be noted that the array parameter Array is declared with a dimension specified using another parameter. In general, any dimension for an array parameter may be specified by a simple INTEGER parameter. The value of that simple parameter is used to establish the dimension of the array.

When an array parameter is declared in an invoked routine, the resulting array specifies a consecutive portion of the original argument array. When the original array involves several dimensions, it is important to consider the order in which the elements are stored in computer memory (see Array storage). Any modification of values in the array parameter will be applied to the original array in the associated position, immediately as they occur to the array parameter.

E-10 Argument: array element

When an array element is passed as an argument, it may be treated in the invoked program unit as either a simple (unsubscripted) parameter or as an array parameter. When the corresponding parameter is a simple variable, the parameter is treated identically to the way a simple variable (see Argument: simple variable) is treated.

When the corresponding parameter is an array parameter, it is treated similarly to an array argument, except that the first element in the array parameter is the element passed as an argument.

```
(1)   real ARR(9,2)
(2)   do i = 1,9
(3)       do j = 1,2
(4)           ARR(i,j) = i * 10 + j
(5)       enddo
(6)   enddo
(7)   call PrintArr(ARR(2,2), 6)
(8)   end
(9)
(10)  subroutine PrintArr( Array, k)
(11)  real Array(k)
(12)  do i = 1,k
(13)      print, Array(i)
(14)  enddo
(15)  end
```

The preceding program will cause the following to be displayed:

```
22.000000
32.000000
42.000000
52.000000
62.000000
72.000000
```

The resulting parameter Array represents consecutive positions in the array argument, starting with the element passed as the array argument.

E-11 Argument: substring of array element

When a substring of a CHARACTER array element is passed as an argument, it is treated identically to a simple variable passed as an argument (see Argument: substring of single variable).

E-12 Argument: function and subroutine names

It is possible to pass a function or subroutine name as an argument to a program unit. In this way, the invoked program unit can invoke the function or subroutine supplied to it as a parameter. This is illustrated by the following program:

```
(1)  program Tables
(2)      external Square, Cube
(3)      call PrintTable("Squares", Square)
(4)      call PrintTable("Cubes", Cube)
(5)  end
(6)
(7)  subroutine PrintTable(Title,fun)
(8)      character Title
(9)      real fun
(10)     print, Title
(11)     do x = 0,1,.1
(12)         print, x, fun(x)
(13)     enddo
(14) end
(15)
(16) real function Square(x)
(17)     real x
(18)     Square = x ** 2
(19) end
(20)
(21) real function Cube(x)
(22)   real x
(23)   Cube = x ** 3
(24) end
```

The execution of the main program Tables causes the subroutine PrintTable to be invoked with the functions Square and Cube as parameters. The external statement, line (2), is required to order that these names not be treated as simple variables in the

main program. The subroutine PrintTable prints a title, passed as the first parameter, and then 11 lines of output, for each function. In this way, PrintTable displays first a table of squares and then a table of cubes.

In a similar way, a subroutine name could be passed to another program unit. This subroutine could then be invoked using the CALL statement.

E-13 Recursion

A function or subroutine is said to be active if the program unit has been invoked and a corresponding END or RETURN statement has not yet been executed for that program unit. Recursive activation occurs when a program unit is invoked and it is already active.

Consider the following fragment of a program:

```
          ....
(1)       K = Factorial(3)
          ....
(2)       integer function Factorial(N)
(3)           if N > 1
(4)               Factorial = N * Factorial(N-1)
(5)           else
(6)               Factorial = 1
(7)           endif
(8)       end
          ....
```

The function Factorial computes the factorial (the factorial of a positive integer is the product of all positive integers less than or equal to the number in question) of the number passed to it. This is accomplished by returning 1 when the argument value does not exceed 1; otherwise, Factorial is recursively invoked to calculate the factorial of the number one less and that result is multiplied by the number. The following sequence of statements would be executed:

```
(1)  invoke Factorial(3)
   (2)  Factorial(3)
   (4)  invoke Factorial(2)
      (2)  invoke Factorial(1)
      (4)  invoke Factorial(1)
         (2)  Factorial(1)
         (6)  Factorial = 1
         (8)  return 1
      (4)  Factorial = 2 * 1
      (8)  return 2
   (4)  Factorial = 3 * 2
   (8)  return 6
(1)  K = 6
```

In the preceding diagram, indentation has been used to illustrate the levels of function activation.

One consequence of recursion is that each time a function occurs to the right of the assignment operation (=), the function will be used in an expression in the same way a variable is used.

```
(1)   integer function Factorial(N)
(2)       Factorial = 1
(3)       do i = 1,N
(4)           Factorial = Factorial * N
(5)       enddo
(6)   end
```

The preceding example would be diagnosed as containing an error in line (4) because there is an incorrect number of parameters in the invocation of Factorial.

In a similar way, subroutines may be invoked recursively. It is permissible to CALL a subroutine which is already activated.

The variables declared within a subroutine or function are local to the activation of that program unit. Thus, when a program unit is recursively invoked, all local variables are initially undefined. An assignment to one of these variables will have no effect on the local variables of any other activation of a program unit.

E-14 EXTERNAL Statement

Syntax: EXTERNAL name, name, ..., name

The EXTERNAL statement is used to declare the names in the list as external names (functions or subroutines). It may be used to declare any external name in this way.

An EXTERNAL statement must be used to define names as external when the names would otherwise be considered as local variables. Two situations in which this may occur are:

(1) When passing a function or subroutine name as an argument to another program unit.

(2) When invoking a function which does not require a parameter list.

E-15 Intrinsic Functions

The intrinsic functions are included to perform several common computations. If a function or subroutine is defined in the program by the same name, then the defined program unit is referenced, not the intrinsic function.

Intrinsic functions are not required to be given explicit types in the program units in which they are referenced. Thus, the CHAR intrinsic function may be used without explicitly giving it a type in a CHARACTER statement.

A brief description of the intrinsic functions is given below:

ABS(X)

This function returns, as a REAL value, the absolute value of the REAL argument X.

ACOS(X)

This function returns, as a REAL value, the trigonometric ARCCOSINE (in radians) of the REAL argument X, where $-1 < X < 1$ and $0 <$ ACOS(X) $<$ PI.

AINT(X)

This function returns a REAL value representing the integral portion of the REAL argument X. This value is obtained by truncating the fraction part of X.

ALOG(X)

This function returns, as a REAL value, the natural logarithm of the REAL argument X. The argument must be greater than zero.

ALOG10(X)

This function returns, as a REAL value, the logarithm (base 10) of the REAL argument X. The argument must be greater than zero.

AMAX0(I1, I2, ..., IN)

This function returns, as a REAL value, the maximum of the INTEGER arguments I1, I2, ..., IN.

