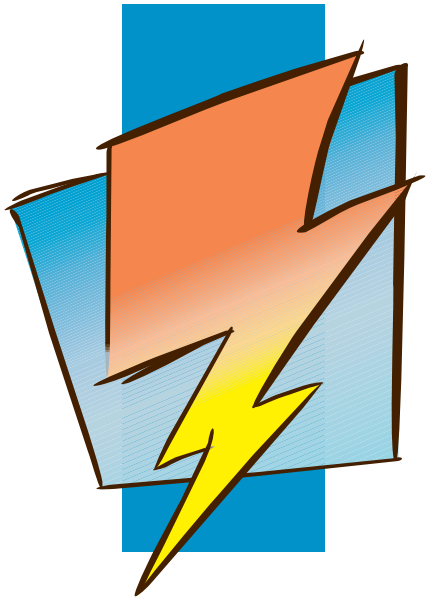
# *Open Watcom FORTRAN 77*

# *Language Reference*



***Version 2.0***

Open Watcom

# Notice of Copyright

Copyright © 2002-2023 the Open Watcom Contributors. Portions Copyright © 1984-2002 Sybase, Inc. and its subsidiaries. All rights reserved.

Any part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of anyone.

For more information please visit https://github.com/open-watcom/open-watcom-v2 .

# Preface

Open Watcom FORTRAN 77 is an implementation of the American National Standard programming language FORTRAN, ANSI X3.9-1978, commonly referred to as FORTRAN 77. The language level supported by Open Watcom FORTRAN 77 compilers includes the full language definition as well as significant extensions to the language. Open Watcom FORTRAN 77 compilers are based upon some well known FORTRAN language compilers, namely the University of Waterloo's WATFOR and WATFIV-S compilers (implementations for the International Business Machines 370 series) and the WATFOR-11 compiler (an implementation for the Digital Equipment PDP11).

This manual describes the language level supported by Open Watcom FORTRAN 77 including extensions to the standard language. Shaded areas in the book denote a Open Watcom FORTRAN 77 language extension. Occasionally, where an entire section or chapter deals with a language extension, the text may not be shaded. Users should note that extensions which are supported by this compiler may not be supported by other compilers. We leave the choice to use a particular extension to the discretion of the programmer.

An accompanying manual, the User's Guide, contains system specific topics such as how to run the software on your system, file system support, compiler options, etc.

# Acknowledgements

This book was produced with the Open Watcom GML electronic publishing system, a software tool developed by WATCOM. In this system, writers use an ASCII text editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The Open Watcom GML software, which runs on a variety of operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

Much of the information contained in this document was taken from the ANSI publication "American National Standard Programming Language FORTRAN, ANSI X3.9-1978". We recommend that anyone who is interested in the definitive description of FORTRAN 77 obtain a copy of this document. Their address is: American National Standards Institute, Inc., 1430 Broadway, New York, New York, U.S.A. 10018.

July, 1997.

# Table of Contents


Language Reference ..... 1

1 FORTRAN Source Program Format ..... 3
1.1 Character Set ..... 3
1.2 Extended Character Set ..... 3
1.3 Source Program Format ..... 4
1.3.1 Comment Line ..... 4
1.3.2 Debug Line (Extension) ..... 4
1.3.3 Initial Line ..... 4
1.3.4 Continuation Line ..... 4
1.3.5 Significance of the Blank Character ..... 5
1.3.6 Significance of Lower Case Characters (Extension) ..... 5
1.3.7 Examples ..... 5
1.4 Order of FORTRAN Statements and Lines ..... 6

2 FORTRAN Statements ..... 9
2.1 Classifying Statements ..... 9
2.2 FORTRAN Statement Summary ..... 11
2.3 ADMIT Statement ..... 12
2.4 ALLOCATE Statement ..... 13
2.5 Statement Label Assignment (ASSIGN) Statement ..... 16
2.6 AT END Statement ..... 18
2.7 BACKSPACE Statement ..... 19
2.8 BLOCK DATA Statement ..... 21
2.9 CALL Statement ..... 23
2.10 CASE Statement ..... 25
2.11 CHARACTER Statement ..... 26
2.11.1 Standard CHARACTER Statement ..... 26
2.11.2 Extended CHARACTER Statement: Data Initialization ..... 28
2.12 CLOSE Statement ..... 29
2.13 COMMON Statement ..... 31
2.14 COMPLEX Statement ..... 33
2.14.1 Standard COMPLEX Statement ..... 33
2.14.2 Extended COMPLEX Statement: Length Specification ..... 33
2.14.3 Extended COMPLEX Statement: Data Initialization ..... 34
2.15 CONTINUE Statement ..... 36
2.16 CYCLE Statement ..... 37
2.17 DATA Statement ..... 38
2.18 DEALLOCATE Statement ..... 42
2.19 DIMENSION Statement ..... 44
2.20 DO Statement ..... 45
2.20.1 Standard DO Statement ..... 45
2.20.2 Extended DO Statement ..... 45
2.20.3 Description of DO Statement ..... 45
2.21 DOUBLE COMPLEX Statement ..... 49
2.21.1 Simple DOUBLE COMPLEX Statement ..... 49
2.21.2 DOUBLE COMPLEX Statement: Data Initialization ..... 49
2.22 DOUBLE PRECISION Statement ..... 51
2.22.1 Standard DOUBLE PRECISION Statement ..... 51
2.22.2 Extended DOUBLE PRECISION Statement: Data Initialization ..... 51
2.23 DO WHILE Statement ..... 53
2.24 ELSE Statement ..... 54

# Table of Contents


2.25 ELSE IF Statement ..... 55
2.26 END Statement ..... 56
2.27 END AT END Statement ..... 57
2.28 END BLOCK Statement ..... 58
2.29 END DO Statement ..... 59
2.30 ENDFILE Statement ..... 60
2.31 END GUESS Statement ..... 62
2.32 END IF Statement ..... 63
2.33 END LOOP Statement ..... 64
2.34 END MAP Statement ..... 65
2.35 END SELECT Statement ..... 66
2.36 END STRUCTURE Statement ..... 67
2.37 END UNION Statement ..... 68
2.38 END WHILE Statement ..... 69
2.39 ENTRY Statement ..... 70
2.40 EQUIVALENCE Statement ..... 71
2.41 EXECUTE Statement ..... 73
2.42 EXIT Statement ..... 74
2.43 EXTERNAL Statement ..... 75
2.44 FORMAT Statement ..... 76
2.45 FUNCTION Statement ..... 77
2.45.1 Standard FUNCTION Statement ..... 77
2.45.2 Extended FUNCTION Statement ..... 78
2.46 Unconditional GO TO Statement ..... 79
2.47 Computed GO TO Statement ..... 80
2.48 Assigned GO TO Statement ..... 81
2.49 GUESS Statement ..... 83
2.50 Arithmetic IF Statement ..... 84
2.51 Logical IF Statement ..... 85
2.52 Block IF Statement ..... 87
2.52.1 Standard Block IF Statement ..... 87
2.52.2 Extended Block IF Statement ..... 87
2.53 IMPLICIT Statement ..... 89
2.53.1 Standard IMPLICIT Statement ..... 89
2.53.2 Extended IMPLICIT Statement ..... 89
2.53.3 IMPLICIT NONE Statement ..... 90
2.53.4 Description of IMPLICIT Statement ..... 90
2.54 INCLUDE Statement ..... 92
2.55 INQUIRE Statement ..... 93
2.55.1 INQUIRE by FILE ..... 93
2.55.2 INQUIRE by UNIT ..... 93
2.55.3 Inquiry Specifiers ..... 94
2.55.4 Definition Status of Specifiers - Inquire by File ..... 97
2.55.5 Definition Status of Specifiers - Inquire by Unit ..... 98
2.56 INTEGER Statement ..... 99
2.56.1 Standard INTEGER Statement ..... 99
2.56.2 Extended INTEGER Statement: Length Specification ..... 99
2.56.3 Extended INTEGER Statement: Data Initialization ..... 100
2.57 INTRINSIC Statement ..... 102
2.58 LOGICAL Statement ..... 103
2.58.1 Standard LOGICAL Statement ..... 103
2.58.2 Extended LOGICAL Statement: Length Specification ..... 103

# Table of Contents


2.58.3 Extended LOGICAL Statement: Data Initialization ..... 104
2.59 LOOP Statement ..... 106
2.60 MAP Statement ..... 107
2.61 NAMELIST Statement ..... 108
2.62 OPEN Statement ..... 111
2.63 OTHERWISE Statement ..... 115
2.64 PARAMETER Statement ..... 116
2.65 PAUSE Statement ..... 117
2.66 PRINT Statement ..... 118
2.66.1 Standard PRINT Statement ..... 118
2.66.2 Extended PRINT Statement ..... 118
2.66.3 Description of PRINT Statement ..... 118
2.67 PROGRAM Statement ..... 121
2.68 QUIT Statement ..... 122
2.69 READ Statement ..... 123
2.69.1 Standard READ Statement ..... 123
2.69.2 Extended READ Statement ..... 123
2.69.3 Description of READ Statement ..... 123
2.70 REAL Statement ..... 127
2.70.1 Standard REAL Statement ..... 127
2.70.2 Extended REAL Statement: Length Specification ..... 127
2.70.3 Extended REAL Statement: Data Initialization ..... 128
2.71 RECORD Statement ..... 130
2.72 REMOTE BLOCK Statement ..... 131
2.73 RETURN Statement ..... 132
2.74 REWIND Statement ..... 133
2.75 SAVE Statement ..... 135
2.76 SELECT Statement ..... 137
2.77 STOP Statement ..... 139
2.78 STRUCTURE Statement ..... 140
2.79 SUBROUTINE Statement ..... 141
2.80 UNION Statement ..... 142
2.81 UNTIL Statement ..... 143
2.82 VOLATILE Statement ..... 144
2.83 Block WHILE Statement ..... 145
2.84 WHILE Statement ..... 146
2.85 WRITE Statement ..... 147

3 Names, Data Types and Constants ..... 151
3.1 Symbolic Names ..... 151
3.2 Data Types ..... 152
3.3 Data Type of a Name ..... 152
3.4 Constants ..... 153
3.4.1 Integer Constants ..... 153
3.4.2 Real Constants ..... 153
3.4.3 Double Precision Constant ..... 154
3.4.4 Complex Constant ..... 154
3.4.5 Double Precision Complex Constant (Extension) ..... 154
3.4.6 Logical Constant ..... 155
3.4.7 Character Constant ..... 155
3.4.8 String Constant (Extension) ..... 155
3.4.9 Hollerith Constants (Extension) ..... 155

# Table of Contents


3.4.10 Hexadecimal Constants (Extension) ..... 156
3.4.11 Octal Constants (Extension) ..... 156
3.5 Symbolic Constants ..... 157

4 Arrays ..... 159
4.1 Introduction ..... 159
4.2 Properties of Arrays ..... 159
4.3 Array Elements ..... 160
4.4 Classifying Array Declarators by Dimension Declarator ..... 162
4.4.1 Constant Array Declarator ..... 162
4.4.2 Adjustable Array Declarator ..... 162
4.4.3 Assumed-size Array Declarator ..... 162
4.4.4 Allocatable Array Declarator ..... 163
4.5 Classifying Array Declarators by Array Name ..... 164
4.5.1 Actual Array Declarator ..... 164
4.5.2 Dummy Array Declarator ..... 164
4.6 Use of Array Names ..... 164

5 Character Substrings ..... 165
5.1 Introduction ..... 165
5.2 Substring Names ..... 165
5.3 Extensions ..... 166

6 Structures, Unions and Records ..... 167
6.1 Structures and Records ..... 167
6.2 Arrays of Records ..... 168
6.3 Unions ..... 169

7 Expressions ..... 173
7.1 Arithmetic Expressions ..... 173
7.1.1 Arithmetic Operators ..... 173
7.1.2 Rules for Forming Standard Arithmetic Expressions ..... 174
7.1.3 Arithmetic Constant Expression ..... 176
7.1.4 Data Type of Arithmetic Expressions ..... 177
7.2 Character Expressions ..... 178
7.2.1 Character Operators ..... 178
7.2.2 Rules for Forming Character Expressions ..... 178
7.2.3 Character Constant Expressions ..... 179
7.3 Relational Expressions ..... 179
7.3.1 Relational Operators ..... 179
7.3.2 Form of a Relational Expression ..... 179
7.3.2.1 Arithmetic Relational Expressions ..... 180
7.3.2.2 Character Relational Expressions ..... 180
7.4 Logical Expressions ..... 180
7.4.1 Logical Operators ..... 181
7.4.2 Rules for Forming Logical Expressions ..... 183
7.4.3 Logical Constant Expressions ..... 184
7.5 Evaluating Expressions ..... 184
7.6 Constant Expressions ..... 185

8 Assignment Statements ..... 187
8.1 Introduction ..... 187

# Table of Contents


8.2 Arithmetic Assignment ..... 187
8.3 Logical Assignment ..... 188
8.4 Statement Label Assignment ..... 188
8.5 Character Assignment ..... 189
8.6 Extended Assignment Statement ..... 190

9 Program Structure Control Statements ..... 193
9.1 Introduction ..... 193
9.2 IF - ELSE - END IF ..... 193
9.3 ELSE IF ..... 194
9.4 DO - END DO ..... 196
9.5 DO WHILE - END DO ..... 197
9.6 LOOP - END LOOP ..... 198
9.7 WHILE - END WHILE ..... 199
9.8 WHILE - Executable-statement ..... 199
9.9 UNTIL ..... 200
9.10 SELECT - END SELECT ..... 201
9.11 EXECUTE and REMOTE BLOCK ..... 204
9.12 GUESS-ADMIT-END GUESS ..... 206
9.13 QUIT ..... 208
9.14 EXIT ..... 209
9.15 CYCLE ..... 210
9.16 AT END ..... 211
9.17 Notes on Structured Programming Statements ..... 212

10 Input/Output ..... 215
10.1 Introduction ..... 215
10.2 Reading and Writing ..... 215
10.3 Records ..... 216
10.3.1 Formatted Record ..... 216
10.3.2 Unformatted Record ..... 216
10.3.3 Endfile Record ..... 216
10.4 Files ..... 217
10.4.1 External Files ..... 217
10.4.2 Internal Files ..... 218
10.5 Units ..... 219
10.6 Specifiers ..... 221
10.6.1 The Unit Specifier ..... 221
10.6.2 Format Specifier ..... 221
10.6.3 Record Specifier ..... 222
10.6.4 Input/Output Status Specifier ..... 222
10.6.5 Error Specifier ..... 222
10.6.6 End-of-File Specifier ..... 223
10.7 Printing of Formatted Records ..... 223

11 Format ..... 225
11.1 Introduction ..... 225
11.2 The FORMAT Statement ..... 225
11.3 FORMAT as a Character Expression ..... 225
11.4 Format Specification ..... 226
11.5 Repeatable Edit Descriptors ..... 227
11.6 Nonrepeatable Edit Descriptors ..... 228

# Table of Contents


11.7 Editing ..... 228
11.7.1 Apostrophe Editing ..... 229
11.7.2 H Editing ..... 229
11.7.3 Positional Editing: T, TL, TR and X Editing ..... 229
11.7.4 Slash Editing ..... 230
11.7.5 Colon Editing ..... 230
11.7.6 S, SP and SS Editing ..... 230
11.7.7 P Editing ..... 230
11.7.8 BN and BZ Editing ..... 231
11.7.9 $ or \ Editing (Extension) ..... 231
11.7.10 Numeric Editing: I, F, E, D and G Edit Descriptors ..... 232
11.7.10.1 Integer Editing: Iw and Iw.m Edit Descriptors ..... 232
11.7.10.2 Floating-point Editing: F, E, D and G Edit Descriptors ..... 233
11.7.10.3 F Editing ..... 233
11.7.10.4 E and D Editing ..... 234
11.7.10.5 G Editing ..... 235
11.7.10.6 Complex Editing ..... 236
11.7.11 L Edit Descriptor ..... 236
11.7.12 A Edit Descriptor ..... 236
11.7.13 Z Editing (Extension) ..... 237
11.8 Format-Directed Input/Output ..... 238
11.9 List-Directed Formatting ..... 238
11.9.1 List-Directed Input ..... 239
11.9.2 List-Directed Output ..... 240
11.10 Namelist-Directed Formatting (Extension) ..... 240
11.10.1 Namelist-Directed Input (Extension) ..... 241
11.10.2 Namelist-Directed Output ..... 242

12 Functions and Subroutines ..... 243
12.1 Introduction ..... 243
12.2 Statement Functions ..... 243
12.2.1 Referencing a Statement Function ..... 244
12.2.2 Statement Function Restrictions ..... 245
12.3 Intrinsic Functions ..... 246
12.3.1 Specific Names and Generic Names of Intrinsic Functions ..... 246
12.3.2 Type Conversion: Conversion to integer ..... 248
12.3.3 Type Conversion: Conversion to real ..... 248
12.3.4 Type Conversion: Conversion to double precision ..... 249
12.3.5 Type Conversion: Conversion to complex ..... 249
12.3.6 Type Conversion: Conversion to double complex ..... 249
12.3.7 Type Conversion: Character conversion to integer ..... 250
12.3.8 Type Conversion: Conversion to character ..... 250
12.3.9 Truncation ..... 250
12.3.10 Nearest Whole Number ..... 251
12.3.11 Nearest Integer ..... 251
12.3.12 Absolute Value ..... 251
12.3.13 Remainder ..... 252
12.3.14 Transfer of Sign ..... 252
12.3.15 Positive Difference ..... 253
12.3.16 Double Precision Product ..... 253
12.3.17 Choosing Largest Value ..... 253
12.3.18 Choosing Smallest Value ..... 254

# Table of Contents


12.3.19 Length ..... 254
12.3.20 Length Without Trailing Blanks ..... 254
12.3.21 Index of a Substring ..... 255
12.3.22 Imaginary Part of Complex Number ..... 255
12.3.23 Conjugate of a Complex Number ..... 255
12.3.24 Square Root ..... 256
12.3.25 Exponential ..... 256
12.3.26 Natural Logarithm ..... 257
12.3.27 Common Logarithm ..... 257
12.3.28 Sine ..... 258
12.3.29 Cosine ..... 258
12.3.30 Tangent ..... 259
12.3.31 Cotangent ..... 259
12.3.32 Arcsine ..... 259
12.3.33 Arccosine ..... 260
12.3.34 Arctangent ..... 260
12.3.35 Hyperbolic Sine ..... 261
12.3.36 Hyperbolic Cosine ..... 261
12.3.37 Hyperbolic Tangent ..... 261
12.3.38 Gamma Function ..... 262
12.3.39 Natural Log of Gamma Function ..... 262
12.3.40 Error Function ..... 262
12.3.41 Complement of Error Function ..... 263
12.3.42 Lexically Greater Than or Equal ..... 263
12.3.43 Lexically Greater Than ..... 263
12.3.44 Lexically Less Than or Equal ..... 263
12.3.45 Lexically Less Than ..... 264
12.3.46 Binary Pattern Processing Functions: Boolean AND ..... 264
12.3.47 Binary Pattern Processing Functions: Boolean Inclusive OR ..... 264
12.3.48 Binary Pattern Processing Functions: Boolean Exclusive OR ..... 265
12.3.49 Binary Pattern Processing Functions: Boolean Complement ..... 265
12.3.50 Binary Pattern Processing Functions: Logical Shift ..... 265
12.3.51 Binary Pattern Processing Functions: Arithmetic Shift ..... 266
12.3.52 Binary Pattern Processing Functions: Circular Shift ..... 267
12.3.53 Binary Pattern Processing Functions: Bit Testing ..... 268
12.3.54 Binary Pattern Processing Functions: Set Bit ..... 268
12.3.55 Binary Pattern Processing Functions: Clear Bit ..... 268
12.3.56 Binary Pattern Processing Functions: Change Bit ..... 269
12.3.57 Binary Pattern Processing Functions: Arithmetic Shifts ..... 269
12.3.58 Allocated Array ..... 270
12.3.59 Memory Location ..... 270
12.3.60 Size of Variable or Structure ..... 270
12.3.61 Volatile Reference ..... 270
12.4 External Functions ..... 271
12.4.1 Referencing an External Function ..... 272
12.4.2 Actual Arguments for an External Function ..... 272
12.4.3 External Function Subprogram Restrictions ..... 273
12.5 Subroutines ..... 273
12.5.1 Referencing a Subroutine: The CALL Statement ..... 273
12.5.2 Actual Arguments for a Subroutine ..... 273
12.5.3 Subroutine Subprogram Restrictions ..... 274
12.6 The ENTRY Statement ..... 274

# Table of Contents


12.6.1 ENTRY Statements in External Functions ..... 275
12.6.2 ENTRY Statement Restrictions ..... 275
12.7 The RETURN Statement ..... 276
12.7.1 RETURN Statement in the Main Program (Extension) ..... 276
12.7.2 RETURN Statement in Function Subprograms ..... 276
12.7.3 RETURN Statement in Subroutine Subprograms ..... 277
12.8 Subprogram Arguments ..... 277
12.8.1 Dummy Arguments ..... 278
12.8.2 Actual Arguments ..... 278
12.8.3 Association of Actual and Dummy Arguments ..... 278
12.8.3.1 Length of Character Actual and Dummy Arguments ..... 279
12.8.3.2 Variables as Dummy Arguments ..... 279
12.8.3.3 Arrays as Dummy Arguments ..... 279
12.8.3.4 Procedures as Dummy Arguments ..... 280
12.8.3.5 Asterisks as Dummy Arguments ..... 281

Appendices ..... 283

A. Open Watcom FORTRAN 77 Extensions to Standard FORTRAN 77 ..... 285

# *Language Reference*

# 1 FORTRAN Source Program Format

## 1.1 Character Set

The FORTRAN *character set* consists of twenty-six letters, ten digits, and thirteen special characters.

The letters are:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

The digits are: `0 1 2 3 4 5 6 7 8 9`

The special characters are:

| Character | Name of Character |
|---|---|
| | Blank |
| = | Equals |
| + | Plus |
| - | Minus |
| * | Asterisk |
| / | Slash |
| ( | Left Parenthesis |
| ) | Right Parenthesis |
| , | Comma |
| . | Decimal Point |
| $ | Currency Symbol |
| ' | Apostrophe |
| : | Colon |

The FORTRAN character set is a subset of the character set of the computing system which you are using. We shall refer to the larger character set as the *processor character set.*

## 1.2 Extended Character Set

Open Watcom FORTRAN 77 also includes the following special characters.

| Character | Name of Character |
|---|---|
| ! | Exclamation Mark |
| % | Percentage Symbol |
| \ | Back slash |

# 1.3 Source Program Format

Open Watcom FORTRAN 77 supports one source program format. A FORTRAN program is composed of *lines*. There are three types of lines; the *comment* line, the *initial* line, and the *continuation* line.

## 1.3.1 Comment Line

*Comment* lines are denoted by placing a "C" or "*" in column one of the line. Open Watcom FORTRAN 77 also allows the use of a lowercase "c" as a comment indicator. *Blank* lines are treated as comment lines. Comment lines may be placed anywhere in the program source (i.e., they may appear before a FORTRAN statement, they may be intermingled with continuation lines, or they may appear after a statement). There is no restriction on the number of comment lines. Comment lines may contain any characters from the processor character set.

Open Watcom FORTRAN 77 allows end-of-line comments. If a "!" character appears in column 1 or anywhere in the statement portion of a source line, the remainder of that line is treated as a comment unless the "!" appears inside quotation marks or in column 6.

## 1.3.2 Debug Line (Extension)

*Debug* lines are denoted by placing a "D" or "d" in column one of the line. Debug lines contain FORTRAN statements. There is no restriction on the number of debug lines. Normally, the FORTRAN statements on debug lines are ignored by the compiler. See the User's Guide for information on activating debug statements.

## 1.3.3 Initial Line

An *initial* line is the first line of a FORTRAN statement. Column 6 of this line must be blank or contain the digit "0". A comment line can never be an initial line. Columns 1 through 5 of an initial line may contain a *statement label.* Statement labels are composed entirely of digits. The statement label may be thought of as an integral number and, as such, leading 0 digits are not significant. For example, the label composed of the digits "00123" is the same as the label "123". The same label may not identify more than one statement in a *program unit.* A program unit is a series of comment lines and FORTRAN statements ending in an **END** statement. The body of the FORTRAN *statement* is entered starting in column 7 and stopping at column 72. Column 73 and on is called the *sequence field* and is ignored by the compiler.

## 1.3.4 Continuation Line

A statement may be *continued* on a new line. A continuation character is placed in column 6. The continuation character may not be a blank character or a "0" character. FORTRAN 77 requires that the continuation character be selected from the FORTRAN character set but Open Watcom FORTRAN 77 allows any character from the processor's character set. The statement number field must be blank. The previous statement is continued on the new line, starting in column 7 and continuing to column 72. Under the control of a compiler option, Open Watcom FORTRAN 77 permits the source statement to extend to column 132.

FORTRAN 77 allows up to 19 continuation lines to continue a statement. Open Watcom FORTRAN 77 extends this by allowing more than 19 continuation lines. A minimum of 61 continuation lines are permitted when the source statement ends at column 72. A minimum of 31 continuation lines are permitted when the source statement ends at column 132. The maximum number of continuation lines depends on the sum of the lengths of all the continuation lines.

## 1.3.5 Significance of the Blank Character

Except in the following cases, blank characters have no meaning within a program unit.

**(1)** Character and Hollerith constants.

**(2)** Apostrophe and H edit descriptors in format specifications.

For example, the symbolic name `A B` is the same as the symbolic name `AB`.

## 1.3.6 Significance of Lower Case Characters (Extension)

Except in the following cases, lower case characters are treated as if they were the upper case equivalent. This is a Open Watcom FORTRAN 77 extension to the usual rules of FORTRAN.

**(1)** Character and Hollerith constants.

**(2)** Apostrophe and H edit descriptors in format specifications.

Hence, `TOTAL`, `total`, and `Total` represent the same symbolic name and `3F10.2` and `3f10.2` represent the same format edit descriptor.

## 1.3.7 Examples

*Example:*

```
C This and the following five lines are comment lines.
c The following statement "INDEX = INDEX + 2" has a
c statement number and is continued by placing a "$"
c in column 6.
* Column Numbers
*234567890

10    INDEX = INDEX
     $ + 2

* The above blank lines are treated like comment lines.
```

The following example demonstrates the use of comment lines, blanks lines, and continuation lines. We use the symbol "$" to denote continuation lines although any character other than a blank or "0" could have been used.

*Example:*

```
* From the quadratic equation
*
*       2
*     ax  + bx + c = 0
*
* we derive the following two equations:
*                 ____________
*          +   / 2
*       -b - \/ b  - 4ac
*  x = ---------------------
*               2a
*
* and express these equations in FORTRAN as:

      X1 = ( -B + SQRT( B**2 - 4 * A * C ) )
     $   /           ( 2 * A )

      X2 = ( -B - SQRT( B**2 - 4 * A * C ) )
     $   /           ( 2 * A )
```

# 1.4 Order of FORTRAN Statements and Lines

The first statement of a program unit may be a **PROGRAM**, **FUNCTION**, **SUBROUTINE**, or **BLOCK DATA** statement. The **PROGRAM** statement identifies the start of a main program and there may only be one of these in an executable FORTRAN program. Execution of a FORTRAN program begins with the first *executable* statement in the main program. The other statements identify the start of a subprogram. If the first statement of a program unit is not one of the above then the program unit is considered to be a main program.

Although you may not be familiar with all of the terms used here, it is important that you understand that FORTRAN 77 has specific rules regarding the ordering of FORTRAN statements. You may wish to refer to this section at later times. In general, the following rules apply to the order of statements and comment lines within a program unit:

1. Comment lines and **INCLUDE** statements may appear anywhere.
2. **FORMAT** statements may appear anywhere in a subprogram.
3. All specification statements must precede all **DATA** statements, *statement function* statements, and executable statements.
4. All statement function statements must precede all executable statements.
5. **DATA** statements may appear anywhere after the specification statements.
6. **ENTRY** statements may appear anywhere except between a block **IF** statement and its corresponding **END IF** statement, or between a **DO** statement and its corresponding terminal statement. Open Watcom FORTRAN 77 extends these rules to apply to all program structure blocks resulting from the use of statements introduced to the language by Open Watcom FORTRAN 77 (e.g., **WHILE**, **LOOP**, **SELECT**).
7. **IMPLICIT** statements must precede all other specification statements, except **PARAMETER** statements. A specification statement that defines the type of a symbolic constant must appear

before the **PARAMETER** statement that defines the name and value of a symbolic constant. A **PARAMETER** statement that defines the name and value of a symbolic constant must precede all other statements containing a reference to that symbolic constant.

The following chart illustrates the required order of FORTRAN statements. Vertical lines delineate varieties of statements that may be interspersed, while horizontal lines mark varieties of statements that may not be interspersed.

<table>
<tr><td rowspan="4">Comment Lines<br><br>INCLUDE Statement</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="3">ENTRY and FORMAT Statements</td><td>PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Other Specification Statements</td></tr>
<tr><td>Statement Function Statements<br><br>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

*Required Order of Comment Lines and Statements*

For example, **DATA** statements may be interspersed with *statement function* statements and executable statements but *statement function* statements must precede executable statements.

# 2 FORTRAN Statements

## 2.1 Classifying Statements

The following table is a summary of Open Watcom FORTRAN 77 statement classification.

***Column 1*** indicates that the statement is a specification statement.

***Column 2*** indicates that the statement is not allowed as the terminal statement of a DO-loop.

***Column 3*** indicates that the statement is not executable.

***Column 4*** indicates that the statement is not allowed as the object of a logical **`IF`** or **`WHILE`** statement.

***Column 5*** indicates that the statement cannot have control of execution transferred to it by using a statement label.

***Column 6*** indicates that the statement is allowed in a block data subprogram.

| Statement | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| ADMIT | | * | | * | * | |
| ALLOCATE | | | | | | |
| ASSIGN | | | | | | |
| AT END | | * | | * | * | |
| BACKSPACE | | | | | | |
| BLOCK DATA | | * | * | * | * | |
| CALL | | | | | | |
| CASE | | * | | * | * | |
| CHARACTER | * | * | * | * | * | * |
| CLOSE | | | | | | |
| COMMON | * | * | * | * | * | * |

| Statement | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| COMPLEX | * | * | * | * | * | * |
| CONTINUE | | | | | | |
| CYCLE | | | | | | |
| DATA | | * | * | * | * | * |
| DEALLOCATE | | | | | | |
| DIMENSION | * | * | * | * | * | * |
| DO | | * | | * | | |
| DOUBLE COMPLEX | * | * | * | * | * | * |
| DOUBLE PRECISION | * | * | * | * | * | * |
| DO WHILE | | * | | * | | |
| ELSE | | * | | * | * | |
| ELSE IF | | * | | * | * | |
| END | | * | | * | | * |
| END AT END | | * | | * | * | |
| END BLOCK | | * | | * | * | |
| END DO | | | | * | * | |
| ENDFILE | | | | | | |
| END GUESS | | * | | * | | |
| END IF | | * | | * | | |
| END LOOP | | * | | * | * | |
| END MAP | * | * | * | * | * | * |
| END SELECT | * | * | * | * | * | |
| END STRUCTURE | * | * | * | * | * | * |
| END UNION | | * | | * | * | * |
| END WHILE | | * | | * | * | |
| ENTRY | | * | | * | * | |
| EQUIVALENCE | * | * | * | * | * | * |
| EXECUTE | | | | | | |
| EXIT | | | | | | |
| EXTERNAL | * | * | * | * | * | |
| FORMAT | | * | * | * | * | |
| FUNCTION | | * | * | * | * | |
| assigned GO TO | | * | | | | |
| computed GO TO | | | | | | |
| unconditional GO TO | | * | | | | |
| GUESS | | * | | * | | |
| arithmetic IF | | * | | | | |
| logical IF | | | | * | | |
| block IF | | * | | * | | |

| Statement | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| IMPLICIT | * | * | * | * | * | * |
| INCLUDE | | * | * | * | * | * |
| INQUIRE | | | | | | |
| INTEGER | * | * | * | * | * | * |
| INTRINSIC | * | * | * | * | * | |
| LOGICAL | * | * | * | * | * | * |
| LOOP | | * | | * | | |
| MAP | * | * | * | * | * | * |
| NAMELIST | * | * | * | * | * | |
| OPEN | | | | | | |
| OTHERWISE | | * | | * | * | |
| PARAMETER | * | * | * | * | * | * |
| PAUSE | | | | | | |
| PRINT | | | | | | |
| PROGRAM | | * | * | * | * | |
| QUIT | | | | | | |
| READ | | | | | | |
| REAL | * | * | * | * | * | * |
| RECORD | * | * | * | * | * | * |
| REMOTE BLOCK | | * | * | * | * | |
| RETURN | | * | | | | |
| REWIND | | | | | | |
| SAVE | * | * | * | * | * | * |
| SELECT | | * | | * | | |
| STOP | | * | | | | |
| STRUCTURE | * | * | * | * | * | * |
| SUBROUTINE | | * | * | * | * | |
| UNION | * | * | * | * | * | * |
| UNTIL | | * | | * | * | |
| VOLATILE | * | * | * | * | * | * |
| WHILE | | * | | * | | |
| WRITE | | | | | | |

## 2.2 FORTRAN Statement Summary

The following sections describe each FORTRAN 77 statement. The statement descriptions are organized alphabetically for quick reference. The syntax models for each statement are presented in shaded or unshaded boxes. The unshaded box denotes a standard FORTRAN 77 statement. The shaded box denotes a Open Watcom FORTRAN 77 extension to the language. Users should note that extensions which are supported by this compiler may not be supported by other compilers. We leave the choice to use a particular extension to the discretion of the programmer.

In the following sections the use of square brackets ([ ]) denotes items which may be optionally specified. The use of the ellipsis (...) denotes items which may be repeated as often as desired.

# 2.3 ADMIT Statement

```
ADMIT
```

The **ADMIT** statement is used in conjunction with the structured **GUESS** statement. The **ADMIT** statement marks the beginning of an alternative block of statements that are executed if a **QUIT** statement is executed in a previous **GUESS** or **ADMIT** block.