AMAX1(X1, X2, ..., XN)

This function returns, as a REAL value, the maximum of the REAL arguments X1, X2, ..., XN.

AMIN0(I1, I2, ..., IN)

This function returns, as a REAL value, the minimum of the INTEGER arguments I1, I2, ..., IN.

AMIN1(X1, X2, ..., XN)

This function returns, as a REAL value, the minimum of the REAL arguments X1, X2, ..., XN.

AMOD(X1, X2)

This function returns, as a REAL value, the REAL modulus of X1, modulo X2, computed as X1 − INT(X1/X2) * X2, X2 cannot be zero.

ANINT(X)

This function returns a REAL value representing the nearest integer value to the REAL argument X.

ASIN(X)

This function returns, as a REAL value, the trigonometric ARCSINE (in radians) of the REAL argument X, where $-1 < X < 1$ and $-(PI/2) < ASIN(X) < (PI/2)$.

ATAN(X)

This function returns, as a REAL value, the ARCTANGENT (in radians) of the REAL argument X, where $-(PI/2) < ATAN(X) < (PI/2)$.

CHAR(I)

This function returns, as a single character, the I-th character in the collating sequence of the computer. I is an INTEGER argument.

CNVC2I(C)

This function returns, as an INTEGER value, the number represented in the CHARACTER argument C.

CNVC2R(C)

This function returns, as a REAL value, the number represented in the CHARACTER argument C.

CNVH2I(C)

This function returns, as an INTEGER value, the number represented in the CHARACTER argument C, interpretted as containing hexadecimal digits.

CNVI2C(I)

This function returns, as a CHARACTER string, a representation of the INTEGER argument I. The format of the character string is identical to that produced by a PRINT statement without a FORMAT specification.

CNVI2H(I)

This function returns, as a CHARACTER string, a representation (in hexadecimal format) of the INTEGER argument I.

CNVR2C(X)

This function returns, as a CHARACTER string, a representation of the REAL argument X. The format of the character string is identical to that produced by a PRINT statement without a FORMAT specification.

COS(X)

This function returns, as a REAL value, the trigonometric COSINE of the REAL argument X, where X is expressed in radians.

COSH(X)

This function returns, as a REAL value, the hyperbolic COSINE of the REAL argument X.

DATE()

This function returns a character string which represents the current date. The format of the character string is dependent upon the system (see System Dependencies).

DIM(X1, X2)

This function returns, as a REAL value, the positive difference between the REAL arguments X1 and X2: if X1 > X2, then (X1 − X2) is returned; otherwise, 0 is returned.

EXP(X)

This function returns, as a REAL value, the mathematical constant e raised to the power X. X is a REAL argument.

FLOAT(I)

This function returns, as a REAL value, the INTEGER argument I.

IABS(I)

This function returns, as an INTEGER value, the absolute value of the INTEGER argument I.

ICHAR(C)

This function returns, as an INTEGER value, the position in the collating sequence of the CHARACTER argument.

IDIM(I1, I2)

This function returns, as an INTEGER value, the position difference between the INTEGER arguments I1 and I2: if I1 > I2, then (I1 − I2) is returned; otherwise, 0 is returned.

IFIX(X)

This function returns, as a INTEGER value, the REAL argument X.

INDEX(C1, C2)

This function returns an INTEGER value representing the position that the CHARACTER argument C2 first occurs in the CHARACTER argument C1. If the contents of C2 are not found anywhere in C1, zero is returned.

INT(X)

This function returns, as an INTEGER value, the REAL argument X.

ISIGN(I1, I2)

This function returns, as an INTEGER value, the absolute value of the second INTEGER argument with the sign of the first INTEGER argument.

LEN(C)

This function returns, as an INTEGER value, the length of the character string C.

LGE(C1, C2)

This function returns, as an INTEGER value, 1 when C1 is greater than or equal to C2; otherwise, 0 is returned. C1 and C2 are CHARACTER arguments.

LGT(C1, C2)

This function returns, as an INTEGER value, 1 when C1 is greater than C2; otherwise, 0 is returned. C1 and C2 are CHARACTER arguments.

LLE(C1, C2)

This function returns, as an INTEGER value, 1 when C1 is less than or equal to C2; otherwise, 0 is returned. C1 and C2 are CHARACTER arguments.

LLT(C1, C2)

This function returns, as an INTEGER value, 1 when C1 is less than C2; otherwise, 0 is returned. C1 and C2 are CHARACTER arguments.

MAX0(I1, I2, ..., IN)

This function returns, as a INTEGER value, the maximum of the INTEGER arguments I1, I2, ..., IN.

MAX1(X1, X2, ..., XN)

This function returns, as an INTEGER value, the maximum of the REAL arguments X1, X2, ..., XN.

MIN0(I1, I2, ..., IN)

This function returns, as an INTEGER value, the minimum of the INTEGER arguments I1, I2, ..., IN.

MIN1(X1, X2, ..., XN)

This function returns, as an INTEGER value, the minimum of the REAL values X1, X2, ..., XN.

MOD(I1, I2)

This function returns, as an INTEGER value, the modulus of I1, modulo I2, computed as I1−(I1/I2) * I2. I2 cannot be zero. I1 and I2 are INTEGER arguments.

NINT(X)

This function returns an INTEGER value representing the nearest integer value to the REAL argument X.

PEEK1(I)

This function returns, as an INTEGER value, the contents of one byte of memory located by the value of the INTEGER argument I.

PEEK2(I)

This function returns, as an INTEGER value, the contents of two consecutive bytes of memory located by the value of the INTEGER argument I.

POKE1(I1, I2)

This function causes the value of the INTEGER argument I2 to be stored at the one-byte memory location located by the value of the INTEGER argument I1. The function returns an INTEGER value representing the previous contents of that memory location.

POKE2(I1, I2)

This function causes the value of the INTEGER argument I2 to be stored at the two-byte memory location located by the value of the INTEGER argument I1. The function returns an INTEGER value representing the previous contents of that memory location.

RND(X)

This function returns, as a REAL value, a random value between 0 and 1. This value is predictably calculated from the value of the REAL argument X, unless this value is 0. When 0 is passed, an unpredictable random number is returned.

RPT(C, I)

This function returns, as a CHARACTER value, the string obtained by repeating the first CHARACTER argument the number of times indicated by the second INTEGER argument. The second argument must be non-negative; when it is zero, a null string is returned.

SIGN(X1, X2)

This function returns, as a REAL value, the value of the second REAL argument with the sign of the first REAL argument.

SIN(X)

This function returns, as a REAL value, the trigonometric SINE of the REAL argument X, where X is in radians.

SINH(X)

This function returns, as a REAL value, the hyperbolic SINE of the REAL argument X.