*Example:*

```
*     Assume incorrect sex code
      GUESS
          IF( SEX .EQ. 'F' )QUIT
          IF( SEX .EQ. 'M' )QUIT
          PRINT *, 'Invalid sex code encountered'
          CALL INVSEX( SEX )
          .
          .
          .
*     Wrong assumption - sex code is fine
      ADMIT
          .
          .
          .
      END GUESS
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.4 ALLOCATE Statement

```
ALLOCATE (array([l:]u[,[l:]u,...])[,...][,LOCATION=loc])
    or
ALLOCATE (array([l:]u[,[l:]u,...])[,...][,STAT=ierr])
    or
ALLOCATE (char*len)
```

***where:***

***array*** is the name of an allocatable array.

***l*** is an integer expression that sets the lower bound of the array dimension.

***u*** is an integer expression that sets the upper bound of the array dimension.

***char*** is the name of an allocatable character variable.

***len*** is an integer expression that sets the length of the character variable.

***LOCATION = loc***

`loc` is an integer expression that specifies the location of the allocated memory.

***STAT = ierr***

`ierr` is an allocation status specifier. The integer variable or integer array element `ierr` is defined with 0 if the allocation succeeded, 1 if the allocation failed, and 2 if the array is already allocated. The **`STAT=`** specifier may not be used with the **`LOCATION=`** specifier.

Allocatable arrays and character variables may be dynamically allocated and deallocated at execution time. An array must have been declared allocatable by specifying its dimensions using colons only. No array bounds are specified.

*Example:*

```
DIMENSION A(:), B(:,:)
```

In the above example, `A` is declared to be a one-dimensional allocatable array and `B` is declared to be a two-dimensional allocatable array.

A character variable must have been declared allocatable by specifying its size as (*).

*Example:*

```
CHARACTER C*(*)
```

For an allocatable array, the **`ALLOCATE`** statement establishes the lower and upper bounds of each array dimension and calculates the amount of memory required for the array.

For an allocatable character variable, the **`ALLOCATE`** statement establishes the number of characters in the character variable and thus the size of the character variable.

If there is no **`LOCATION=`** specifier, it then attempts to dynamically allocate memory for the array or character variable. The success of the allocation can be checked by using the **`STAT=`** specifier.

If there is a **LOCATION=** specifier, the expression in the specification is evaluated and that value is used as the address of the array or character variable. This permits the programmer to specify a substitute memory allocator or to map the array or character variable onto a fixed memory location.

*Example:*

```
DIMENSION A(:), B(:,:)
    .
    .
    .
ALLOCATE( A(N) )
ALLOCATE( B(0:4,5) )
```

More than one allocatable array or character variable may appear in an **ALLOCATE** statement, separated by commas.

*Example:*

```
DIMENSION A(:), B(:,:)
    .
    .
    .
ALLOCATE( A(N), B(0:4,5) )
```

If the allocation fails and the **STAT=** specifier was not used, an execution-time error occurs. If the **STAT=** specifier is used, the specified variable returns a zero value if the allocation succeeded, and a non-zero value if the allocation failed.

*Example:*

```
DIMENSION A(:), B(:,:)
    .
    .
    .
ALLOCATE( A(N), B(0:4,5), STAT=IALLOC )
IF( IALLOC .NE. 0 ) PRINT *, 'Allocation failure'
```

An attempt to allocate a previously allocated array or character variable results in an execution-time error. If the **LOCATION=** specifier was not used, the array or character variable must be deallocated first before it can be allocated a second time (see the **DEALLOCATE** statement).

An absolute memory location may be specified using the **LOCATION=** specifier.

*Example:*

```
      CHARACTER*1 SCREEN(:,:)
      N = 80*25
*$IFDEF __386__
      ALLOCATE( SCREEN(0:1,0:N-1), LOCATION='B8000'x )
*$ELSE
      ALLOCATE( SCREEN(0:1,0:N-1), LOCATION='B8000000'x )
*$ENDIF
      DO I = 0, N-1
          SCREEN(0,I) = '*'
      ENDDO
      END
```

The above example maps the array `SCREEN` onto the IBM PC colour monitor screen memory and then fills the screen with asterisks (16-bit real-mode only). The character is stored in `SCREEN(0,I)` and the

character attribute (unchanged in this example) is stored in `SCREEN(1,I)`. The column major ordering of arrays must be taken into consideration when mapping an array onto a fixed area of memory.

The following example is similar but uses an allocatable character variable.

*Example:*

```
      CHARACTER*(*) SCREEN
      INTEGER SCRSIZE, I
      PARAMETER ( SCRSIZE = 80*25*2 )
*$IFDEF __386__
      ALLOCATE( SCREEN*SCRSIZE, LOCATION='B8000'X )
*$ELSE
      ALLOCATE( SCREEN*SCRSIZE, LOCATION='B8000000'X )
*$ENDIF
      DO I = 1, SCRSIZE, 2
          SCREEN(I:I) = '*'
      ENDDO
      END
```

A user-defined memory allocator may be specified using the **`LOCATION=`** specifier.

*Example:*

```
      CHARACTER*1 BUFFER(:)
      N = 128
      ALLOCATE( BUFFER(0:N-1), LOCATION=MYALLOC(N) )
          .
          .
          .
      END
```

Perhaps a better way to check for a successful allocation, in this case, would be the following.

*Example:*

```
      CHARACTER*1 BUFFER(:)
      N = 128
      LOC = MYALLOC( N )
      IF( LOC .EQ. 0 ) STOP
      ALLOCATE( BUFFER(0:N-1), LOCATION=LOC )
          .
          .
          .
      END
```

For more information on arrays, see the chapter entitled "Arrays" on page 159.

# 2.5 Statement Label Assignment (ASSIGN) Statement

```
ASSIGN s TO i
```

***where:***

***s*** is a statement label

***i*** is an integer variable name

The statement label `s` is assigned to the integer variable `i.` The statement label must appear in the same program unit as the **ASSIGN** statement. The statement label must be that of an executable statement or a **FORMAT** statement.

After a statement label has been assigned to an integer variable, that variable may only be used in an assigned **GO TO** statement or as a format identifier in an input/output statement. The integer variable must not be used in any other way (e.g., in an arithmetic expression). It may, however, be redefined with another statement label using the **ASSIGN** statement or it may be assigned an integer value (e.g., in an arithmetic assignment statement).

*Example:*

```
          INTEGER RET
          X = 0.0
          ASSIGN 100 TO RET
          GO TO 3000
    100   X = X + 1
          ASSIGN 110 TO RET
          GO TO 3000
    110   X = X + 1
          .
          .
          .
    *     Print both X and its square root
    3000  Y = SQRT( X )
          PRINT *, X, Y
          GO TO RET
```

In the above example, we illustrate the use of the **ASSIGN** statement and the assigned **GO TO** statement to implement a "local subroutine" in a program unit. A sequence of often-used code can be "called" using the unconditional **GO TO** statement and "return" is accomplished using the assigned **GO TO** statement. Care must be exercised to properly assign the return label value.

*Example:*

```
          IF( FIRST )THEN
              ASSIGN 100 TO LFRMT
          ELSE
              ASSIGN 200 TO LFRMT
          END IF
          WRITE( UNIT=5, FMT=LFRMT) X, Y, Z
    100   FORMAT(1X,3F10.5)
    200   FORMAT(1X,3E15.7)
```

It should be noted that the **ASSIGN** statement does not assign the numeric value of the statement label to the variable.

*Example:*

```
          ASSIGN 100 TO LABEL2
          PRINT *, LABEL2
```

Try the above example; the value printed will not be 100.

Consider the following example.

*Example:*

```
    * Illegal use of a GOTO statement.
          LABEL2 = 123
          LABEL3 = LABEL2 + 10
          GO TO LABEL3
```

LABEL3 is assigned the integer value 133. The assigned **GO TO** statement, which follows it, is illegal and a run-time error will occur when it is executed.

Statement label values are quite different from integer values and the two should never be mixed. In the following example, the assignment statement is illegal since it involves an integer variable that was specified in an **ASSIGN** statement.

*Example:*

```
    * Illegal use of an ASSIGNed variable in an expression.
          ASSIGN 100 TO LABEL2
          LABEL3 = LABEL2 + 10
```

Note that if the assignment statement was preceded by

```
    LABEL2 = 100
```

the assignment statement would have been legal.

# 2.6 AT END Statement

```
AT END DO    [: block-label]

     or

AT END, stmt
```

***where:***

***stmt*** is an executable statement other than an **AT END** statement.

The **AT END** control statement is an extension of the **END=** option of the **READ** statement for sequential files. It allows a statement or a block of code following the **READ** statement to be executed when an end-of-file condition is encountered during the read. The **AT END** statement or block is by-passed if no end-of-file occurs. It is not valid to use this control statement with direct-access or internal files. It is not valid to use this statement when **END=** is also specified in the **READ** statement. The **AT END** statement or block must immediately follow the **READ** statement to which it applies.

*Example:*

```
READ( UNIT=1, FMT='(I5,F10.4)' ) I, X
AT END DO
    PRINT *, 'END-OF-FILE ENCOUNTERED ON UNIT 1'
    EOFSW = .TRUE.
END AT END
```

The second form of the **AT END** statement is illustrated below.

*Example:*

```
READ( UNIT=1, FMT='(F10.4)' ) X
AT END, EOFSW = .TRUE.
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.7 BACKSPACE Statement

```
BACKSPACE u
BACKSPACE (alist)
```

***where:***

***u*** is an external unit identifier.

***alist*** is a list of backspace specifiers separated by commas:

```
[UNIT =] u
IOSTAT = ios
ERR = s
```

Execution of a **BACKSPACE** statement causes the file connected to the specified unit to be positioned at the beginning of the preceding record. If the preceding record is an endfile record then the file is positioned at the beginning of the endfile record.

***Backspace Specifiers***

***[UNIT =] u***

`u` is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional **UNIT=** specifier is omitted then the specifier must be the first item in the list of specifiers.

***IOSTAT = ios***

is an *input/output status specifier.* The integer variable or integer array element `ios` is defined with zero if no error condition occurs or a positive integer value if an error condition occurs.

***ERR = s***

is an *error specifier* and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s.`

*Example:*

```
        LOOP
            READ( UNIT=8, END=100, FMT=200 ) RECORD
        ENDLOOP
100     BACKSPACE( UNIT=8 )
        WRITE( UNIT=8, FMT=200 ) NEWREC
```

In the previous example, we illustrate how one might append a record to the end of an existing file.

*Notes:*

1. The unit must be connected for sequential access.
2. If the file is positioned before the first record then the **BACKSPACE** statement has no effect.
3. It is illegal to backspace a file that does not exist.

The FORTRAN 77 standard specifies that it is illegal to backspace over records that were written using list-directed formatting; Open Watcom FORTRAN 77 allows it.

If the file has been opened with access `'APPEND'`, which is a form of sequential access in which the file is positioned at the endfile record, then the **BACKSPACE** statement cannot be used.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.8 BLOCK DATA Statement

```
BLOCK DATA [sub]
```

***where:***

***sub*** is an optional symbolic name of the block data subprogram and must not be the name of an external procedure, main program, common block, other block data subprogram, or any local name in the block data subprogram.

The **BLOCK DATA** statement is used to define the start of a block data subprogram. A block data subprogram is used to provide initial values for variables and array elements in *named common blocks.*

The only statements which are allowed to appear in a block data subprogram are:

1. **IMPLICIT**
2. **PARAMETER**
3. **DIMENSION**
4. **COMMON**
5. **SAVE**
6. **EQUIVALENCE**
7. **DATA**
8. **STRUCTURE, END STRUCTURE**
9. **UNION, END UNION**
10. **MAP, END MAP**
11. **RECORD**
12. **END**
13. type statements

*Example:*

```
BLOCK DATA INITCB
DIMENSION A(10), B(10)
COMMON /CB/ A, B
DATA A/10*1.0/, B/10*2.0/
END
```

In the above example, the arrays `A` and `B` in the named common block `CB` are initialized.

*Notes:*

1. More than one named common block may appear in a block data subprogram.
2. All entities of the named common block(s) must be specified.
3. Not all entities need be given initial values.
4. Only entities that appear in (or are associated, through the **EQUIVALENCE** statement, with entries in) a named common block may be given initial values.
5. Only one unnamed block data subprogram may occur in an executable program.

6. A named block data subprogram may occur only once in an executable program.

# 2.9 CALL Statement

```
CALL sub [( [a [, a] ... ] )]
```

***where:***

***sub*** is a symbolic name of a subroutine and must not be the name of a main program, function, common block, or block data subprogram. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 permits the calling of functions.

***a*** is an actual argument.

The **CALL** statement is used to invoke the execution of a subroutine subprogram or function.

*Example:*

```
X = 1.0
Y = 1.1
Z = 1.2
CALL QUAD( X, Y, Z )
     .
     .
     .
END

SUBROUTINE QUAD( ARGA, ARGB, ARGC )
REAL ARGA, ARGB, ARGC
PRINT *, 2.0*ARGA**2 + 4.0*ARGB + ARGC
END
```

In the above example, the variables `X, Y` and `Z` are passed to the subroutine `QUAD.` This subroutine computes an expression and prints the result.

*Notes:*

1. The parameters in the **CALL** statement are called *actual arguments.*
2. The parameters in the **SUBROUTINE** statement are called *dummy arguments.*
3. The actual arguments in a subroutine or function reference must agree in order, number and type with the corresponding dummy arguments.
4. An actual argument may be an expression, array name, intrinsic function name, external procedure name (i.e., a subroutine or function name), a dummy procedure name (i.e., one that was an argument to the calling subroutine or function), or an *alternate return specifier* (subroutines only). An alternate return specifier takes the form `*s,` where `s` is the statement label of an executable statement that appears in the same program unit as the **CALL** statement. An expression may not be a character expression involving the concatenation of an operand whose length specification is `(*)` unless the operand is the symbolic name of a constant.
5. Actual arguments are associated with dummy arguments by passing the address of the actual arguments.

It is important to note that versions of FORTRAN compilers that implement the previous FORTRAN language standard may have associated arguments by passing the value of the actual argument and assigning it to the dummy argument and then updating the actual argument upon return from the subprogram (this is called "value-result" argument handling). The FORTRAN 77 language standard prohibits this technique for handling argument association.

The following example illustrates the importance of this rule.

*Example:*

```
I=1
CALL ASSOC( I, I )
END
SUBROUTINE ASSOC( M, N )
M = M + 1
PRINT *, M, N
END
```

In the above example, `M` and `N` refer to the same variable; they are both associated to `I` in the calling subprogram. The value 2 will be printed twice.

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

# 2.10 CASE Statement

```
CASE cl
```

***where:***

***cl*** is a list, enclosed in parentheses, of one or more *cases* separated by commas, or the **DEFAULT** keyword. A *case* is either

**(a)** a single integer, logical or character constant expression or

**(b)** an integer, logical or character constant expression followed by a colon followed by another expression or the same type. This form of a case defines a range of values consisting of all integers or characters greater than or equal to the value of the expression preceding the colon and less than or equal to the value of the expression following the colon.

The **CASE** statement is used in conjunction with the **SELECT** statement. The **CASE** statement marks the start of a new **CASE** block which is a series of zero or more statements ending in another **CASE** statement, a **CASE DEFAULT** statement, or an **END SELECT** statement.

A particular case value or range of values must not be contained in more than one **CASE** block.

The **CASE DEFAULT** statement is used to indicate a block of statements that are to be executed when no other case is selected.

*Example:*

```
SELECT CASE ( CH )
CASE ( 'a' : 'z' )
    PRINT *, 'Lower case letter'
CASE ( 'A' : 'Z' )
    PRINT *, 'Upper case letter'
CASE ( '0' : '9' )
    PRINT *, 'Digit'
CASE DEFAULT
    PRINT *, 'Special character'
END SELECT
```

In order to retain compatibility with earlier versions of WATCOM FORTRAN 77 compilers, the **OTHERWISE** statement may be used in place of the **CASE DEFAULT** statement.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.11 CHARACTER Statement

The **CHARACTER** statement is a type declaration statement and can be used to declare a name to be of type character. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type character, it cannot appear in another type declaration statement.

There are various forms of the **CHARACTER** statement. The following sections describe them.

## 2.11.1 Standard CHARACTER Statement

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

***where:***

***name*** is one of the following forms:

```
v[*len]

a(d)[*len]

a[*len](d)
```

***v*** is a variable name, symbolic name of a constant, function name or dummy procedure name.

***a*** is an array name.

***(d)*** is that part of the array declarator defining the dimensions of the array.

***len*** is called the *length specification* and is the length (number of characters) of a character variable, character array element, symbolic character constant or character function. It has one of the following forms:

- ***(1)*** An unsigned positive integer constant.
- ***(2)*** A positive integer constant expression enclosed in parentheses.
- ***(3)*** An asterisk in parentheses `(*)`.

The length specification immediately following the word `CHARACTER` is the length specification for each entity in the statement not having its own length specification. If omitted, the default is 1. An entity with its own length specification overrides the default length specification or the length specification immediately following the word `CHARACTER`. Note that for an array the length specification applies to *each* element of the array.

*Example:*

```
          DIMENSION C(-5:5)
          CHARACTER A, B*10(10), C*20
          CHARACTER*7 X, Y, Z*4
```

The `(*)` length specification is only allowed for external functions, dummy arguments or symbolic character constants. If a dummy argument has a length specification of `(*)`, it assumes the length of the corresponding actual argument. If the actual argument is an array name, the length assumed by the dummy argument is the length of an array element of the actual array.

*Example:*

```
          SUBROUTINE OUTCHR( STR )
          CHARACTER STR*(*)
          PRINT *, STR
          END
```

In this example, `STR` is a character variable whose length is the length of the actual argument. Thus `OUTCHR` can be called with a character entity of any length.

If an external function has a length specification of `(*)` declared in a function subprogram, the function name must appear as the name of a function in a **`FUNCTION`** or **`ENTRY`** statement in the same subprogram. When the function is called, the function assumes the length specified in the program unit that called it. In the following example, when `F` is called its length is assumed to be 10.

*Example:*

```
          CHARACTER*(10) F
               .
               .
               .
          PRINT *, F()
               .
               .
               .
          END

          CHARACTER*(*) FUNCTION F
          F = 'HELLO'
          END
```

The following example is illegal since `F` does not appear in a **`FUNCTION`** or **`ENTRY`** statement.

*Example:*

```
    * Illegal definition of function F.
          CHARACTER*(*) F
               .
               .
               .
          PRINT *, F()
               .
               .
               .
          END
```

The length specified for a character function in the program unit that referenced it must agree with the length specified in the subprogram that defines the character function. Note that there is always agreement if the function is defined to have a length specification of `(*)`.

If a symbolic name is of type character and has a length specification of `(*),` it assumes the length of the corresponding character constant expression in the **PARAMETER** statement.

The length specification of a character statement function or statement function dummy argument must not be `(*).`

## 2.11.2 Extended CHARACTER Statement: Data Initialization

```
CHARACTER[*len[,]] name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***len*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **CHARACTER** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
CHARACTER*5 A/'AAAAA'/, B*3(10)/10*'111'/
```

In the previous example, `A` is initialized with the character constant `'AAAAA'` and each element of the array `B` is initialized with the character constant `'111'.`

# 2.12 CLOSE Statement

```
CLOSE (cllist)
```

***where:***

***cllist*** is a list of close specifiers separated by commas:

```
[UNIT =] u
IOSTAT = ios
ERR = s
STATUS = sta
```

A **CLOSE** statement is used to terminate the connection of a file to the specified unit.

***Close Specifiers***

***[UNIT =] u***

`u` is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional **UNIT=** specifier is omitted then the specifier must be the first item in the list of specifiers.

***IOSTAT = ios***

is an input/output status specifier. The integer variable or integer array element `ios` is defined with zero if no error condition occurs or a positive integer value if an error condition occurs.

***ERR = s***

is an error specifier and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s.`

***STATUS = sta***

is a status specifier and `sta` is a character expression whose value when trailing blanks are removed evaluates to one of `'KEEP'` or `'DELETE'.`

***KEEP*** `'KEEP'` may not be specified for a file whose status is `'SCRATCH'` (see description of the **OPEN** statement). If the file exists, it will exist after execution of the **CLOSE** statement. If the file does not exist, it will not exist after execution of the **CLOSE** statement. If not specified, `'KEEP'` is assumed, unless the file status is `'SCRATCH'` in which case `'DELETE'` is assumed.

***DELETE*** If `'DELETE'` is specified, the file will not exist after execution of the **CLOSE** statement.

*Example:*

```
          LOOP
              READ( UNIT=8, END=100, FMT=200 ) RECORD
          ENDLOOP
100       CLOSE( UNIT=8 )
```

In the previous example, we illustrate how one might process the records in a file and then terminate the connection of the file to unit 8 using the **CLOSE** statement.

*Notes:*

1. Execution of a **CLOSE** statement specifying a unit that is not connected to a file or a unit that is connected to a file that does not exist has no effect.

2. It is possible to connect the unit to another file after a **CLOSE** statement has been executed.

3. It is possible to connect the unit to the same file after a **CLOSE** statement has been executed, provided that the file still exists.

4. It is possible to connect the file to another unit after a **CLOSE** statement has been executed, provided that the file still exists.

5. At the termination of execution of the program, for whatever the reason of termination, any units that are connected are closed. Each unit is closed with status `'KEEP'` unless the file status was `'SCRATCH',` in which case the unit is closed with status `'DELETE'.` The effect is the same as if a **CLOSE** statement is executed without a **STATUS=** specifier.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.13 COMMON Statement

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

***where:***

***cb*** is a common block name.

***nlist*** is a list of names each separated by a comma.

If `cb` is omitted, the *blank common block* is assumed. If specified, `cb` is called a *named common block.* The names appearing in `nlist` can be variable names, array names, and array declarators. Dummy arguments are not allowed in `nlist.`

The **COMMON** statement allows sharing of blocks of storage between subprograms. Each name appearing in the `nlist` following a common block name `cb` is declared to belong to that common block. A variable or an array name can belong to only one common block. A common block name can occur more than once in the same **COMMON** statement as well as in more than one **COMMON** statement. Lists following successive appearances of the same common block name in **COMMON** statements are considered a continuation of the list of names belonging to the common block. A variable or an array can appear in a **COMMON** statement only once.

Common blocks are defined as follows. A common block is one consecutive block of storage. It consists of all the storage sequences of all the entities specified in all the lists declared to belong to that common block. The order in which each entity appears in a common block is defined by the order in which they appear in the lists. Storage sequences associated to a common block through the **EQUIVALENCE** statement are considered to belong to that common block. In this way a common block may only be extended beyond the last storage unit. The size of a common block is the sum of all the storage sequences of all the names belonging to that common block plus any storage sequence which extends the common block through equivalence association.

An **EQUIVALENCE** statement must not cause storage sequences of two different common blocks to become associated nor should they extend the common block by adding storage units preceding the first storage unit of the common block.

*Example:*

```
DIMENSION A(5)
COMMON /COMBLK/ A,B(10),C
```

In this example, the common block `COMBLK` contains the array `A` followed by the array `B` and finally the variable `C.`

*Example:*

```
      REAL A,B,C,D
      DIMENSION D(5)
      EQUIVALENCE (B,D)
      COMMON A,B,C
```

In this example, `A, B, C,` and `D` belong to the blank common block; `A, B,` and `C` have been explicitly defined to be in the blank common block whereas `D` has been equivalenced to a variable in common, namely `B.` Also note that the **EQUIVALENCE** statement has caused the extension of the common block beyond its last storage unit. In this example, array `D` has extended the common block by 3 storage units.

*Example:*

```
    * Illegal definition of a common block.
      DIMENSION A(5)
      EQUIVALENCE (A(2),B)
      COMMON /XYZ/ B
```

This example demonstrates an illegal use of the **COMMON** statement. `B` is in the named common block `XYZ` since it appeared in a **COMMON** statement. `A` is in the common block `XYZ` since it was equivalenced to `B.` However, `A` illegally extends the common block by adding 1 storage unit before the first storage unit of the common block.

The following outlines the differences between a blank common block and a named common block.

**(1)** All named common blocks with the same name in an executable program must be the same size. Blank common blocks do not have to be the same size.

**(2)** Entities in named common blocks can be initialized by using **DATA** statements in block data subprograms; entities in blank common blocks cannot.

**(3)** Entities in named common blocks can become undefined after the execution of a **RETURN** or **END** statement; entities in blank common blocks cannot. This situation can arise when all subprograms which refer to the named common block become inactive. A typical case occurs when program overlays are used. If the named common block is placed in an overlay, then the entities in the named common block will become undefined when the overlay is replaced by another. Of course, if the named common block is referenced in the main program then this could never happen. The main program and any named common blocks referenced in the main program remain memory-resident until the application terminates.

The **SAVE** statement should be used if entities in named common blocks must not become undefined.

The FORTRAN 77 standard specifies that a common block cannot contain both numeric and character data; Open Watcom FORTRAN 77 allows common blocks to contain both numeric and character data.

The FORTRAN 77 standard specifies that a named common block must be initialized in a block data subprogram. Open Watcom FORTRAN 77 permits the initialization of named common blocks in other subprograms.

# 2.14 COMPLEX Statement

The **COMPLEX** statement is a type declaration statement and can be used to declare a name to be of type complex. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type complex, it cannot appear in another type declaration statement.

There are various forms of the **COMPLEX** statement. The following sections describe them.

## 2.14.1 Standard COMPLEX Statement

```
COMPLEX name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

This form is the standard form of the **COMPLEX** statement.

*Example:*

```
DIMENSION C(-5:5)
COMPLEX A, B(10), C
```

In the previous example, `A` is defined to be a variable of type complex and `B` and `C` are defined to be arrays of type complex.

## 2.14.2 Extended COMPLEX Statement: Length Specification

```
COMPLEX[*len[,]] name [,name] ...
```

***where:***

***name*** is one of the following forms:

```
v[*len]

a[*len](d)

a(d)[*len]
```

***v*** is a variable name, array name, symbolic name of a constant, function name or dummy procedure name.

***a*** is an array name.

***(d)*** is that part of the array declarator defining the dimensions of the array.

***len*** is called the *length specification* and is an unsigned positive integer constant or an integer constant expression enclosed in parentheses whose value is 8 or 16.

This form of the **COMPLEX** statement is a Open Watcom FORTRAN 77 extension to the FORTRAN 77 language. The length specification specifies the number of bytes of storage that will be allocated for the name appearing in the **COMPLEX** statement. The default length specification is 8. A length specification of 16 specifies that the data type of the name appearing in the **COMPLEX** statement is to be double precision complex.

The length specification immediately following the word `COMPLEX` is the length specification for each entity in the statement not having its own length specification. If a length specification is not specified the default length specification is used. An entity with its own specification overrides the default length specification or the length specification immediately following the word `COMPLEX`. Note that for an array the length specification applies to *each* element of the array.

*Example:*

```
DIMENSION C(-5:5)
COMPLEX A, B*16(10), C*16
COMPLEX*16 X
```

In the previous example, `X` is declared to be a variable of type double precision complex, `A` is declared to be a variable of type complex and `B` and `C` are declared to be arrays of type double precision complex.

## 2.14.3 Extended COMPLEX Statement: Data Initialization

```
COMPLEX[*len[,]] name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***len*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **COMPLEX** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
COMPLEX A/(.4,-.3)/, B(10)/10*(0,1)/
```

In the previous example, `A` is initialized with the complex constant `(.4,-.3)` and each element of the array `B` is initialized with the complex constant `(0,1).`

# 2.15 CONTINUE Statement

```
CONTINUE
```

Execution of a **CONTINUE** statement has no effect. This statement is often used in conjunction with **DO** statements. It is usually identified with a label. It often provides a convenient reference for statements which have the ability to transfer control of execution.

*Example:*

```
          DO 10 X = -5.1, 12.8, 0.125
              .
              .
              .
10        CONTINUE

          IF( A .LT. B ) GO TO 20
          IF( A .GT. C ) GO TO 20
              .
              .
              .
20        CONTINUE
```

# 2.16 CYCLE Statement

```
CYCLE   [: block-label]
```

The **CYCLE** statement may be used to cause a transfer of control from within a loop to the terminal statement of a corresponding **DO**, **DO WHILE**, **WHILE** or **LOOP** statement. If `block-label` is present then control is transferred to the terminal statement of the block identified by that block label. The **CYCLE** statement is an extension to the FORTRAN 77 language.

*Example:*

```
LOOP
    WRITE( UNIT=*, FMT='(A)' ) 'Enter a number'
    READ( UNIT=*, FMT='(F10.4)', IOSTAT=IOS ) X
    IF( IOS .NE. 0 ) CYCLE
    IF( X .LT. 0 ) EXIT
    PRINT *, X, SQRT( X )
END LOOP
END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.17 DATA Statement

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

***where:***

***nlist*** is a list of variable names, array element names, substring names and implied-DO lists.

***clist*** is a list of the form:

`a [,a] ...`

***a*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

The items of `nlist` are initialized with the values specified in `clist` in the following manner. The first item in `nlist` is assigned the value of the first item in `clist,` the second item in `nlist` is assigned the value of the second item in `clist,` etc. In this way all items of `nlist` are initialized.

The number of items in `nlist` must equal the number of items in `clist` so that a one-to-one correspondence exists between the two lists. If an array without a subscript list appears in `nlist` there must be an element in `clist` for each element of the array.

If the type of an entity in `nlist` is character or logical then the type of its corresponding item in `clist` must also be character or logical respectively. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 permits an item of type character to be initialized with integer data.

An item of type character is initialized using the rules of assignment. If the length of the item in `nlist` is greater than the length of the corresponding character constant in `clist,` the rightmost remaining characters in the item are initialized with blanks. If the length of the item in `nlist` is less than the length of the character constant in `clist,` the character constant is truncated to the length of the item in `nlist.` Note that initializing a character entity causes all of the characters in the entity to become defined and that each character constant defines exactly one character variable, array element or substring.

If the type of an entity in `nlist` is integer, real, double precision or complex then the corresponding item in `clist` can be one of integer, real, double precision or complex. If necessary the constant in `clist` is converted to the type of the item in `nlist` according to the rules of arithmetic conversion (see the chapter entitled "Assignment Statements" on page 187).

A variable, array element or substring can only be initialized once. If two entities are associated (for example equivalenced), only one of the items can be initialized.

*Example:*

```
CHARACTER*30 MSG
LOGICAL TRUE
REAL X, Y(10)
DATA X/1.0/, TRUE/.TRUE./, MSG/'ENTER DATA'/
DATA Y/10*5/
```

An *implied-DO list* in a **DATA** statement has the following form:

```
( dlist, i = m1, m2[, m3] )
```

***where:***

***dlist*** is a list of array element names and implied-DO lists.

***i*** is the name of an integer variable called the implied-DO-variable.

***m1,m2,m3*** are each integer constant expressions. The expressions may contain implied-DO-variables of other implied-DO lists that have this implied-DO list in their ranges.

The range of the implied-DO list is the list `dlist.` An iteration count and the value of the implied-DO-variable are computed from `m1,m2` and `m3` in the same way as for a DO-loop except that the iteration count must be positive. An implied-DO-variable does not affect the definition of a variable by the same name in the same program unit. An implied-DO list is processed as follows. Each item in the implied-DO list is processed once for each iteration with the appropriate substitution of values for any occurrence of the implied-DO-variable. The following example initializes the upper right triangle of the array A.

*Example:*

```
DIMENSION A(5,5)
DATA ((A(I,J),J=1,I),I=1,5)/15*0/
```

Dummy arguments, functions, and entities in blank common are not allowed in `nlist.` Entities in a named common block can be initialized only within a *block data subprogram.*

The following extensions to data initialization are supported by Open Watcom FORTRAN 77.

1. Character constants can initialize a variable of any type. If the item in `nlist` is of numeric type and is being initialized with character data, the size of the item in `nlist` is the maximum number of characters that can be stored in the space allocated for that item. The rules for initializing such items, are the same as for items of type character. See the chapter entitled "Names, Data Types and Constants" on page 151 for the number of bytes required for a particular data type.

   *Example:*

   ```
   INTEGER I,J
   DATA I/'AA'/,J/'123456'/
   ```

   In the previous example, `I` and `J` each occupy 4 character storage units. `I` will be initialized with the characters `AA` followed by 2 blank characters. `J` will be initialized with the characters `1234.` Note the the character constant initializing `J` is truncated on the right to the number of character storage units occupied by `J.`

2. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 permits an item of type character to be initialized with integer data.

   *Example:*

```
          CHARACTER C, D
          DATA C/65/, D/66/
          END
```

3. Open Watcom FORTRAN 77 allows data initialization using hollerith constants. Initializing items using hollerith constants behaves in the same way as initializing items using character constants. Note that hollerith data can initialize entities of any type. See the chapter entitled "Names, Data Types and Constants" on page 151 for a description of hollerith constants.

4. Open Watcom FORTRAN 77 allows data initialization using hexadecimal or octal constants. Hexadecimal or octal constants can be used to initialize memory with any binary pattern.

   Items are initialized with hexadecimal constants in the following way. Two hexadecimal digits are required to initialize one byte of storage. If the number of characters in the hexadecimal constant is less than 2 times the number of bytes of storage allocated for the entity being initialized, the entity is padded on the left with zeroes. If the number of characters in the hexadecimal constant is greater than 2 times the number of bytes of storage allocated for the entity being initialized, the constant is truncated on the left to the size (in bytes) of the entity being initialized.

   Items are initialized with octal constants in the following way. Each octal digit initializes three bits of storage. If the number of digits in the octal constant times 3 is less than the number of bits of storage allocated for the entity being initialized, the entity is padded on the left with zero bits. If the number of digits in the octal constant times 3 is greater than the number of bits of storage allocated for the entity being initialized, bits are truncated on the left to the size (in bits) of the entity being initialized.

   Note that hexadecimal or octal data can initialize entities of any type. See the chapter entitled "Names, Data Types and Constants" on page 151 for a description of hexadecimal and octal constants.

   *Example:*

```
          DOUBLE PRECISION DPREC
          COMPLEX CMPLX
    * Initialize an integer variable with the value 5
          DATA I/Z05/
    * Initialize a real variable with the value 5.0
          DATA X/Z41500000/
    * Initialize a double precision variable
    *   with the value 5D0
          DATA DPREC/Z4150000000000000/
    * Initialize a complex variable
    *   with the value (5.0,5.0)
          DATA CMPLX/Z4150000041500000/
              .
              .
              .
          END
```

Caution should be used when initializing items with hexadecimal constants, in particular those whose type is real or double precision, since the data they represent depends on the computer being used. In the previous example, the hexadecimal constant used to initialize the variable `X,` represents the number 5.0 on a computer with an IBM 370 architecture. The number 5.0 will have a different floating-point representation on other computers.

# 2.18 DEALLOCATE Statement

```
DEALLOCATE (arraylist [, STAT = ierr])
```

***where:***

***arraylist*** is a list of allocatable array names separated by commas.

***ierr*** is an integer variable that returns the status of the attempted deallocation.

Allocatable arrays may be dynamically allocated and deallocated at execution time. An array must have been declared allocatable by specifying its dimensions using colons only. No array bounds are specified.