SLEEP(X)

This function causes the program to pause for a period of time determined by the REAL argument X. When the program resumes execution, the function returns the current time of day in the same manner as the TOD intrinsic function. The values of the return value and of the argument depends upon the system (see System Dependencies).

SQRT(X)

This function returns, as a REAL value, the square root of the REAL argument X. The argument must be greater than or equal to zero.

SUBSTR(C, I1, I2)

This function returns, as a CHARACTER value, the string obtained from the CHARACTER string C, starting at the position indicated by the second INTEGER argument, for a length indicated by the third INTEGER argument. This function differs from the substring operation in the following ways:

- the argument C may be an expression
- because the argument I2 is a length, a null string may result
- when argument I2 is negative, it is treated as having a value zero

- when argument I1 is non-positive, it is treated as having a value one
- when argument I1 is larger than the length of the string C, it is treated as having that value

SYS(I, A, A, ..., A)

This function is used to invoke a user-written routine that is present outside of the FORTRAN environment (see System Dependencies).

TAN(X)

This function returns, as a REAL value, the trigonometric TANGENT of the REAL argument X, where X is expressed in radians.

TANH(X)

This function returns, as a REAL value, the hyperbolic TANGENT of the REAL argument X.

TIME()

This function returns a character string which represents the current time. The format of the character string is dependent upon the system (see System Dependencies).

TOD()

This function returns, as a REAL number, a value representing the current time. The value which is returned is dependent upon the system (see System Dependencies).

VARPTR(V)

This function returns, as an INTEGER value, the address of the argument V. The argument must be an item which is capable of being assigned a value.

WARNING: The implementation of character strings permits the addresses of these elements to change during the execution of a program. Consequently, the use of this function with character data is not advised.

Chapter F

Input/Output

F-1 Introduction to Files

Data is transmitted to and from FORTRAN programs using files. A file is a collection of records, where each record is a contiguous stream of characters. One advantage of using files is that information can be processed one record at a time. Another advantage is that several programs can access the same information when that information is stored in a file. These programs could be written in different languages. For these and other reasons, it is very common to store information in files using diskettes or other media.

In FORTRAN, an OPEN statement is used to connect a file with a unit number. This unit number is referenced in data-transmission statements (READ, WRITE and PRINT) in order to transmit records of information to and from programs. When the data transmission is complete, a CLOSE statement may be used to disconnect that file from the unit. Strictly speaking, it is never necessary to open or close a unit, since defaults apply for all options in these statements. In order to access particular files, it is necessary to open a unit referencing that file.

Data occurs in FORTRAN programs in variables and arrays. This data is transformed into records according to specifications in the program (formatted

input/output) using FORMAT or by using default specifications (list-directed input/output). The following chapter describes FORMAT specifications.

The records in a file may be accessed in the order they were originally written (sequential input/output) or may be accessed in any order (direct input/output) by specifying the number of the record to be accessed. This topic is discussed in more detail in following sections.

Because there is no guarantee that an input/output operation will always succeed, there are various facilities available for error detection. These are described subsequently.

F-2 Sequential Input/Output

The records in each file are stored consecutively in the order in which they are originally written. When records are read (READ statement) sequentially, they are encountered in this order, starting with the first one written. The REC option is never used when reading or writing records sequentially.

F-3 Direct Input/Output

Direct input or output may be used on an existing file to read or write records in any order. The REC option is used to specify the number of the record to be read. By convention, this numbering starts with zero (0) for the first record originally written on the file. Each additional record originally written has a number one larger than that for the preceding record written for that file.

F-4 Error Handling

FORTRAN provides error-handling facilities in a number of ways. It is often necessary to use these capabilities in production systems since there is always the possibility of a hardware error. It is also convenient to use the error-detection mechanisms in order to detect the end of files. The microFORTRAN system will display error messages when an input/output error is detected, and no error facility is specified to handle the error.

In order to detect the kind of input/output error which may occur, the IOSTAT option can be used in the FORTRAN input/output statements. This option specifies an INTEGER variable that is assigned a value which indicates the status of the associated input/output operation. A value of zero indicates that the operation was

successful. Other values (see Input/Output Error Codes) are used to indicate the various error conditions which may occur.

When reading a file, an attempt to read beyond the end of the file may be handled by the END option. This option specifies the number of a statement to which control is transferred when attempting to read beyond the end of the file. It should be noted that an attempt to read beyond the end of a file is not treated as an input/output error.

An input/output error can be detected by using the ERR option. Like the END option, a transfer to the statement, with the statement number indicated, will occur if an input/output error occurs.

F-5 OPEN Statement

Syntax: OPEN (option, option, ..., option)

---

where option is one of:

(a)	unit	=	int
(b)	file	=	char
(c)	access	=	char
(d)	recl	=	int
(e)	iostat	=	variable

and "int" is an INTEGER expression and "char" is a character expression.

The OPEN statement is used to connect a file to a unit number. The UNIT option must be specified. If it is the first option, it is necessary only to supply the INTEGER expression giving the number of the unit.

The FILE option, if specified, indicates the name of the file (see File Names) to be connected. If this option is not specified, the default file name "FTNx" is used, where "x" is the character representation of the unit number. Thus, a reference to unit 8, without a FILE option, refers to a file named "FTN8". An exception to this rule is the default file for units 5 and 6. These units are connected, by default, to the terminal.

The ACCESS option is used to specify whether the file will be accessed sequentially or directly. If omitted, it is assumed that the file will be accessed sequentially. To specify sequential or direct access, the associated character

expression should be evaluated as either "SEQUENTIAL" or "DIRECT", respectively.

The RECL option is used to specify the maximum size of a record which can occur in the file. If omitted, the maximum size is assumed to be 80 characters.

The IOSTAT option is used to specify an INTEGER variable to which will be assigned a value indicating the status of the OPEN operation.

F-6 CLOSE Statement

Syntax: CLOSE (unit=int)

---

where "int" is an INTEGER expression.

The CLOSE statement disconnects a unit number from a specific file. The required UNIT option indicates which unit is to be disconnected. It is necessary only to supply the expression specifying the unit to be disconnected.

F-7 READ Statement

Syntax:	READ (option, ..., option) item, item, ... item
	READ fmt-spec, item, item, ... item
or	READ, item, item, ... item

---

where "option" is one of

(a) unit = int
(b) fmt = fmt-spec
(c) rec = int
(d) err = stmnt
(e) end = stmnt
(f) iostat = variable

and where

(a) "int" is an INTEGER expression
(b) "stmnt" is a statement number
(c) "variable" is a variable or array element
(d) "fmt-spec" is either the statement number of a FORMAT statement, a CHARACTER expression or '*'

and where "item" is one of:

(a) a variable, array element, or array name
(b) a list (item, ..., item, var = exp, exp), or a list (item, ..., item, var = exp, exp, exp) in which "var" is a variable and "exp" is an expression

The READ statement is used to transmit data from a file to a FORTRAN program. When the option list is not present, unit 5 (by default, the terminal) is used for input and a line entered is treated as a record. Otherwise, the option list is used to determine the unit to be referenced. The absence of an option list also causes list-directed input to be used.

When an option list is present, the options have the following effect. If the UNIT option is present, the associated INTEGER expression is used to determine the unit number. When the FMT option is not present or specified as "*", list-directed input is used. When the FMT option is present and not specified as "*", the input is converted from its representation on a record and assigned to data items according to the FORMAT specification. The FORMAT specifications are given either in the associated CHARACTER expression or by the FORMAT statement with the statement number given in the FMT option. When the REC option is omitted, the next record to be read is the one following the one accessed in the last input/output operation for that unit. When the REC option is given, the associated INTEGER expression is evaluated and the resultant value is used as the number of the next record to be read. The END, ERR and IOSTAT options are used to handle errors as described previously in this chapter.

When the UNIT option is specified first, only the unit number need be specified. When there are no items in this list, a record is read from the input file although no transmission to variables occurs. When the UNIT option is first and the FMT option is second, only the unit number and the FORMAT specification need be specified

```
read(unit=7,fmt=900)a,b,c
read(7,fmt=900)a,b,c,
read(7,900)a,b,c
```

The preceding READ statements are equivalent.

Following the optional options list is the list of items to which data is to be transmitted. Data items to which data is transmitted are allowed to be any of the following:

(a) simple variable: a data value is assigned to the variable

(b) substring operation with either a simple variable or an array element: the indicated substring is assigned a character value

(c) array: assignment of values occurs as if the array elements were listed, separated by commas, in the order in which they occur in the array (see Array Storage)

Thus,

```
integer a(3,2)
read,a
```

is equivalent to

```
integer a(3,2)
read,a(1,1),a(2,1),a(3,1),a(1,2),a(2,2),a(3,2)
```

When a parenthesized list of the forms

```
(item, ..., item, var = exp, exp), or
(item, ..., item, var = exp, exp, exp)
```

is present, the list of items is processed a number of times, with the variable "var" assigned values according to the manner described with DO loops. This is called an implied DO.

```
read, (a(i), i= 4,7)
read, a(4), a(5), a(6), a(7)
```

The two preceding statements are equivalent. An array A with two dimensions N and M may be read, by rows, with the following statement:

```
read, ((A(i,j),j=1,M),i=1,N)
```

List-directed Input:

List-directed input is used when the FMT option is not specified, or when it is given as '*'. In this case, each record is treated as a number of zones, delimited by commas and by the end of the record. As each item is processed in the input list, the next zone is selected and its contents are used to determine the value to be assigned. When another input item is to be processed, and all zones from a record have been processed, the next record in the file is automatically read and used for input.

When a character item is to be assigned a value, the input zone is assigned to the character item.

When a numeric item is to be assigned a value, the numeric value is interpreted in the same way as if it were used in an assignment statement in the program.

Format-directed Input:

Format-directed input is used when the FMT option is specified and is not given as '*'. In this case, the contents of the FORMAT string are used to direct the conversion of values from the record to the input data items. These FORMAT items are described in the following chapter.

F-8 PRINT and WRITE Statements

Syntax: WRITE (option,...,option) item, item, ..., item
PRINT fmt-spec, item, item, ..., item
PRINT, item, item, ..., item

---

where "option" is one of

(a) unit = int
(b) fmt = fmt-spec
(c) rec = int
(d) err = stmnt
(e) end = stmnt
(f) iostat = variable

and where

(a) "int" is an INTEGER expression
(b) "stmnt" is a statement number
(c) "variable" is a variable or array element
(d) "fmt-spec" is either the statement number of a FORMAT statement, a CHARACTER expression or '*'

and where "item" is one of

(a) an expression
(b) a list (item, ..., item, var = exp, exp), or
a list (item, ..., item, var = exp, exp, exp)
in which "var" is a variable and "exp" is an expression

The PRINT statement is equivalent to a WRITE statement in which the options list specifies unit 6 and an optional format specification. Consequently, only the WRITE statement is described.

At least one option must be present in the options list for the WRITE statement. If the UNIT option is not specified, then unit 6 (the terminal) is used; otherwise, the associated INTEGER expression is used to determine the unit number. When the FMT option is not present or is given as '*', list-directed output is used. When the option is present and not specified as '*', the data values are converted from their internal representation to the representation on a record according to the FORMAT

specification. The FORMAT specifications are given either in the associated CHARACTER expression or by the FORMAT statement with the statement number given in the FMT option. When the REC option is omitted, the next record to be written is the one following the one accessed in the last input/output operation for that unit. When the REC option is given, the associated INTEGER expression is evaluated and the resultant value is used as the number of the next record to be written. The END, ERR, and IOSTAT options are used to handle errors as described previously in this chapter.

When the UNIT option is specified first, only the unit number need be specified. When the UNIT option is first and the FMT option is second, only the unit number and the FORMAT specification need be specified.

```
write(unit=9,fmt=1037)a,b
write(9,fmt=1037)a,b
write(9,1037)a,b
```

The preceding WRITE statements are equivalent.

Following the optional options list is the list of items to be transmitted. These items are either expressions (the value is transmitted) or an array (the values of the array elements, in order, are transmitted).

Thus,

```
integer a(3,2)
write(6) a
```

is equivalent to

```
integer a(3,2)
write(6) a(1,1),a(2,1),a(3,1),a(1,2),a(2,2),a(3,2)
```

When a parenthesized list of the forms

```
(item, ..., item, var = exp, exp), or
(item, ..., item, var = exp, exp, exp)
```

is present, the list of items is processed a number of times, with the variable "var" assigned values according to the manner described with DO loops. This is called an implied DO.

```
write(6) (a(i),i=4,7)
write(6) a(4),a(5),a(6),a(7)
```

The two preceding statements are equivalent. When no output items are specified, a blank record is written.

List-directed Output:

Each record produced by list-directed output has a space character automatically inserted in the first position. In this way, devices which support carriage control are automatically supplied with an appropriate default value.

With list-directed output, the format of items on records is determined by the type of value to be transmitted:

(a) CHARACTER data: the data is directly transferred to the record.

(b) INTEGER data: the number is converted to character format and transferred to the record right-justified in a zone 6 characters wide.

(c) REAL data: if the absolute value of the number is between 99999999 and .00000001 then the number is converted to a fractional number with 8 digits and transferred to a zone 10 characters wide. Otherwise, the number is displayed in scientific notation with 8 significant digits in a zone 15 characters wide.