*Example:*

```
DIMENSION A(:), B(:,:)
```

In the above example, `A` is declared to be a one-dimensional allocatable array and `B` is declared to be a two-dimensional allocatable array.

The **DEALLOCATE** statement frees up any memory allocated for the specified array(s). It then disassociates the specified array(s) from the memory to which it was associated. The deallocation does not necessarily succeed. For example, an attempt to deallocate an array that was not previously allocated will cause an error.

*Example:*

```
DIMENSION A(:), B(:,:)
   .
   .
   .
ALLOCATE( A(N), B(0:4,5) )
   .
   .
   .
DEALLOCATE( A )
```

More than one allocatable array may appear in an **DEALLOCATE** statement, separated by commas.

*Example:*

```
DIMENSION A(:), B(:,:)
   .
   .
   .
ALLOCATE( A(N), B(0:4,5) )
   .
   .
   .
DEALLOCATE( A, B )
```

If the deallocation fails and the **STAT=** specifier was not used, an execution-time error occurs. If the **STAT=** specifier is used, the specified variable returns a zero value if the deallocation succeeded, and a non-zero value if the deallocation failed.

*Example:*

```
DIMENSION A(:), B(:,:)
  .
  .
  .
ALLOCATE( A(N), B(0:4,5), STAT=IALLOC )
IF( IALLOC .NE. 0 ) PRINT *, 'Allocation failure'
  .
  .
  .
DEALLOCATE( A, B, STAT=IFREE )
IF( IFREE .NE. 0 ) PRINT *, 'Deallocation failure'
```

An attempt to deallocate an unallocated array results in an execution-time error. The array must be allocated first (see the **ALLOCATE** statement).

An array that was allocated using the **LOCATION=** specifier need not be deallocated.

For more information on arrays, see the chapter entitled "Arrays" on page 159.

# 2.19 DIMENSION Statement

```
DIMENSION a(d) [,a(d)] ...
```

***where:***

***a*** is the name of the array.

***d*** defines the dimension of the array and the range of its subscripts. See the chapter entitled "Arrays" on page 159 for more information on dimensioning arrays.

Each name `a` appearing in a **`DIMENSION`** statement defines `a` to be an array in the program unit containing the **`DIMENSION`** statement. A name can only be dimensioned once in a program unit. Note that a name can also be dimensioned in a **`COMMON`** statement and type declaration statements.

*Example:*

```
DIMENSION A(10), B(-5:5), C(I,J), D(4,*)
```

In this example `A` is a 1-dimensional array containing 10 elements, each element referenced as `A(1)`, `A(2)`, **`'...'`**, `A(9)`, `A(10)`. `B` is a 1-dimensional array containing 11 elements, each element referenced as `B(-5)`, `B(-4)`, **`'...'`**, `B(4)`, `B(5)`. `C` is a 2-dimensional array containing `I` rows and `J` columns. `C`, `I`, and `J` must be *dummy arguments* or belong to a common block. `D` is a 2-dimensional array containing 4 rows. The `*` in the last dimension indicates that `D` is an *assumed size array.* `D` must be a *dummy argument.* The number of columns is determined from the number of elements of the *actual argument.* For example, if the actual argument contains 8 elements then `D` would contain 2 columns (i.e., 8 elements / 4 rows).

For more information on dimensioning arrays refer to the chapter entitled "Arrays" on page 159. See also the description of the **`ALLOCATE`** and **`DEALLOCATE`** statements for information on dynamically allocatable arrays.

# 2.20 DO Statement

Two forms of the **DO** statement are presented. The second form is a Open Watcom FORTRAN 77 extension to the FORTRAN 77 language.

## 2.20.1 Standard DO Statement

```
DO s [,] i = e1, e2 [, e3]
```

***where:***

| | |
|---|---|
| ***s*** | is the statement label of an executable statement, called the *terminal statement,* which follows the **DO** statement in the same program unit. |
| ***i*** | is an integer, real, or double precision variable, called the DO-variable. |
| ***e1, e2, e3*** | are each an integer, real, or double precision expression. |

## 2.20.2 Extended DO Statement

```
DO [s[,]] i = e1, e2 [, e3]    [: block-label]
```

***where:***

| | |
|---|---|
| ***s*** | is an optional statement label of an executable statement, called the *terminal statement,* which follows the **DO** statement in the same program unit. |
| ***i*** | is an integer, real, or double precision variable, called the DO-variable. |
| ***e1, e2, e3*** | are each an integer, real, or double precision expression. |
| ***block-label*** | is an optional block label. |

This form of the **DO** statement is an extension to the FORTRAN 77 language. If no statement label is present then the terminal statement of the DO-loop must be an **END DO** statement. In all other respects, the rules are the same as those given for the standard **DO** statement.

## 2.20.3 Description of DO Statement

The range of a DO-loop consists of all of the executable statements that appear following the **DO** statement that specifies the DO-loop, up to and including the terminal statement of the DO-loop. Only certain statements can be the terminal statement of a DO-loop. See the section entitled "Classifying Statements" on page 9 at the beginning of this chapter for a list of these statements.

Transfer of control into the range of a DO-loop from outside the range is not permitted.

A DO-loop may be executed 0 or more times. The following sequence occurs when a **DO** statement is encountered.

**(i)** An *initial* value, `m1,` is calculated by evaluating expression `e1.` A *terminal* value, `m2,` is calculated by evaluating expression `e2.` An *incrementation* value, `m3,` is calculated by evaluating expression `e3` if it is present; otherwise `m3` has the value one. If `e3` is specified, `m3` must not be zero. The type of `m1, m2, and m3` is determined from the DO-variable and any conversions of type are done as required.

**(ii)** The DO-variable is defined with the initial value `m1.`

**(iii)** The iteration count (i.e., the maximum number of times that the DO-loop will be executed) is calculated as follows:

```
MAX( INT((m2 - m1 + m3)/m3), 0 )
```

The iteration count will be zero whenever:

```
m1 > m2 and m3 > 0, or
m1 < m2 and m3 < 0.
```

The number of times that the DO-loop is executed may be reduced if control is transferred outside the range of the DO-loop, or if a **RETURN** or **STOP** statement is executed.

The steps involved in each iteration of the DO-loop are as follows:

**(i)** Check the iteration count. If it is not zero then start execution of the first executable statement of the DO-loop. If the count is zero then iteration of the DO-loop is complete.

**(ii)** Execute statements until the terminal statement is encountered. During this time, the DO-variable may not be redefined.

**(iii)** Execute the terminal statement. Unless execution of the terminal statement causes a transfer of control, proceed with the next step which is "incrementation" processing.

**(iv)** The DO-variable is incremented by the value `m3.` The iteration count is decremented by one. Go back to step (i).

*Example:*

```
          DO 10 I = -5, 5
              PRINT *, I, I*I
   10     CONTINUE
```

In this example, the initial value is -5, the terminal value is 5, and the incrementation value is 1 (the default). The DO-variable is `I.` The DO-loop is executed

```
MAX( INT((5 - (-5) + 1)/1), 0 )
```

or 11 times. The successive values of `I,` inside the range of the DO-loop, are -5, -4, -3, ..., 0, 1, ..., 4, 5. When the DO-loop is terminated, the value of `I` will be 6. It should be noted that when a DO-loop variable is of type real, the iteration count may be one less than expected. Because of rounding errors, the value of `m2 - m1 + m3` may be slightly less than the exact value and when the INT function is applied, the resulting iteration count is one less than expected.

*Example:*

```
          DO 10 X = -5, 6, 2
              PRINT *, X, X*X
    10    CONTINUE
```

In this example, the terminal value has been changed to 6 and the incrementation value has been changed to 2. The DO-variable is `X`, a real variable. Thus the values of `e1`, `e2` and `e3` are converted to type real. The DO-loop is executed

```
        MAX( INT((6 - (-5) + 2)/2), 0 )
        MAX( INT(    13       /  2), 0 )
```

or 6 times. The successive values of `X`, inside the range of the DO-loop, are -5.0, -3.0, -1.0, 1.0, 3.0, 5.0. When the DO-loop is terminated, the value of `X` will be 7.0.

DO-loops may be nested, that is, another DO-loop may be contained within the range of the outer DO-loop. More than one DO-loop may have the same terminal statement.

*Example:*

```
          DO 10 I = -5, 5
          DO 10 J = -2, 3
    10        ARRAY( I, J ) = 0.0
```

This is equivalent to the following example.

*Example:*

```
          DO 10 I = -5, 5
              DO 20 J = -2, 3
                  ARRAY( I, J ) = 0.0
    20        CONTINUE
    10    CONTINUE
```

If a **DO** statement appears within the range of a DO-loop, its range must be entirely contained within the range of the outer DO-loop.

*Example:*

```
    * Illegal use of nested DO-loops.
          DO 20 I = -5, 5
              DO 10 J = -2, 3
                  ARRAY( I, J ) = 0.0
    20        CONTINUE
    10    CONTINUE
```

The above example is illegal since the terminal statement of the first DO-loop precedes that of the second DO-loop.

Similarly, the range of a DO-loop that appears within the range of an IF-block, ELSE IF-block, or ELSE-block must be entirely contained within that IF-block, ELSE IF-block, or ELSE-block, respectively. This rule applies to all Open Watcom FORTRAN 77 structured block extensions.

*Example:*

```
    * Illegal nesting of a DO-loop and an IF-block.
          IF( A .LT. B )THEN
              DO 10 I = 1, 5
                  PRINT *, 'Iteration number', I
          END IF
                  VECTOR( I ) = I
    10        CONTINUE
```

The above example is illegal since the range of the IF-block must terminate after the range of the DO-loop. Note how statement indentation helps to illustrate the problem with this example.

It is also illegal to attempt to transfer control into the range of a DO-loop. The following example illustrates this error.

*Example:*

```
    * Illegal transfer into the range of a DO-loop.
          GO TO 20
          .
          .
          .
          DO 10, I = 100, 0, -1
              PRINT *, 'Counting down from 100 to 0', I
    20        PRINT *, I, SQRT( FLOAT(I) )
    10    CONTINUE
```

The following example shows a more subtle form of this error.

*Example:*

```
    * Illegal transfer into the range of a DO-loop.
          DO 10 I = 1, 10
    *         Skip row 5 of 10x10 matrix
              IF( I .EQ. 5 )GO TO 10
              DO 10 J = 1, 10
                  A( I, J ) = 0.0
    10    CONTINUE
```

Since the **CONTINUE** statement is included in the range of the inner DO-loop, an error message is issued.

The following example illustrates the Open Watcom FORTRAN 77 structured **DO** statement.

*Example:*

```
          DO I = -5, 5
              DO J = -2, 3
                  ARRAY( I, J ) = 0.0
              END DO
          END DO
```

In keeping with more modern programming practices, this feature allows the programmer to write DO-loops without resorting to the use of statement labels. A well-chosen indentation style further enhances the readability of the program.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.21 DOUBLE COMPLEX Statement

The **DOUBLE COMPLEX** statement is a type declaration statement and can be used to declare a name to be of type double complex. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type double complex, it cannot appear in another type declaration statement.

There are various forms of the **DOUBLE COMPLEX** statement. The following sections describe them.

## 2.21.1 Simple DOUBLE COMPLEX Statement

```
DOUBLE COMPLEX name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

The **DOUBLE COMPLEX** statement is an extension to the FORTRAN 77 language.

*Example:*

```
DIMENSION C(-5:5)
DOUBLE COMPLEX A, B(10), C
```

In the previous example, `A` is defined to be a variable of type double complex and `B` and `C` are defined to be arrays of type double complex.

## 2.21.2 DOUBLE COMPLEX Statement: Data Initialization

```
DOUBLE COMPLEX name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **DOUBLE COMPLEX** statement is also an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
DOUBLE COMPLEX A/(4D4,5.1D4)/, B(10)/10*(5D1,3.1D1)/
```

In the previous example, `A` is initialized with the double precision complex constant `(4D4,5.1D4)` and each element of the array `B` is initialized with the double precision complex constant `(5D1,3.1D1).`

# 2.22 DOUBLE PRECISION Statement

The **DOUBLE PRECISION** statement is a type declaration statement and can be used to declare a name to be of type double precision. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type double precision, it cannot appear in another type declaration statement.

There are various forms of the **DOUBLE PRECISION** statement. The following sections describe them.

## 2.22.1 Standard DOUBLE PRECISION Statement

```
DOUBLE PRECISION name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

This form is the standard form of the **DOUBLE PRECISION** statement.

*Example:*

```
DIMENSION C(-5:5)
DOUBLE PRECISION A, B(10), C
```

In the previous example, `A` is defined to be a variable of type double precision and `B` and `C` are defined to be arrays of type double precision.

## 2.22.2 Extended DOUBLE PRECISION Statement: Data Initialization

```
DOUBLE PRECISION name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **DOUBLE PRECISION** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
DOUBLE PRECISION A/4D4/, B(10)/10*5D1/
```

In the previous example, `A` is initialized with the double precision constant `4D4` and each element of the array `B` is initialized with the double precision constant `5D1`.

# 2.23 DO WHILE Statement

```
DO [s[,]] WHILE (e)    [: block-label]
```

***where:***

***s*** is an optional statement label of an executable statement, called the *terminal statement,* which follows the **DO** statement in the same program unit.

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

***block-label*** is an optional block label.

The **DO WHILE** statement is an extension to the FORTRAN 77 language.

*Example:*

```
          X = 0.0
          DO 10 WHILE( X .LT. 100.0 )
              PRINT *, X, SQRT( X )
              X = X + 1.0
    10    CONTINUE
```

If no statement label is present, the terminal statement of the DO-loop must be an **END DO** statement.

*Example:*

```
          X = 0.0
          DO WHILE( X .LT. 100.0 )
              PRINT *, X, SQRT( X )
              X = X + 1.0
          ENDDO
```

The following example illustrates the use of an integer arithmetic expression.

*Example:*

```
          I = 10
          DO WHILE( I )
              PRINT *, I
              I = I - 1
          ENDDO
          END
```

The **DO WHILE** statement, is similar to the **DO** statement. All nesting rules that apply to the **DO** statement also apply to the **DO WHILE** statement. The difference is the way in which the looping is accomplished; the DO-loop is executed while the logical expression of the **DO WHILE** statement has a true value or until control is transferred out of the DO-loop.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.24 ELSE Statement

```
ELSE
```

The **ELSE** statement is used in conjunction with the **IF** or **ELSE IF** statement. The range of the **ELSE** block is terminated by a matching **END IF** statement.

*Example:*

```
      IF( A .LT. B )THEN
          PRINT *, 'A is less than B'
      ELSE
          PRINT *, 'A is greater than or equal to B'
      END IF
```

Transfer of control into the range of an **ELSE** block is illegal. It is interesting to note that the **ELSE** statement may be identified by a statement label but it must not be referenced by any statement!

*Example:*

```
* Illegal branch to a labelled ELSE statement.
      IF( A .LT. B )THEN
          PRINT *, 'A is less than B'
100   ELSE
          PRINT *, 'A is greater than or equal to B'
          GO TO 100
      END IF
```

The above is an example of an illegal way to construct an infinitely repeating loop. The following is the correct way to do this.

*Example:*

```
      IF( A .LT. B )THEN
          PRINT *, 'A is less than B'
      ELSE
100       PRINT *, 'A is greater than or equal to B'
          GO TO 100
      END IF
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.25 ELSE IF Statement

```
ELSE IF (e) THEN
```

***where:***

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

The **ELSE IF** statement is used in conjunction with the **IF** statement. The range of the **ELSE IF** block is terminated by another **ELSE IF** statement, an **ELSE** statement, or an **END IF** statement.

*Example:*

```
      IF( A .LT. B )THEN
          PRINT *, 'A is less than B'
      ELSE IF( A .EQ. B )THEN
          PRINT *, 'A is equal to B'
      ELSE
          PRINT *, 'A is greater than B'
      END IF
```

Transfer of control into the range of an **ELSE IF** block is illegal. It is interesting to note that the **ELSE IF** statement may be identified by a statement label but it must not be referenced by any statement!

*Example:*

```
    * Illegal transfer into the range of
    * an ELSE IF statement.
          IF( A .EQ. 0.0 )GO TO 110
          IF( A .LT. B )THEN
              PRINT *, 'A is less than B'
          ELSE IF( A .EQ. B )THEN
              PRINT *, 'A is equal to B or'
    110       PRINT *, 'A is equal to 0'
          ELSE
              PRINT *, 'A is greater than B'
          END IF
```

The above is an example of an illegal attempt to branch into the range of an **ELSE IF** block.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.26 END Statement

```
END
```

The **END** statement indicates the end of a sequence of statements and comment lines of a program unit. Execution of an **END** statement in a function or subroutine subprogram has the same effect as a **RETURN** statement. Control is returned to the invoking program unit. Execution of an **END** statement in a main program causes termination of execution of the program.

*Example:*

```
SUBROUTINE EULER( X, Y, Z )
     .
     .
     .
END
```

Upon executing the **END** statement, execution control is returned to the calling program unit.

*Example:*

```
PROGRAM PAYROL
     .
     .
     .
END
```

Upon executing the **END** statement, execution of the program is terminated.

Some rather special rules apply to the **END** statement. The statement is written in columns 7 to 72 of an initial line. In other words, it must not be continued. Also, no other statement in the program unit may have an initial line that appears to be an **END** statement.

*Example:*

```
* An illegal ENDIF statement.
      IF( A .LT. B )THEN
           .
           .
           .
      END
     &IF
```

The above **END IF** statement is illegal since the initial line appears to be an **END** statement.

# 2.27 END AT END Statement

```
END AT END
```

The **END AT END** statement is used in conjunction with the structured **AT END** statement. The **END AT END** statement marks the end of a sequence of statements which are part of an AT END-block. The **AT END** statement marks the beginning of the AT END-block. The AT END-block is executed when the preceding **READ** statement terminates because of an end-of-file condition.

*Example:*

```
READ( UNIT=1, FMT='(3I5)' ) I, J, K
AT END DO
    PRINT *, 'END-OF-FILE ENCOUNTERED ON UNIT 1'
    EOFSW = .TRUE.
END AT END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.28 END BLOCK Statement

```
END BLOCK
```

The **END BLOCK** statement is used to terminate a REMOTE-block. The **END BLOCK** statement is implicitly a transfer statement, since it returns program control from a REMOTE-block.

*Example:*

```
REMOTE BLOCK A
    I=I+1
    PRINT *, 'I=',I
END BLOCK
```

For more information, see the description of the **EXECUTE** and **REMOTE BLOCK** statements or the chapter entitled "Program Structure Control Statements" on page 193.

# 2.29 END DO Statement

```
END DO
```

The **END DO** statement is used to terminate the range of a "structured" **DO** statement. A structured **DO** statement is one in which a statement label is not present. For more information, see the description of the structured **DO** statement or the chapter entitled "Program Structure Control Statements" on page 193.

*Example:*

```
DO X = -5.1, 12.8, 0.125
     .
     .
     .
END DO
```

*Example:*

```
X = -5.1
DO WHILE( X .LE. 12.8 )
     .
     .
     .
     X = X + 0.125
END DO
```

# 2.30 ENDFILE Statement

```
ENDFILE u
ENDFILE (alist)
```

***where:***

***u*** is an external unit identifier.

***alist*** is a list of endfile specifiers separated by commas:

```
[UNIT =] u
IOSTAT = ios
ERR = s
```

Execution of an **ENDFILE** statement causes an endfile record to be written to the file connected to the specified unit. The file is then positioned after the endfile record. If the file may be connected for *direct access,* only those records before the endfile record are considered to have been written. Thus, only those records before the endfile record may be read during subsequent direct access connections to the file.

***Endfile Specifiers***

***[UNIT =] u***

`u` is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional **UNIT=** specifier is omitted then the specifier must be the first item in the list of specifiers.

***IOSTAT = ios***

is an *input/output status specifier.* The integer variable or integer array element `ios` is defined with zero if no error condition exists or a positive integer value if an error condition exists.

***ERR = s***

is an *error specifier* and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s.`

*Example:*

```
          LOOP
              READ( UNIT=7, END=100, FMT=200 )RECORD
              WRITE( UNIT=8, FMT=200 )RECORD
          ENDLOOP
  100     ENDFILE( UNIT=8 )
```

In the previous example, we illustrate how one might read all the records from one file (unit 7), write them to another file (unit 8) and then write an endfile record to the end of the file on unit 8.

*Notes:*

1. The unit must be connected for sequential access.

2. After execution of an **ENDFILE** statement, a **BACKSPACE** or **REWIND** statement must be used to reposition the file before any other input/output statement which refers to this file can be executed.

3. If the file did not exist before execution of the **ENDFILE** statement then it will be created after execution of this statement.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.31 END GUESS Statement

```
END GUESS
```

The **END GUESS** statement is used in conjunction with the structured **GUESS** statement. The **END GUESS** statement marks the end of a series of GUESS-ADMIT blocks.

*Example:*

```
CHARACTER CH
READ *, CH
GUESS
    IF( CH .LT. 'a' )QUIT
    IF( CH .GT. 'z' )QUIT
    PRINT *, 'Lower case letter'
ADMIT
    IF( CH .LT. 'A' )QUIT
    IF( CH .GT. 'Z' )QUIT
    PRINT *, 'Upper case letter'
ADMIT
    IF( CH .LT. '0' )QUIT
    IF( CH .GT. '9' )QUIT
    PRINT *, 'Digit'
ADMIT
    PRINT *, 'Special character'
END GUESS
END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.32 END IF Statement

```
END IF
```

The **END IF** statement is used in conjunction with the block **IF** statement. The **END IF** statement marks the end of a sequence of statements which are to be conditionally executed.

*Example:*

```
IF( X .LT. 100.0 )THEN
    PRINT *, 'X IS LESS THAN 100'
END IF
```

The **END IF** statement can also be used in conjunction with the **ELSE** and **ELSE IF** statements. For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.33 END LOOP Statement

```
END LOOP
```

The **END LOOP** statement is used in conjunction with the structured **LOOP** statement. The **END LOOP** statement marks the end of a sequence of statements which are to be repeated. The **LOOP** statement marks the beginning of the loop. The LOOP-block is executed until control is transferred out of the LOOP-block.

The **QUIT** statement may be used to transfer control out of a LOOP-block.

*Example:*

```
LOOP
    READ *, X
    IF( X .GT. 99.0 ) QUIT
    PRINT *, X
END LOOP
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.34 END MAP Statement

```
END MAP
```

The **END MAP** statement is used in conjunction with the **MAP** declarative statement. The **END MAP** statement marks the end of a **MAP** structure. The following example maps out a 4-byte integer on an Intel 80x86-based processor.

*Example:*

```
STRUCTURE /MAPINT/
    UNION
        MAP
            INTEGER*4 LONG
        END MAP
        MAP
            INTEGER*2 LO_WORD
            INTEGER*2 HI_WORD
        END MAP
        MAP
            INTEGER*1 BYTE_0
            INTEGER*1 BYTE_1
            INTEGER*1 BYTE_2
            INTEGER*1 BYTE_3
        END MAP
    END UNION
END STRUCTURE

RECORD /MAPINT/ I

I%LONG = '01020304'x
PRINT '(2Z4)', I%LO_WORD, I%HI_WORD
END
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.35 END SELECT Statement

```
END SELECT
```

The **END SELECT** statement is used in conjunction with the **SELECT** statement. The **END SELECT** statement marks the end of a series of **CASE** blocks.

*Example:*

```
SELECT CASE ( CH )
CASE ( 'a' : 'z' )
    PRINT *, 'Lower case letter'
CASE ( 'A' : 'Z' )
    PRINT *, 'Upper case letter'
CASE ( '0' : '9' )
    PRINT *, 'Digit'
CASE DEFAULT
    PRINT *, 'Special character'
END SELECT
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.36 END STRUCTURE Statement

```
END STRUCTURE
```

The **END STRUCTURE** statement is used in conjunction with the **STRUCTURE** declarative statement. The **END STRUCTURE** statement marks the end of a structure definition.

*Example:*

```
STRUCTURE /ADDRESS/
    CHARACTER*20 STREET
    CHARACTER*20 CITY
    CHARACTER*20 STATE
    CHARACTER*20 COUNTRY
    CHARACTER*10 ZIP_CODE
END STRUCTURE

STRUCTURE /PEOPLE/
    CHARACTER*20 NAME
    RECORD /ADDRESS/ ADDR
    INTEGER*2 AGE
END STRUCTURE
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.37 END UNION Statement

```
END UNION
```

The **END UNION** statement is used in conjunction with the **UNION** declarative statement. The **END UNION** statement marks the end of a series of **MAP** structures. The following example maps out a 4-byte integer on an Intel 80x86-based processor.

*Example:*

```
STRUCTURE /MAPINT/
    UNION
        MAP
            INTEGER*4 LONG
        END MAP
        MAP
            INTEGER*2 LO_WORD
            INTEGER*2 HI_WORD
        END MAP
        MAP
            INTEGER*1 BYTE_0
            INTEGER*1 BYTE_1
            INTEGER*1 BYTE_2
            INTEGER*1 BYTE_3
        END MAP
    END UNION
END STRUCTURE

RECORD /MAPINT/ I

I%LONG = '01020304'x
PRINT '(2Z4)', I%LO_WORD, I%HI_WORD
END
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.38 END WHILE Statement

```
END WHILE
```

The **END WHILE** statement is used in conjunction with the structured **WHILE** statement. The **END WHILE** statement marks the end of a sequence of statements which are to be repeated. The **WHILE** statement marks the beginning of the WHILE-block. The WHILE-block is executed while the logical expression (or integer arithmetic expression) of the **WHILE** statement has a true (or non-zero) value or until control is transferred out of the WHILE-block.

*Example:*

```
X = 1.0
WHILE( X .LT. 100 )DO
    PRINT *, X, SQRT( X )
    X = X + 1.0
END WHILE
```

*Example:*

```
I = 10
WHILE( I )DO
    PRINT *, I
    I = I - 1
ENDWHILE
END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.39 ENTRY Statement

```
ENTRY name [( [d [, d] ...] )]
```

***where:***

***name*** is a symbolic name of an entry in a function or subroutine subprogram. If the **ENTRY** statement appears in a subroutine subprogram then `name` is a *subroutine name.* If the **ENTRY** statement appears in a function subprogram then `name` is an *external function name.*

***d*** is a variable name, array name, dummy procedure name, or an asterisk. `d` is called a *dummy argument.* An asterisk is allowed only in a subroutine subprogram.

The **ENTRY** statement is used to define an alternate entry into a subprogram.

*Example:*

```
          PRINT *, TMAX2( 121.0, -290.0 )
          PRINT *, TMAX3( -1.0, 12.0, 5.0 )
          END

          FUNCTION TMAX3( ARGA, ARGB, ARGC )
              T3 = ARGC
              GO TO 10
          ENTRY TMAX2( ARGA, ARGB )
              T3 = ARGA
    10        TMAX2 = ARGA
              IF( ARGB .GT. TMAX2 ) TMAX2 = ARGB
              IF( T3 .GT. TMAX2 ) TMAX2 = T3
          END
```

In the above example, an entry was defined to permit us to find the maximum of two real variables. Either the entry name `TMAX2` or the function name `TMAX3` could have been used as the variable for returning the maximum value since they agree in type. It is not necessary to precede an **ENTRY** statement with a transfer statement as the **ENTRY** statement is not an executable statement; the next statement executed will be the first executable statement following the **ENTRY** statement.

*Notes:*

1. No dummy arguments need be specified in the **ENTRY** statement. If this is the case, the parentheses () are optional.

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

# 2.40 EQUIVALENCE Statement

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

***where:***

***nlist*** is a list of at least two names, each name separated by a comma.

The names appearing in `nlist` can be variable names, array names, array element names, character names, character substring names, and character array element substring names. Dummy arguments are not allowed in `nlist`.

The **EQUIVALENCE** statement specifies that the storage occupied by the entities appearing in `nlist` all start at the same place. It in no way changes the characteristics of an object. For example, if a variable is equivalenced to an array, the variable does not inherit the properties of the array. Similarly, if a variable of type integer is equivalenced to a variable of type real, there is no implied type conversion.

If an array element name appears in an **EQUIVALENCE** statement, the number of subscript expressions must be the same as the number of dimensions specified when the array was declared and each subscript expression must be in the range specified. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 allows a single subscript expression for a multi-dimensional array. An array name used by itself is equivalent to specifying the first element of the array.

If a character substring appears in an **EQUIVALENCE** statement, the substring defined by the substring expression must be properly contained in the character entity being substring. A character name used by itself is equivalent to specifying the first character of the character variable.

*Example:*

```
REAL A,B
DIMENSION A(10),B(20)
EQUIVALENCE (A,B(16))
```

In the above example, the first 5 elements of `A` occupy the same storage as the last 5 elements of `B`.

*Example:*

```
DIMENSION A(10)
EQUIVALENCE (C,A(2)),(D,A(4))
```

In the above example, `C` is assigned the same storage unit as `A(2)` and `D` is assigned the same storage unit as `A(4)`.

The following example illustrates a Open Watcom FORTRAN 77 extension.

*Example:*
```
      REAL A(2,10),B(20),C(2,2,5)
      EQUIVALENCE (A(5),B(1)),(B(1),C(1))
```

In the above example, a single subscript is specified for arrays `A` and `C`. The following table shows the mapping of a 2-dimensional array onto a 1-dimensional array.

```
      A(1,1)  ==  A(1)
      A(2,1)  ==  A(2)
      A(1,2)  ==  A(3)
      A(2,2)  ==  A(4)
      A(1,3)  ==  A(5)
      A(2,3)  ==  A(6)
              .
              .
              .
```

In the above table, "==" is read as "is equivalent to". In FORTRAN, arrays are stored in "column major" format (i.e., arrays are stored column by column rather than row by row).

*Example:*
```
      CHARACTER*5 A, D
      EQUIVALENCE (A(3:5), D(1:3))
```

In this example, the last 3 characters of `A` occupy the same character storage units as the first 3 characters of `D`.

There are certain restrictions on **`EQUIVALENCE`** statements. It is not possible to equivalence a storage unit to 2 different storage units. This is illustrated by the following example.

*Example:*
```
    * Illegally equivalencing a storage unit to
    * 2 different storage units.
          DIMENSION A(2)
          EQUIVALENCE (A(1),B),(A(2),B)
```

`B` has been given 2 different storage units.

It is also not possible to specify that consecutive storage units be non-consecutive. For example,

*Example:*
```
    * Illegally equivalencing consecutive storage units to
    * non-consecutive storage units.
          DIMENSION A(2),B(2)
          EQUIVALENCE (A(1),B(2)),(A(2),B(1))
```

`A(1)` and `A(2)` are consecutive but `B(1)` and `B(2)` are not.

The FORTRAN 77 standard specifies that character and numeric data cannot be equivalenced; Open Watcom FORTRAN 77 allows character and numeric data to be equivalenced.

# 2.41 EXECUTE Statement

```
EXECUTE name
```

***where:***

***name*** is the name of a **REMOTE BLOCK** located in the same program unit.

The **EXECUTE** statement allows a named block of code to be executed. The named block of code may be defined anywhere in the same program unit and is delimited by the **REMOTE BLOCK** and **END BLOCK** statements. Executing a REMOTE-block is similar in concept to calling a subroutine, with the advantage that shared variables do not need to be placed in a **COMMON** block or passed in an argument list. When execution of the REMOTE-block is complete (i.e., when the **END BLOCK** statement is executed), control returns to the statement following the **EXECUTE** statement which invoked it.

*Example:*

```
EXECUTE INCR
PRINT *, 'FIRST'
EXECUTE INCR
PRINT *, 'SECOND'
    .
    .
    .
REMOTE BLOCK INCR
    I=I+1
    PRINT *, 'I=',I
END BLOCK
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.42 EXIT Statement

```
EXIT    [: block-label]
```

The **EXIT** statement is used to transfer control:

1. from within a loop (DO, DO WHILE, WHILE or LOOP) to the statement following the loop,
2. from within a GUESS or ADMIT block to the statement following the **ENDGUESS** statement, or
3. from within a remote block to the statement following the **EXECUTE** statement that invoked the remote block.

The **EXIT** statement may be used to cause a transfer of control to the first executable statement that follows the terminal statement of the block which contains it. Examples of such terminal statements are **END DO**, **END LOOP**, **END WHILE**, **UNTIL**, etc. If `block-label` is present then control is transferred out of the block identified by that block label. The **EXIT** statement is an extension to the FORTRAN 77 language.