When the space remaining on a record is too small to contain the zone for an item, the current record is transmitted to the file and a new record is constructed. Once all items have been transmitted to a record, the record is written to the file. When there are no items in the output list, a blank record is written to the file.

Format-directed Output:

Format-directed output proceeds as described with list-directed output, except that the conversion of items to the record is according to the specifications in the FORMAT given. In this way, the format of the records is controlled by the program, rather than using the default formatting of list-directed output. When a record is transmitted to a device with carriage control, the first character in the record is used as the control character.

F-9 Carriage Control

The first character in an output record to be displayed on the terminal screen is used for positioning purposes. It is not visible in the line displayed. The following characters have the indicated effect:

"1" This causes the screen to be cleared and the associated line to be displayed at the top of the screen.

" " This causes the associated line to be displayed at the position indicated by the terminal cursor.

"0" This causes a blank line to be displayed at the position indicated by the terminal cursor, followed by the line to be displayed.

"+" This causes the current record to be displayed one line upwards on the screen from the position indicated by the cursor.

"-" This causes two blank lines to be displayed before the display of the record.

When a line is written, the cursor is placed at the line following the displayed line. When the line is written on the bottom row of the screen, the screen will "scroll" upwards one line leaving the cursor positioned on the first position of the bottom row. This bottom row will be blank.

When the PRINT or WRITE statement is executed with format-directed output, the record contents (and consequently the first character in the record) are completely determined by the program. When list-directed output is used, a space character is appended as the first character in a record.

F-10 REWIND statement

Syntax: REWIND (unit = int)

---

where "int" is an INTEGER expression indicating the unit to be accessed.

This statement causes the positioning of a file to be as it was when the file was first opened. Thus, the next record to be read sequentially from the file will be the first record and the next record written will be the first record.

Chapter G

Format

G-1 Introduction

Format specifications are used to control the transmission of data in READ, WRITE and PRINT statements. The FMT option specifies either a CHARACTER expression or the statement number of a FORMAT statement in which the format specification is given. For example,

```
    I=5
    PRINT 20,I,I*I
20 FORMAT(I4,I5)
    END
```

would cause the following line to be transmitted

```
bbb5bbb25
```

where a "b" indicates the space character. The format directives I4 and I5 specify that an INTEGER value is to be displayed in a zone four and five characters wide, respectively. The program could have been written equivalently as:

```
I = 5
PRINT "I4,I5",I,I*I
END
```

This program differs from the preceding example in that the format specification is given by a CHARACTER expression.

In general, the format specification is processed an item at a time, from left to right. When a data-transmission item (A,I,F,E) is encountered, the next item in the input/output list causes data to be transmitted.

(a) In a READ statement, the data-transmission item causes the information in the zone of the record specified by the data-transmission item to be transmitted to the input item.

(b) In a WRITE or PRINT statement, the value of the output item is transmitted to a zone of the record in a format specified by the data-transmission item.

When a data-transmission item is encountered and no more input/output items are present in the input/output statement, the processing of the format specifications is complete. Similarly, if the end of the format specification is encountered and no more input/output items are present, the processing of format specifications is complete. At the completion of processing of the format specification, the current record is transmitted to the file when a WRITE or PRINT statement is being executed.

When the end of a format specification is encountered and more input/ output items remain to be processed, the next record is obtained:

(a) the next record is read in a READ statement
(b) the current record is written in a PRINT or WRITE statement

The next record is then processed, starting at the beginning of the record. The format specification is then reused according to the rules described in a formatting section (see FORMAT, Reuse).

The next record in a file may also be read or written if there is insufficient space in the current record for the zone associated with a format item. In this case, the next record is then processed, starting at the beginning of it and the processing of the format list continues.

It is an error for a format item to specify a zone larger than the length of a record. On input, the input record read must be at least as large as the zone for the format item; on output, the maximum record size for the file must be as large as the zone.

G-2 FORMAT Statement

Syntax: number FORMAT(format-specification)

Each FORMAT statement contains a format specification enclosed in parentheses. The statement must have a statement number before the FORMAT keyword. The format specification is detailed in the following sections.

G-3 Format Specifications Generally

In its simplest form, a format specification is a list of format items separated by commas:

```
10    format(4x, 3i2, f7.3)
20    format(' PAGE', I3)
```

The items cause one of the following to occur

(a) data to be transmitted to or from a record
(b) data to be inserted or skipped on a record
(c) a record to be written or read

Some items may be preceded by a repetition factor, in which case the format specification is treated as if the item occurred as many times as given by the repetition factor. Thus, 3i2 is equivalent to writing i2,i2,i2.

An item can also be a parenthesized list of items, optionally preceded by a repetition factor. The following FORMAT statements are equivalent (except for reuse of format):

```
10    format(3x,2(I3,4X))
10    format(3x,I3,4X,I3,4X)
```

Thus, one use of a parenthesized list is to compactly specify repeated format. It is possible to imbed a parenthesized list as an item in a parenthesized list.

Reuse of format

When the end of a format specification is encountered and more input/output items exist in the READ, WRITE or PRINT statement being executed, the format specification is reused. When no parenthesized lists exist within the format specification, the entire format specification is reused. Otherwise, the following rules are used to locate the point in the format specification where it is re-used.

(1) The last closing parenthesis in the format specification is located (this does not include the enclosing parentheses in a FORMAT statement).

(2) The matching open parenthesis is then located.

(3) Processing of the format specification continues with the parenthesized list located. If a repetition factor exists for that list, it is also processed.

Consider the following format specification:

```
i8,2(5X,I4),3(5X,4(I2,2X)),i2
```

The format specification is reused at the character "3".

G-4 Data Transmission Items

The following items are used to cause data to be transmitted either from a record (READ) or to a record (WRITE and PRINT).

(i) A or a : transmit CHARACTER data

(ii) I or i : transmit INTEGER data

(iii) F or f : transmit REAL data (fixed format)

(iv) E or e : transmit REAL data (scientific format)

These are described in the following subsections.

A-Format

Syntax: rA
or A
or rAn
or An

---

where "r" (repetition factor) and "n" (length) are integer constants.

A repetition factor of one is used when the repetition factor is not specified. A length of one is used when "n" is not specified, a READ statement is being processed, and the CHARACTER element is undefined; otherwise the length to be transmitted is the length of the CHARACTER item.

When a PRINT or WRITE statement is being executed, the A format item causes the contents of the associated CHARACTER value to be transmitted directly to the record. When the length is less than the length of the CHARACTER value, it is truncated to the proper size. Space characters are concatenated to the value when the size of the value is smaller than the indicated length.

When a READ statement is executed, the associated input item is assigned a CHARACTER value obtained by extracting the indicated number of characters from the record.

I-Format

Syntax: rIn
or In

---

where "r" (repetition factor) and "n" (length) are integer constants.