*Example:*

```
LOOP
    WRITE( UNIT=*, FMT='(A)' ) 'Enter a number'
    READ( UNIT=*, FMT='(F10.4)', IOSTAT=IOS ) X
    IF( IOS .NE. 0 ) EXIT
    IF( X .LT. 0 ) EXIT
    PRINT *, X, SQRT( X )
END LOOP
END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.43 EXTERNAL Statement

```
EXTERNAL p [,p] ...
```

***where:***

***p*** is the name of an external procedure, dummy procedure or block data subprogram.

The **EXTERNAL** statement identifies a symbolic name to be a dummy procedure or an external procedure and allows these names to be passed as an actual argument. In the following example, SAM, ERRRTN and POLY are declared to be external procedures.

*Example:*

```
EXTERNAL SAM, ERRRTN, POLY
```

In the following example, F is declared to be an external procedure and is passed as such to subroutine SAM. If the **EXTERNAL** statement were eliminated then the variable F would be passed on to subroutine SAM since there is no way of knowing that F is an external function.

*Example:*

```
EXTERNAL F
     .
     .
     .
CALL SAM( F )
```

The appearance of an intrinsic function in an **EXTERNAL** statement declares that name to be an external procedure and the intrinsic function by that name is no longer available in that program unit. This allows the programmer to define a function by the same name as an intrinsic function. In the following example, the programmer's SIN function will be called instead of the intrinsic SIN function.

*Example:*

```
EXTERNAL SIN
     .
     .
     .
CALL SIN( .1 )
```

A statement function name must not appear in an **EXTERNAL** statement. A name must only appear in an **EXTERNAL** statement once.

# 2.44 FORMAT Statement

```
label    FORMAT fs
```

***where:***

***fs*** is a format specification and is described in the chapter entitled "Format" on page 225.

***label*** is the statement label used by an I/O statement to identify the **FORMAT** statement to be used. The **FORMAT** statement must be labelled.

*Example:*

```
          REAL X
          X = 234.43
          PRINT 100, X
    100   FORMAT(F10.2)
          END
```

In the previous example, the **PRINT** statement uses the format specification in the **FORMAT** statement whose statement label is 100 to display the value of `X.`

For more information on the **FORMAT** statement, see the chapter entitled "Format" on page 225.

# 2.45 FUNCTION Statement

A **FUNCTION** statement is used to define the start of a function subprogram. There are two forms of the **FUNCTION** function statement. The second form is a Open Watcom FORTRAN 77 extension.

## 2.45.1 Standard FUNCTION Statement

```
[type] FUNCTION fun ( [d [, d] ...] )
```

***where:***

***type*** is one of `LOGICAL`, `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX` or `CHARACTER [*len]`.

***fun*** is a symbolic name of a function subprogram.

***d*** is a variable name, array name, or a dummy procedure name. `d` is called a *dummy argument.*

***len*** is called the length specification and is the length (number of characters) of the result of the character function. It has one of the following forms:

***(1)*** An unsigned positive integer constant.

***(2)*** A positive integer constant expression enclosed in parentheses.

***(3)*** An asterisk in parentheses, `(*)`.

*Example:*

```
PRINT *, TMAX3( -1.0, 12.0, 5.0 )
END

FUNCTION TMAX3( ARGA, ARGB, ARGC )
    TMAX3 = ARGA
    IF( ARGB .GT. TMAX3 ) TMAX3 = ARGB
    IF( ARGC .GT. TMAX3 ) TMAX3 = ARGC
END
```

In the above example, the function `TMAX3` is defined to find the maximum of three real variables.

*Notes:*

1. No dummy arguments need be specified in the **FUNCTION** statement. However, the parentheses () are mandatory.

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

## 2.45.2 Extended FUNCTION Statement

```
[type[*len]] FUNCTION fun[*len] ( [d [, d] ...] )
```

***where:***

***type*** is one of `LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER` or `RECORD /typename/`

***fun*** is a symbolic name of a function subprogram.

***d*** is a variable name, array name, or a dummy procedure name. `d` is called a *dummy argument.*

***len*** is called the length specification and has one of the following forms:

***(1)*** An unsigned positive integer constant.

***(2)*** A positive integer constant expression enclosed in parentheses.

***(3)*** An asterisk in parentheses, `(*).`

For valid values of `len,` refer to the appropriate type declaration statement.

This form of the **`FUNCTION`** statement is an extension to the FORTRAN 77 language.

*Example:*

```
INTEGER*2 MOD2, I, J
I = 12
J = 5
PRINT *, MOD2( I, J )
END

INTEGER*2 FUNCTION MOD2( I, J )
INTEGER*2 I, J
INTEGER II, JJ
II = I
JJ = J
MOD2 = MOD(II, JJ)
END
```

*Notes:*

1. No dummy arguments need be specified in the **`FUNCTION`** statement. However, the parentheses () are mandatory.

2. The length specification can appear only once in the **`FUNCTION`** statement.

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

# 2.46 Unconditional GO TO Statement

```
GO TO s
```

***where:***

***s*** is the statement label of an executable statement that appears in the same program unit as the **GO TO** statement.

*Example:*

```
          GO TO 10
          .
          .
          .
10        S = S + 1
```

When the **GO TO** statement is executed, control is transferred to the statement identified by that label. In the above example, the **GO TO** statement causes execution to proceed to the statement labelled 10.

*Example:*

```
    * An illegal GO TO statement
          GO TO 100
          .
          .
          .
100       FORMAT( 1X, 3F10.2 )
```

The above example contains an illegal **GO TO** statement since the statement identified by the label 100 is not executable.

# 2.47 Computed GO TO Statement

```
GO TO (s [,s]...) [,] i
```

***where:***

***i*** is an integer expression.

***s*** is the statement label of an executable statement that appears in the same program unit as the computed **GO TO** statement.

The integer expression `i` is evaluated and the `i`th label is selected for transfer of control. If `i` is less than 1 or greater than the number of statement labels in the list then execution control continues with the next executable statement that follows the computed **GO TO** statement.

*Example:*

```
          GO TO (110, 120, 130, 140) INDEX
    100   CALL AUDIT
```

In the above example, control is transferred to the statement identified by the label 110 if `INDEX` has the value 1, the label 120 if `INDEX` has the value 2, etc. If `INDEX` has a value that is negative, zero or larger than 4, control continues with the statement labelled 100. In this example, the integer expression consists simply of an integer variable.

*Example:*

```
          GO TO (100, 200, 100, 200, 100, 200), I/10
```

The above example illustrates that statement labels may be repeated in the list and that a "," may follow the closing right parenthesis.

# 2.48 Assigned GO TO Statement

```
GO TO i [[,] (s [,s]...)]
```

***where:***

***i*** is an integer variable name.

***s*** is the statement label of an executable statement that appears in the same program unit as the assigned **GO TO** statement.

The variable `i` must be defined with the value of a statement label of an executable statement that appears in the same program unit (see the **ASSIGN** statement). The execution of the assigned **GO TO** statement causes a transfer of control to the statement that is identified by that label.

*Example:*

```
          INTEGER RET
          X = 0.0
          ASSIGN 100 TO RET
          GO TO 3000
100       X = X + 1
          ASSIGN 110 TO RET
          GO TO 3000
110       X = X + 1
          .
          .
          .
*          Print both X and its square root
3000      Y = SQRT( X )
          PRINT *, X, Y
          GO TO RET
```

In the above example, we illustrate the use of the **ASSIGN** statement and the assigned **GO TO** statement to implement a "local subroutine" in a program unit. A sequence of often-used code can be "called" using the unconditional **GO TO** statement and "return" is accomplished using the assigned **GO TO** statement. Care must be exercised to properly assign the return label value.

If a list of statement labels is present then the statement label assigned to `i` must be in the list. If it is not in the list, an error will occur when the assigned **GO TO** statement is executed. Unlike the computed **GO TO** statement, execution does not continue with the next statement. This is demonstrated by the following example. Note that the "," preceding the statement label list is optional.

*Example:*

```
* Illegal use of the assigned GO TO:
*   Statement label 100 does not appear in the statement
*   label list of the assigned GO TO statement.
      ASSIGN 100 TO ICASE
      GO TO ICASE, (110, 120, 130)
*      beginning of selections
100           PRINT *, 100
      GO TO 200
110           PRINT *, 110
      GO TO 200
120           PRINT *, 120
      GO TO 200
130           PRINT *, 130
*      end of selections
200     END
```

# 2.49 GUESS Statement

```
GUESS    [: block-label]
```

The **GUESS** statement is an extension to the FORTRAN 77 language. The **GUESS** statement marks the beginning of a block of statements for which a certain assumption or hypothesis has been made. This hypothesis may be tested using logical **IF** statements in conjunction with **QUIT** statements. The **ADMIT** statement may be used to mark the beginning of an alternate hypothesis. The **END GUESS** statement is used to mark the end of a series of GUESS-ADMIT blocks.

*Example:*

```
CHARACTER CH
READ *, CH
GUESS
    IF( CH .LT. 'a' )QUIT
    IF( CH .GT. 'z' )QUIT
    PRINT *, 'Lower case letter'
ADMIT
    IF( CH .LT. 'A' )QUIT
    IF( CH .GT. 'Z' )QUIT
    PRINT *, 'Upper case letter'
ADMIT
    IF( CH .LT. '0' )QUIT
    IF( CH .GT. '9' )QUIT
    PRINT *, 'Digit'
ADMIT
    PRINT *, 'Special character'
END GUESS
END
```

An optional block label may be specified with the **GUESS** statement.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.50 Arithmetic IF Statement

```
IF (e) s1, s2, s3
```

***where:***

***e*** is an integer, real, or double precision expression.

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

The expression e is evaluated and if the value is less than zero then transfer is made to the statement identified by label s1. If the value is equal to zero then transfer is made to the statement identified by label s2. If the value is greater than zero then transfer is made to the statement identified by label s3.

*Example:*

```
      IF( SIN( X ) ) 10, 20, 30
10    PRINT *, 'SIN(X) IS < 0'
      GO TO 40
20    PRINT *, 'SIN(X) = 0'
      GO TO 40
30    PRINT *, 'SIN(X) > 0'
40    CONTINUE
```

The above example evaluates the sine of the real variable X and prints whether the result is less than 0, equal to 0, or greater than 0.

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

*Example:*

```
      IF( SIN( X ) ) 10, 10, 30
10    PRINT *, 'SIN(X) IS <= 0'
      GO TO 40
30    PRINT *, 'SIN(X) > 0'
40    CONTINUE
```

The above example evaluates the sine of the real variable X and prints whether the result is less than or equal to zero, or that it is greater than 0.

# 2.51 Logical IF Statement

```
IF (e) st
```

***where:***

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

***st*** is an executable statement. Only certain executable statements are allowed. See the section entitled "Classifying Statements" on page 9 at the beginning of this chapter for a list of allowable statements.

The expression `e` is evaluated and must result in a true or a false value. If the result is true then the statement `st` is executed, otherwise it is not executed.

*Example:*

```
IF( A .LT. B )PRINT *, 'A < B'
```

In the above example, the logical expression `A .LT. B` is evaluated and, if it is true, the message `A < B` is printed. A logical expression is one in which the result is either true or false. An expression such as `1 + 2` is clearly not an example of a logical expression.

Logical variables have logical values of true or false and may also be used in the logical expression. Consider the following two examples.

*Example:*

```
LOGICAL RESULT
RESULT = A .LT. B
IF( RESULT )PRINT *, 'A < B'
```

The above example is equivalent to the preceding one but introduces the use of a logical variable.

*Example:*

```
LOGICAL RESULT
RESULT = A .LT. B
IF( .NOT. RESULT )PRINT *, 'A >= B'
```

In the above example, the logical expression is negated through the use of the `.NOT.` operator in order to test for the inverse condition, namely `.GE.`.

Much more complex logical expressions can be constructed and then tested for their truth value.

*Example:*

```
IF( A.LT.B .OR. C.GE.D )PRINT *, 'A<B or C>=D'
```

An example of an integer expression in an **IF** statement follows:

*Example:*

```
          I = 1
    * Integer arithmetic expression
          IF( I )THEN
                  PRINT *, 'Yes'
          ENDIF
    * Equivalent logical expression
          IF( I .NE. 0 )THEN
                  PRINT *, 'Yes'
          ENDIF
          END
```

# 2.52 Block IF Statement

There are two forms of the block **IF** statement. The second is a Open Watcom FORTRAN 77 extension.

## 2.52.1 Standard Block IF Statement

```
IF (e) THEN
```

***where:***

***e*** is a logical expression.

The block **IF** statement is used in conjunction with the **ELSE IF**, **ELSE**, and **END IF** statements.

*Example:*

```
IF( A .LT. B )THEN
    PRINT *, 'A < B'
END IF
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

## 2.52.2 Extended Block IF Statement

```
IF (e) THEN    [: block-label]
```

***where:***

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

***block-label*** is an optional block label.

This form of the block **IF** statement is an extension to the FORTRAN 77 language. It is identical to the standard form of the block **IF** statement with the exception that an integer arithmetic expression and an optional block label are permitted.

*Example:*

```
IF( I .EQ. 10 )THEN : IFBLK
    IF( J .EQ. 20 )THEN
        .
        .
        .
        IF( K. EQ. 0 )QUIT : IFBLK
        .
        .
        .
    END IF
    .
    .
    .
END IF
```

In the previous example, the **QUIT** statement will transfer control to the statement following the second **END IF** statement.

# 2.53 IMPLICIT Statement

Open Watcom FORTRAN 77 supports three forms of the **IMPLICIT** statement. The second and third forms are extensions to the FORTRAN 77 language.

## 2.53.1 Standard IMPLICIT Statement

```
IMPLICIT type (a [,a] ...) [,type (a [,a] ...)]...
```

***where:***

***type*** is one of LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX or CHARACTER[*len].

***a*** is either a single letter or a range of letters denoted by separating the first letter in the range from the last letter in the range by a minus sign.

***len*** is the length of the character entities and is a positive unsigned integer constant or a positive integer constant expression enclosed in parentheses. If len is not specified, the length is 1.

## 2.53.2 Extended IMPLICIT Statement

```
IMPLICIT type[*len] (a [,a] ...)
       [,type[*len] (a [,a] ...)] ...
```

***where:***

***type*** is one of LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX or CHARACTER.

***len*** is a positive unsigned integer constant or a positive integer constant expression enclosed in parentheses. If type is CHARACTER then (*) is also allowed. The possible values of len are as follows:

1. If type is LOGICAL then len can be 1 or 4. The default is 4.
2. If type is INTEGER then len can be 1, 2 or 4. The default is 4.
3. If type is REAL then len can be 4 or 8. The default is 4.
4. If type is DOUBLE PRECISION then len cannot be specified.
5. If type is COMPLEX then len can be 8 or 16. The default is 8.
6. If type is CHARACTER then len can be (*) or any positive integer.

## 2.53.3 IMPLICIT NONE Statement

```
IMPLICIT NONE
```

## 2.53.4 Description of IMPLICIT Statement

The **IMPLICIT** statement defines the default type and length for all variables, arrays, symbolic constants, external functions and statement functions that begin with any letter that has appeared in an **IMPLICIT** statement as a single letter or as a member of a range of letters.

The following example specifies that any name beginning with the letters A, D, E, F or G will have default a default type of integer and any name beginning with the letters X, Y or Z will have a default type of character and length 3.

*Example:*

```
IMPLICIT INTEGER (A,D-G), CHARACTER*3 (X-Z)
```

The next example illustrates the extended form of the **IMPLICIT** statement.

*Example:*

```
IMPLICIT INTEGER*2 (A,B), LOGICAL*1 (C-F)
IMPLICIT COMPLEX*16 (X,Y,Z), REAL*8 (P)
```

Specifying NONE in the **IMPLICIT** statement will cause Open Watcom FORTRAN 77 to issue an error when a symbol is used and has not appeared in a type specification statement.

*Example:*

```
* Referencing X will cause an error
      IMPLICIT NONE
      X = 13143.383
```

In the above example, the **IMPLICIT** statement specifies that the type of all symbols must be explicitly declared in a type specification statement. The assignment statement will cause an error since the type of X has not been explicitly declared.

*Notes:*

1. The implicit type set by an **IMPLICIT** statement may be overridden or confirmed for any variable, array, symbolic constant, external function or statement function name by its appearance in a type statement. The default length specification may also be overridden or confirmed in a type statement.

*Example:*

```
IMPLICIT CHARACTER*10 (S-U)
IMPLICIT INTEGER*2 (P)
CHARACTER STRING
INTEGER POINTS
```

In the above example, the variable `STRING` is of type character but its length is 1 since it has appeared in a **CHARACTER** statement which has a default length of 1. Also, the variable `POINTS` is of type integer but its length is 4 since it has appeared in an **INTEGER** statement which has a default length of 4.

2. A letter cannot appear more than once as a single letter or be included in a range of letters in all **IMPLICIT** statements in a program unit.

3. An **IMPLICIT** statement applies only to the program unit that contains it.

4. The **IMPLICIT** statement does not change the type of intrinsic functions.

5. A program unit can contain more than one **IMPLICIT** statement.

6. Within the specification statements of a program unit, **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements.

7. The **IMPLICIT NONE** statement is allowed only once in a program unit. Furthermore, no other **IMPLICIT** statement can be specified in the program unit containing an **IMPLICIT NONE** statement.

# 2.54 INCLUDE Statement

```
INCLUDE 'inc_spec'
```

***where:***

***inc_spec*** is an include specification. You should refer to the compiler's User's Guide for a detailed description of an include specification and include file processing.

*Example:*

```
INCLUDE 'GBLDEFS'
    .
    .
    .
END
```

# 2.55 INQUIRE Statement

The **INQUIRE** statement is used to ask about certain properties of a named file or its connection to a particular unit. There are two forms of the **INQUIRE** statement; inquire by file name and inquire by unit.

## 2.55.1 INQUIRE by FILE

```
INQUIRE (iflist)
```

***where:***

***iflist*** includes the **FILE=** specifier and may include at most one of each of the inquiry specifiers listed below. Specifiers are separated by commas. The **FILE=** specifier has the form

```
FILE = fin
```

where `fin` is a character expression whose value when trailing blanks are removed is the name of a file being inquired about. The file need not exist or be connected to a unit.

*Example:*

```
LOGICAL EX, OD
INTEGER NUM
INQUIRE( FILE='ROLL',EXIST=EX,OPENED=OD,NUMBER=NUM )
```

In the above example, information is requested on the file `PAYROLL.` In particular, we want to know if it exists, whether it is connected to a unit, and what the unit number is (if it is indeed connected).

## 2.55.2 INQUIRE by UNIT

```
INQUIRE (iulist)
```

***where:***

***iulist*** includes the **UNIT=** specifier and may include at most one of each of the inquiry specifiers listed below. Specifiers are separated by commas. The **UNIT=** specified has the form

```
[UNIT =] u
```

where `u` is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional **UNIT=** specifier is omitted then the specifier must be the first item in the list of specifiers.

*Example:*

```
LOGICAL EX, OD
CHARACTER*30 FN
INQUIRE( UNIT=7, EXIST=EX, OPENED=OD, NAME=FN )
```

In the above example, information is requested on unit 7. In particular, we want to know if the unit exists, whether it is connected to a file, and, if so, what the file name is.

## 2.55.3 Inquiry Specifiers

The following inquiry specifiers are supported.

```
IOSTAT = ios
ERR = s
EXIST = ex
OPENED = od
NUMBER = num
NAMED = nmd
NAME = fn
ACCESS = acc
SEQUENTIAL = seq
DIRECT = dir
FORM = fm
FORMATTED = fmt
UNFORMATTED = unf
RECL = rcl
NEXTREC = nr
BLANK = blnk
```

As an extension to the FORTRAN 77 language, the following inquiry specifiers are also supported.

```
ACTION = act
CARRIAGECONTROL = cc
RECORDTYPE = rct
BLOCKSIZE = bl
```

***IOSTAT = ios*** is an *input/output status specifier.* The integer variable or integer array element `ios` is defined with zero if no error condition exists or a positive integer value if an error condition exists.