A repetition factor of one is used when the repetition factor is not specified. The length must always be specified.

When a WRITE or PRINT statement is being executed, the value of the associated INTEGER input/output item is converted to a character string of size "n". This string is transmitted to the record. When the converted character string exceeds the length "n", "n" asterisks (*) are transmitted.

When a READ statement is being executed, "n" characters on the input record are treated as an INTEGER constant whose value is assigned to the associated input item. A conversion error will result when these "n" characters are not a valid INTEGER constant. Any space characters in the zone are treated as if these characters were '0' digits.

F-Format

Syntax: rFw.d
or Fw.d

---

where "r" (repetition factor), "w" (width) and "d" (decimal places) are integer constants.

A repetition factor of one is used when the repetition is not specified. The width and number of decimal places must always be specified. The width must exceed the number of decimal places on output and must not be less than the number of decimal places on input.

On input, the width establishes a zone in the record. When the zone consists only of space characters, the value of the number to be transmitted is zero. Otherwise, the leading space characters in the zone are ignored. In the following description, it will be assumed that there are no leading space characters in the zone. Once a non-space character has been encountered, all successive space characters are treated as zero digits.

The zone may be considered to be given in two parts: a mantissa (real value) followed by an exponent. The optional exponent part begins with one of the characters e, E, + or -. When none of these are present, the zone consists only of a mantissa. The mantissa portion consists of an optional sign followed by a sequence of digits in which a single decimal point may be embedded. When the decimal point is present, the fractional part of the number is established by the decimal point. When the decimal point is not present, the right-most "d" digits in the mantissa position are considered fractional. Leading zeros are added to the left when less than "d" digits are present in the mantissa.

The exponent part consists of an optional "e" or "E" followed by an optionally signed INTEGER constant. When the exponent portion is present, the value to be transmitted is the value of the mantissa, multiplied by ten raised to the power of the INTEGER constant.

The following examples, for F8.4 format, illustrate how various input zones are interpretted to produce values ("b" stands for a space character).

zone	*value*
bbbbbbbb	0
12345678	1234.5678
bb345678	34.5678
bb3bbb7b	30.007
b-bb5678	-.5678
bb34E+02	.34
bb34E+1b	34000000
b-3bb-b1	-.003

On output, the number is printed right-justified in the zone with a decimal point preceding the decimal places indicated. Leading space characters are inserted if the printed number is shorter than the zone. When the number is negative, it is preceded by a minus sign (-). When there is insufficient space to display the number, the zone is filled with asterisk (*) characters.

E-Format

Syntax: rEw.d
or Ew.d

---

where "r" (repetition factor), "w" (width), and "d" (number of decimal places) are integer constants.

A repetition factor of one is used when the repetition factor is not specified. The width of the field and the number of decimal places must always be given. The width must exceed the number of decimal places on output and must not be less than the number of decimal places on input.

On input, the data value to be transmitted is obtained in the manner described in F-Format.

On output, the number is printed in the zone in scientific format "w" characters long with "d" decimal places and the exponent (including the E) given in four places. If the number is negative, a minus sign (-) precedes the number. When there is insufficient space to display the number, the zone is filled with asterisk (*) characters.

G-5 Insertion Items

The insertion directives cause data to be inserted into an output record or cause data to be skipped on an input record. In the input case, the length of the format item determines the number of characters skipped. The following subsections describe only the output formatting.

X-Format

Syntax: rX

---

where "r" (repetition factor) is an integer constant. A repetition factor must be specified.

This format item causes the processing position in the record to be advanced one position. The position advanced past has a space character inserted if the position has not yet been filled.

Character Constant

Syntax: ' ... '
or " ... "

---

where the periods (.) indicate the contents of the character constant.

This format item causes the CHARACTER constant to be inserted into the output record.

T-Format

Syntax: Tp

---

where "p" (position) is an integer constant.

This format item causes the current processing position of the record to be "p". This causes spaces to be inserted if "p" positions have not yet been filled on an output record.

TL-Format

Syntax: TLp

---

where "p" (position) is an integer constant.

This format item causes the current processing position of the record to be "p" characters to the left of the current position. On input, this enables part of the current record to be reprocessed. On output, this may cause characters already transmitted to be replaced. An attempt to advance beyond the first position of the record causes the first position in the record to become the current processing position.

TR-Format

Syntax: TRp

---

where "p" (position) is an INTEGER constant.

This format item causes the current processing position of the record to be "p" characters to the right of the current position. On input, this enables part of the current record to be skipped. On output, space characters are inserted in positions to which data has not yet been transmitted. The last position in the record becomes the current processing position when an attempt is made to tab beyond the record.

G-6 Other Items

These items affect the transmission of data values to and from records. It is not necessary to separate them from other format items with comma (,) characters.

/ format item

On input, this item causes the next record in the file to be read and processing to continue at the start of the new record. On output, the current record is written and processing of the next record in the file commences.

: *format item*

The processing of the format specifications for READ, WRITE, or PRINT statement is completed when this item is encountered and no more items remain to be processed in the current input/output list.

Notes

Chapter H

Miscellaneous Statements

H-1 Introduction

This chapter describes several miscellaneous statements which are not described elsewhere.

H-2 END Statement

Syntax: END

The END statement is used to mark the end of a program unit (main program, function or subroutine). It must be the last statement in each program unit.

When an END statement is encountered while executing the main program, execution of the program terminates. When an END statement is encountered in a function or subroutine, control is returned to the program unit which invoked the function or subroutine in question.

H-3 STOP Statement

Syntax: STOP

The execution of a STOP statement causes the execution of the program to terminate.

H-4 PAUSE Statement

Syntax: PAUSE

The execution of a PAUSE statement causes the execution of a program to be suspended and the FORTRAN Debugger to be activated (see Debugging). When the debugging command CONTINUE is issued, program execution will continue with the statement following the PAUSE statement in question.

H-5 GOTO Statement

Syntax: GOTO stmnt
or GO TO stmnt

The execution of a GOTO statement causes control to be transferred to the statement (the target statement) with the statement number indicated in the GOTO statement. The target statement must be present in the same program unit as the GOTO statement. It is illegal for the target statement to be located in a structured loop, DO-loop, IF-group, GUESS-group or remote block in which the GOTO statement is not also found. It is legal to transfer control from one of these structures to a target statement outside the bounds of the structure.

H-6 Computed GOTO Statement

Syntax: GOTO (number, ..., number), expression
GOTO (number, ..., number) expression
GO TO (number, ..., number), expression
GO TO (number, ..., number) expression

The computed GOTO statement specifies a parenthesized list of statement numbers, followed by a numeric expression. The execution of this statement causes the expression to be evaluated. When the result is REAL it is truncated to be an INTEGER value. When the result is non-positive or larger than the number of statement numbers in the list, control continues at the statement following the computed GOTO statement. Otherwise, control continues at the statement indicated by the K-th statement number in the list, where K is the result of the evaluation of the expression.

The target statements must all be present in the same program unit. It is illegal for a target statement to be located in a structured loop, DO-loop, IF-group, GUESS-block, or remote block, unless the IF statement is also contained in that structure. It is legal to transfer control from one of these structures to a target statement outside the bounds of the structure, except in the case of a remote block.

H-7 RETURN Statement

Syntax: RETURN

The execution of a RETURN statement causes control to be returned from the current program unit to the program unit which invoked the current program unit. It is illegal to execute a RETURN statement in the main program.

H-8 Logical IF Statement

Syntax: IF (expression) THEN statement
IF (expression) statement

The execution of this statement causes the parenthesized expression to be evaluated. When the result is false (zero), execution continues with the statement following the IF statement. When the result is true (non-zero), the statement following the optional THEN keyword is executed and then execution continues with the statement following the IF statement. The statement following the IF statement must be executable and cannot be any of the following statements; IF, FORMAT, END, INTEGER, REAL, CHARACTER, EXTERNAL, LOOP, WHILE, DO, GUESS, ELSE, ELSEIF, ADMIT, ENDIF, ENDLOOP, UNTIL, ENDGUESS, QUITIF, SUBROUTINE, FUNCTION, REMOTE BLOCK or END BLOCK.

H-9 Arithmetic IF statement

Syntax: IF (expression) number, number, number

The arithmetic IF statement contains a parenthesized numeric expression, followed by three statement numbers. The execution of the statement causes the expression to be evaluated. When the result is negative, control is passed to the statement at the first statement number; when the result is zero, control is passed to the statement at the second statement number; and, when the result is positive, control is passed to the statement at the third statement number.

The target statements must all be present in the same program unit. It is illegal for a target statement to be located in a structured loop, DO-loop, IF-group, GUESS-block, or remote block, unless the IF statement is also contained in that structure. It is legal to transfer control from one of these structures to a target statement outside the bounds of the structure, except for a remote block.

H-10 DO Statement

```
Syntax:    DO stmnt var = expr, expr
     or    DO stmnt var = expr, expr, expr
           .... body of DO loop
     stmnt Statement
```

---

where "expr" is a numeric expression, "var" is a simple (unsubscripted) variable and "stmnt" is a statement number.

The DO is similar to a structured DO loop (see DO loop, structured), except for the manner in which the body of the loop is specified. The DO statement specifies a statement number giving the last statement in the loop. In the structured case, the ENDDO keyword is used to make the end of the loop. In all other manners, the DO is equivalent to the structured DO.

It should be noted that this definition of a DO statement is different from those found in earlier versions of FORTRAN, e.g., FORTRAN IV. Noteably, the body of the DO in FORTRAN IV was always executed at least once; in Waterloo microFORTRAN and FORTRAN-77 the body may not be executed as the termination criteria are tested prior to executing the loop for the first time.

The statement referenced by the DO statement may not be one of the following statements: structured IF, ELSEIF, ELSE, ENDIF, LOOP, WHILE, ENDLOOP, UNTIL, GUESS, ADMIT, ENDGUESS, DO, PROGRAM, FORMAT, SUBROUTINE, FUNCTION, INTEGER, REAL, CHARACTER, EXTERNAL. The structured statements QUITIF and QUIT may be used in DO loops in the same way as they are used in structured loops.

Several nested DO-loops may apparently terminate in the same statement by all specifying the same statement number. In this case, the statement actually is part of only the innermost loop. Another iteration of the outer loop occurs when the adjacent inner loop completes. As a consequence of this situation, it is illegal to use a GOTO statement to transfer control to the statement at the end of the loop, unless the GOTO is located in the innermost loop.

H-11 CONTINUE statement

Syntax: CONTINUE

This statement has no effect on the execution of a program. It is commonly used, in conjunction with a statement number, as the last statement in a DO loop.

Chapter I

FORTRAN Debugger

I-1 Introduction

Most programs have errors embedded within them when they are first entered into the computer. Sometimes these errors are logic errors introduced by the programmer; in other cases, the errors are clerical in nature and caused by incorrectly entering a program. Such errors are commonly termed "bugs" and the process by which they are removed is often called "debugging". The purpose of the FORTRAN Debugger is to facilitate the removal of these bugs.

The Debugger is a subsystem which is invoked whenever an error is encountered during the program execution, when a PAUSE statement is executed or when the "RUN STOP" key is depressed. The facilities of the Debugger can then be used to inspect contents of variables, test program units, continue or retry execution, or terminate execution completely and return to the editing subsystem.

The following sections describe the various commands which can be issued in the Debugger. All commands are entered as a single letter; for example, the CONTINUE command is entered as the letter "c".

I-2 CONTINUE (c)

The CONTINUE command causes the execution of the program to continue. When the Debugger was entered by executing a PAUSE statement, execution continues following that PAUSE statement. When an error caused the Debugger to be entered, the CONTINUE command will cause execution to resume at the statement in which the error was detected.

I-3 QUIT (q)

The QUIT command causes execution to be terminated and the editing subsystem to be invoked.

I-4 EXECUTE (e)

Syntax: e statement

This command causes a FORTRAN statement to be executed, as if that statement were inserted into the program (followed by a PAUSE statement) immediately before the statement which caused the Debugger to be invoked. The following statements may be executed in this way: OPEN, CLOSE, PRINT, READ, WRITE, REWIND, GOTO, EXECUTE, CALL and assignment. The specified statement cannot have a statement number preceding it. The successful execution of a GOTO statement causes the Debugger to be terminated and execution to continue normally at the target statement.

This statement has many powerful uses when debugging programs. For example, the contents of variables may be inspected by executing a PRINT statement:

```
e print,x,y
```

The preceding example will cause the contents of variables x and y to be displayed. A subroutine may be tested by executing a CALL statement and a function may be invoked by executing a function reference in an expression. In these ways, it is often possible to quickly determine the causes of errors.

Sometimes the error may be temporarily corrected by executing a statement. For example, an attempt to use an undefined value might be corrected by executing an assignment statement placing the correct value in appropriate data item. In other

cases, the Debugger should be terminated (see QUIT command) and the program should be changed and re-executed.

It is possible that an error may arise while executing a statement with the EXECUTE command. In this case, the error is diagnosed and the state of the Debugger is returned to the environment that existed before the execution of the statement from the Debugger. Thus, the suspended statement is the one at which the Debugger was originally invoked.

The implementation of Waterloo microFORTRAN prohibits the introduction of a symbol name not already used in the program. When an attempt to use a new name occurs, an error message diagnosing the attempt is displayed.

I-5 WHERE-AM-I (w)

When the Debugger is entered, the statement that caused the entry is displayed. The WHERE command is used to obtain a display of the program units that are currently active. In this way, the flow of control can be analyzed to determine where and how the indicated statement was encountered.

I-6 STEP (s)

This command causes the program to execute the single statement at which the Debugger has suspended execution. Depressing the RETURN key another time causes the next statement to execute. In some implementations, keeping the RETURN key depressed causes the program to execute with each line to be executed displayed immediately before it is executed. In this way, the flow of control may be precisely viewed.

Chapter J

System Dependencies

J-1 Introduction

Broadly speaking, system dependencies are introduced by the hardware used to execute a FORTRAN program and by the system programs (for example, a file system) used by the FORTRAN processors. Thus, the precision and the maximum or minimum magnitude of numeric quantities may vary from computer to computer. Similarly, the format of file names may be different on various systems. In this chapter, these system dependencies are outlined.

J-2 System Dependencies for Commodore SuperPET

This section deals with the system dependencies for the Commodore SuperPET computer.

REAL Numbers

REAL numbers are represented internally in five bytes, the first of which contains a sign and the exponent. The latter four bytes are the binary fraction (normalized).

This representation permits approximately nine digits of accuracy and allows non-zero positive numbers to fall in the approximate range 10**-38 to 10**38.

INTEGER Numbers

INTEGER numbers are represented internally in two bytes. This representation permits integer values in the range -32768 to 32767.

File Names

For a description of file names and devices attached to the Commodore SuperPET see the System Overview Manual.

ERROR CODES

The following error codes are placed in the INTEGER variable in the IOSTAT option associated with input/output statements.

code	*meaning*
0	operation was successful
2	end of file recognized
3	input/output error

SYS Intrinsic Function

The SYS intrinsic function may have from one to nine arguments. The first, mandatory argument must be an INTEGER value which is the address of a routine to be invoked. The other arguments are passed to the invoked routine as arguments in the following manner:

(a) INTEGER arguments are passed as two-byte integers.

(b) REAL arguments are copied to a temporary memory location and the address of the memory location is passed.

(c) CHARACTER arguments are copied to a temporary memory location, an extra character with hexadecimal value 0 is appended to the string, and the address of the memory location is passed.

(d) All other arguments are illegal. The arguments to be passed to the invoked routine are pushed on the the stack, in order, starting with the last and proceeding to the second. The first argument to be passed is loaded into the

A and B registers of the Motorola 6809.

When the invoked routine returns to the mcroFORTRAN system, the contents of the A and B registers are returned to the program as an INTEGER value.

These conventions are compatible with the Waterloo Library. The library is described in the manual for the 6809 Assembler.

TIME Intrinsic Function

This function returns a character string in the format "HH:MM:SS.SS" to represent the hours (HH), minutes (MM) and sixtieths of a second (SS.SS) of the current time.

DATE Intrinsic Function

The format of the character string returned by this function is identical to that entered using the DATE command in the Waterloo microEditor.

TOD and SLEEP Intrinsic Functions

The REAL values used as arguments and return values in these functions represent times in 60-ths of a second.

J-3 System Dependencies for VM/CMS:

This section deals with the system dependencies for the IBM VM/CMS operating system.

REAL Numbers

REAL numbers are represented internally in eight bytes, the first of which contains a sign and an exponent. The latter seven bytes consist of the hexadecimal fraction. This representation permits approximately fourteen digits of accuracy and allows non-zero positive numbers in the approximate range 10**-76 to 10**76.

INTEGER Numbers

INTEGER numbers are represented internally in two bytes. This representation permits integer values in the range -32768 to 32767.

File Names

A file name is given as

name type mode (option option ....)

where "name" is required and "type" and "mode" are optional. "Name" and "type" may be arbitrary 8-character names and "mode" is a two-character specification (default is "A1"). Any CMS option for the file may be specified. The most useful are:

(1) RECFM is used to specify the format of the file. A type F is fixed format, a type V is variable format, and a type A means that ASA control characters are contained on each record. Records are stored with varying lengths in V-format files.

(2) LRECL is used to specify the logical record length. For V-Format files, this attribute defines the maximum record length.

Some examples of file names are:

```
MYFILE
MYFILE   DATA      A1   (RECFM  F LRECL  100)
MYFILE   DATA      A2   (RECFM  FA)
```

Commodore Magazine

This bi-monthly magazine, published by Commodore, provides a vehicle for sharing the latest product information on Commodore systems, programming techniques, hardware interfacing, and applications for the CBM, PET, SuperPET, and VIC Systems. Each issue contains user application features, columns by leading experts, the latest news on user clubs, a question/answer hotline column, and reviews of the latest books and software.

The subscription fee is $15.00 for six issues per year within the U.S. and its possessions, and $25.00 for Canada and Mexico. Make checks payable to COMMODORE BUSINESS MACHINES, and send to:

Editor, Commodore Magazine
Commodore Business Machines, Inc.
681 Moore Road
King of Prussia, PA 19406

The Transactor

The Transactor, which is a monthly publication of Commodore-Canada, is primarily a technical periodical, containing pertinent hardware and software information for the CBM, PET, VIC, and SuperPET systems. Each issue features product reviews, hardware and software evaluations, and programming tips from the finest technical experts on Commodore products. Additionally, The Transactor contains general information such as product updates and trade show reports.

The subscription fee is $10.00 for six issues within Canada and the United States, and $13.00 for all foreign countries. Make checks payable to COMMODORE BUSINESS MACHINES, INC. and send to:

Editor, The Transactor
Commodore Business Machines, Inc.
3370 Pharmacy Avenue
Agincourt, Ontario, Canada M1W 2K4

Waterloo microFORTRAN is a dialect of FORTRAN designed to be used in research and educational environments. The language includes many of the features of FORTRAN-77, augmented with extensions to facilitate programming. Some important aspects of the system include:

- INTEGER, REAL and CHARACTER data types are supported
- CHARACTER data type is more extensive than that specified in FORTRAN-77
- Names are not restricted to six characters; both upper- and lower-case letters may be used in names
- An extensive collection of Structured Programming statements has been added; they are modelled after those used in the WATFIV-S compiler
- Statements may be entered without regard to columns
- An interactive debugging facility is available
- Sequential and random-access input/output is supported
- An extensive FORMAT capability has been implemented
- The capability to access files on large main-frame computers is an integral part of the system
- A full-screen editor may be used to enter and update programs

This book is divided into two major components:

- A collection of tutorial examples may be used to obtain the "flavor" of the language.
- The reference manual provides the comprehensive details of the language

DISTRIBUTED BY

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$10.95/21904

ISBN: 0-672-21904-2