***ERR = s*** is an *error specifier* and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s.`

***EXIST = ex*** `ex` is a logical variable or logical array element.

Inquire by file: The value .TRUE. is assigned if a file exists with the specified name; otherwise the value .FALSE. is assigned.

Inquire by unit: The value .TRUE. is assigned if the specified unit exists; otherwise the value .FALSE. is assigned.

***OPENED = od*** `od` is a logical variable or logical array element.

Inquire by file: The value .TRUE. is assigned if the specified file is connected to a unit; otherwise the value .FALSE. is assigned.

Inquire by unit: The value .TRUE. is assigned if the specified unit is connected to a file; otherwise the value .FALSE. is assigned.

***NUMBER = num***

`num` is an integer variable or integer array element that is assigned the value of the unit number to which the file is connected. If no unit is connected to the file then `num` becomes undefined.

***NAMED = nmd***

`nmd` is a logical variable or logical array element name that is assigned the value .TRUE. if the file has a name; otherwise the value .FALSE. is assigned.

***NAME = fn*** `fn` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `fn` to be a character substring.

It is assigned the name of the file, if the file has a name; otherwise it becomes undefined. The file name that is returned need not be the same as that given in a **`FILE=`** specifier but it must be suitable for use in the **`FILE=`** specification of an **`OPEN`** statement (e.g., the file name returned may have additional system qualifications attached to it).

***ACCESS = acc***

`acc` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `acc` to be a character substring.

It is assigned the value `'SEQUENTIAL'` if the file is connected for sequential access. It is assigned the value `'DIRECT'` if the file is connected for direct access. It is assigned an undefined value if there is no connection.

***SEQUENTIAL = seq***

`seq` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `seq` to be a character substring.

It is assigned the value `'YES'` if **`SEQUENTIAL`** is included in the set of allowed access methods for the file, the value `'NO'` if **`SEQUENTIAL`** is not included in the set of allowed access methods for the file, or `'UNKNOWN'` if Open Watcom FORTRAN 77 is unable to determine whether or not **`SEQUENTIAL`** is included in the set of allowed access methods for the file.

***DIRECT = dir*** `dir` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `dir` to be a character substring.

It is assigned the value `'YES'` if **`DIRECT`** is included in the set of allowed access methods for the file, the value `'NO'` if **`DIRECT`** is not included in the set of allowed access methods for the file, or `'UNKNOWN'` if Open Watcom FORTRAN 77 is unable to determine whether or not **`DIRECT`** is included in the set of allowed access methods for the file.

***FORM = fm*** `fm` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `fm` to be a character substring.

It is assigned the value 'FORMATTED' if the file is connected for formatted input/output, the value 'UNFORMATTED' if the file is connected for unformatted input/output, or an undefined value if there is no connection.

***FORMATTED = fmt***

fmt is a character variable or character array element. Open Watcom FORTRAN 77 also permits fmt to be a character substring.

It is assigned the value 'YES' if **FORMATTED** is included in the set of allowed forms for the file, the value 'NO' if **FORMATTED** is not included in the set of allowed forms for the file, or 'UNKNOWN' if Open Watcom FORTRAN 77 is unable to determine whether or not **FORMATTED** is included in the set of allowed forms for the file.

***UNFORMATTED = unf***

unf is a character variable or character array element. Open Watcom FORTRAN 77 also permits unf to be a character substring.

It is assigned the value 'YES' if **UNFORMATTED** is included in the set of allowed forms for the file, the value 'NO' if **UNFORMATTED** is not included in the set of allowed forms for the file, or 'UNKNOWN' if Open Watcom FORTRAN 77 is unable to determine whether or not **UNFORMATTED** is included in the set of allowed forms for the file.

***RECL = rcl*** rcl is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units (bytes). See the compiler User's Guide for a discussion of record length or size. If there is no connection or if the file is not connected for direct access then the value is undefined. The **RECL=** specifier is also allowed if the file is connected for sequential access.

***NEXTREC = nr***

nr is an integer variable or integer array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written then the value is 1. If the file is not connected for direct access or if the position of the file can not be determined because of an input/output error then nr becomes undefined.

***BLANK = blnk***

blnk is a character variable or character array element. Open Watcom FORTRAN 77 also permits blnk to be a character substring.

It is assigned the value 'NULL' if null blank control is in effect for the file connected for formatted input/output, and is assigned the value 'ZERO' if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the file is not connected for formatted input/output, blnk becomes undefined.

***ACTION = act*** act is a character variable or character array element. Open Watcom FORTRAN 77 also permits act to be a character substring.

It is assigned the value 'READ' if data can only be read from the file, 'WRITE' if data can only be written from the file, and 'READWRITE' if data can be both read and written.

***CARRIAGECONTROL = cc***

`cc` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `cc` to be a character substring.

It is assigned the value `'YES'` if the first character of each record is interpreted as a carriage control character and `'NO'` if no interpretation is placed on the first character of each record.

***RECORDTYPE = rct***

`rct` is a character variable or character array element. Open Watcom FORTRAN 77 also permits `rct` to be a character substring.

It is assigned a value that represents the record type (or record structure) that is used for the file. The value assigned depends on the system on which you are running the program. See the compiler User's Guide for a discussion of record types.

***BLOCKSIZE = bl***

`bl` is an integer variable or integer array element.

It is assigned a value that represents the internal buffer size that is used for input/output operations on the file. The value assigned depends on the system on which you are running the program. See the compiler User's Guide for a discussion of default internal buffer size.

## 2.55.4 Definition Status of Specifiers - Inquire by File

The following table summarizes which specifier variables or array elements become defined with values under what conditions when using the **`FILE=`** specifier.

```
IOSTAT = ios           (1)
EXIST = ex             (2)
OPENED = od            (2)
NUMBER = num           (4)
NAMED = nmd            (3)
NAME = fn              (3)
ACCESS = acc           (5)
SEQUENTIAL = seq       (3)
DIRECT = dir           (3)
FORM = fm              (5)
FORMATTED = fmt        (3)
UNFORMATTED = unf      (3)
RECL = rcl             (5)
NEXTREC = nr           (5)

ACTION = act           (5)
CARRIAGECONTROL = cc   (5)
RECORDTYPE = rct       (5)
BLOCKSIZE = bl         (5)
```

1. The **`IOSTAT=`** specifier variable is always defined with the most recent error status. If an error occurs during execution of the **`INQUIRE`** statement then the error status is defined with a positive integer; otherwise the status is that of the most recent input/output statement which referenced that file.

2. The specifier always becomes defined unless an error condition occurs.

3. This specifier becomes defined with a value only if the file name specified in the **FILE=** specifier is an acceptable file name and the named file exists. Also, no error condition can occur during the execution of the **INQUIRE** statement.

4. This specifier becomes defined with a value if and only if `od` becomes defined with the value .TRUE.. Also, no error condition can occur during the execution of the **INQUIRE** statement.

5. This specifier *may* become defined with a value only if `od` becomes defined with the value .TRUE.. However, there may be other conditions under which this specifier does not become defined with a value. In other words, (5) is a necessary, but not sufficient condition. For example, `blnk` is undefined if the file is not connected for formatted input/output.

## 2.55.5 Definition Status of Specifiers - Inquire by Unit

The following table summarizes which specifier variables or array elements become defined with values under what conditions when using the **UNIT=** specifier.

```
IOSTAT = ios           (1)
EXIST = ex             (2)
OPENED = od            (2)
NUMBER = num           (3)
NAMED = nmd            (3)
NAME = fn              (3)
ACCESS = acc           (3)
SEQUENTIAL = seq       (3)
DIRECT = dir           (3)
FORM = fm              (3)
FORMATTED = fmt        (3)
UNFORMATTED = unf      (3)
RECL = rcl             (3)
NEXTREC = nr           (3)

ACTION = act           (3)
CARRIAGECONTROL = cc   (3)
RECORDTYPE = rct       (3)
BLOCKSIZE = bl         (3)
```

1. The **IOSTAT=** specifier variable is always defined with the most recent error status. If an error occurs during execution of the **INQUIRE** statement then the error status is defined with a positive integer; otherwise the status is that of the most recent input/output statement which referenced that unit.

2. This specifier always becomes defined unless an error condition occurs.

3. This specifier becomes defined with a value only if the specified unit exists and if a file is connected to the unit. Also, no error condition can occur during the execution of the **INQUIRE** statement.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.56 INTEGER Statement

The **INTEGER** statement is a type declaration statement and can be used to declare a name to be of type integer. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type integer, it cannot appear in another type declaration statement.

There are various forms of the **INTEGER** statement. The following sections describe them.

## 2.56.1 Standard INTEGER Statement

```
INTEGER name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

This form is the standard form of the **INTEGER** statement.

*Example:*

```
DIMENSION C(-5:5)
INTEGER A, B(10), C
```

In the previous example, `A` is defined to be a variable of type integer and `B` and `C` are defined to be arrays of type integer.

## 2.56.2 Extended INTEGER Statement: Length Specification

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

***where:***

***name*** is one of the following forms:

```
v[*len]

a[*len](d)

a(d)[*len]
```

***v*** is a variable name, array name, symbolic name of a constant, function name or dummy procedure name.

***a*** is an array name.

***(d)*** is that part of the array declarator defining the dimensions of the array.

***len*** is called the *length specification* and is an unsigned positive integer constant or an integer constant expression enclosed in parentheses whose value is 1, 2 or 4.

This form of the **INTEGER** statement is a Open Watcom FORTRAN 77 extension to the FORTRAN 77 language. The length specification specifies the number of bytes of storage that will be allocated for the name appearing in the **INTEGER** statement. The default length specification is 4. A length specification of 1 or 2 does not change the data type; it merely restricts the magnitude of the integer that can be represented. See the chapter entitled "Names, Data Types and Constants" on page 151 for more information.

The length specification immediately following the word `INTEGER` is the length specification for each entity in the statement not having its own length specification. If a length specification is not specified the default length specification is used. An entity with its own specification overrides the default length specification or the length specification immediately following the word `INTEGER`. Note that for an array the length specification applies to *each* element of the array.

*Example:*

```
DIMENSION C(-5:5)
INTEGER A, B*2(10), C*2
INTEGER*1 X
```

In the previous example, `X` is declared to be a variable of type integer and occupying 1 byte of storage, `A` is declared to be a variable of type integer and occupying 4 bytes of storage and `B` and `C` are declared to be arrays of type integer with *each* element of the array occupying 2 bytes.

## 2.56.3 Extended INTEGER Statement: Data Initialization

```
INTEGER[*len[,]] name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***len*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **INTEGER** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
INTEGER A/100/, B(10)/10*0/
```

In the previous example, `A` is initialized with the integer constant `100` and each element of the array `B` is initialized with the integer constant `0`.

# 2.57 INTRINSIC Statement

```
INTRINSIC f [,f] ...
```

***where:***

*f* is the name of an intrinsic function name.

An **INTRINSIC** statement is used to identify a symbolic name as the name of an intrinsic function. It also allows a specific intrinsic function to be passed as an actual argument. The names of intrinsic functions for type conversion (INT, IFIX, HFIX, IDINT, FLOAT, DFLOAT, SNGL, REAL, DREAL, DBLE, CMPLX, DCMPLX, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT), for choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1), as well as ALLOCATED, ISIZEOF and LOC, must not be used as actual arguments.

A generic intrinsic function does not lose its generic property if it appears in an **INTRINSIC** statement.

A name must only appear in an **INTRINSIC** statement once. A symbolic name must not appear in both an **INTRINSIC** and an **EXTERNAL** statement in a program unit.

*Example:*

```
INTRINSIC SIN
     .
     .
     .
CALL SAM( SIN )
```

In the previous example, the intrinsic function `SIN` was passed to the subroutine `SAM.` If the **INTRINSIC** statement were eliminated then the variable `SIN` would be passed to the subroutine `SAM.`

# 2.58 LOGICAL Statement

The **LOGICAL** statement is a type declaration statement and can be used to declare a name to be of type logical. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type logical, it cannot appear in another type declaration statement.

There are various forms of the **LOGICAL** statement. The following sections describe them.

## 2.58.1 Standard LOGICAL Statement

```
LOGICAL name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

This form is the standard form of the **LOGICAL** statement.

*Example:*

```
DIMENSION C(-5:5)
LOGICAL A, B(10), C
```

In the previous example, `A` is defined to be a variable of type logical and `B` and `C` are defined to be arrays of type logical.

## 2.58.2 Extended LOGICAL Statement: Length Specification

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

***where:***

***name*** is one of the following forms:

```
v[*len]

a[*len](d)

a(d)[*len]
```

***v*** is a variable name, array name, symbolic name of a constant, function name or dummy procedure name.

***a*** is an array name.

***(d)*** is that part of the array declarator defining the dimensions of the array.

***len*** is called the *length specification* and is an unsigned positive integer constant or an integer constant expression enclosed in parentheses whose value is 1 or 4.

This form of the **LOGICAL** statement is a Open Watcom FORTRAN 77 extension to the FORTRAN 77 language. The length specification specifies the number of bytes of storage that will be allocated for the name appearing in the **LOGICAL** statement. The default length specification is 4. A length specification of 1 only changes the storage requirement from 4 bytes to 1 byte; the values of true and false can be represented regardless of the length specification.

The length specification immediately following the word `LOGICAL` is the length specification for each entity in the statement not having its own length specification. If a length specification is not specified the default length specification is used. An entity with its own specification overrides the default length specification or the length specification immediately following the word `LOGICAL`. Note that for an array the length specification applies to *each* element of the array.

*Example:*

```
DIMENSION C(-5:5)
LOGICAL A, B*1(10), C*1
LOGICAL*4 X
```

In the previous example, `X` is declared to be a variable of type logical and occupying 4 bytes of storage, `A` is declared to be a variable of type logical and occupying 4 bytes of storage and `B` and `C` are declared to be arrays of type logical with *each* element of the array occupying 1 byte.

## 2.58.3 Extended LOGICAL Statement: Data Initialization

```
LOGICAL[*len[,]] name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***len*** is as described in the previous section.

***cl*** is a list of the form:

`k [,k] ...`

***k*** is one of the forms:

`c`

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **LOGICAL** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
LOGICAL A/.TRUE./, B(10)/10*.FALSE./
```

In the previous example, `A` is initialized with the logical constant `.TRUE.` and each element of the array `B` is initialized with the logical constant `.FALSE.`.

# 2.59 LOOP Statement

```
LOOP    [:block-label]
```

The **LOOP** statement is used in conjunction with the structured **END LOOP** or **UNTIL** statement. The **LOOP** statement marks the beginning of a sequence of statements which are to be repeated. The **END LOOP** or **UNTIL** statement marks the end of the loop. The LOOP-block is executed until control is transferred out of the LOOP-block or the logical expression (or integer arithmetic expression) of the **UNTIL** statement has a true (or non-zero) value.

The **QUIT** statement may be used to transfer control out of a LOOP-block.

*Example:*

```
LOOP
    READ *, X
    IF( X .GT. 99.0 ) QUIT
    PRINT *, X
END LOOP
```

*Example:*

```
X = 1.0
LOOP
    PRINT *, X, SQRT( X )
    X = X + 1.0
UNTIL( X .GT. 10.0 )
```

An optional block label may be specified with the **LOOP** statement.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.60 MAP Statement

```
MAP
```

The **MAP** statement is used in conjunction with the **END MAP** declarative statement. The **MAP** statement marks the start of a memory mapping structure. A **MAP** structure must appear within a **UNION** block. Any number of variables of any type may appear within a memory map. At least two **MAP** structures must appear within a **UNION** block. A **UNION** block permits the mapping of the same storage in several different ways.

The following example maps out a 4-byte integer on an Intel 80x86-based processor.

*Example:*

```
STRUCTURE /MAPINT/
    UNION
        MAP
            INTEGER*4 LONG
        END MAP
        MAP
            INTEGER*2 LO_WORD
            INTEGER*2 HI_WORD
        END MAP
        MAP
            INTEGER*1 BYTE_0
            INTEGER*1 BYTE_1
            INTEGER*1 BYTE_2
            INTEGER*1 BYTE_3
        END MAP
    END UNION
END STRUCTURE

RECORD /MAPINT/ I

I%LONG = '01020304'x
PRINT '(2Z4)', I%LO_WORD, I%HI_WORD
END
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.61 NAMELIST Statement

```
NAMELIST /name/ vlist [[,]/name/ vlist] ...
```

***where:***

***name*** is the name, enclosed in slashes, of a group of variables. It may not be the same as a variable or array name.

***vlist*** is a list of variable names and array names separated by commas.

The **NAMELIST** statement is used to declare a group name for a set of variables so that they may be read or written with a single namelist-directed **READ**, **WRITE**, or **PRINT** statement.

The list of variable or array names belonging to a **NAMELIST** name ends with a new **NAMELIST** name enclosed in slashes or with the end of the **NAMELIST** statement. The same variable name may appear in more than one namelist.

A dummy variable, dummy array name, or allocatable array may not appear in a **NAMELIST** list. Also, a variable whose type is a user-defined structure may not appear in a **NAMELIST** list.

The **NAMELIST** statement must precede any statement function definitions and all executable statements. A **NAMELIST** name must be declared in a **NAMELIST** statement and may be declared only once. The name may appear only in input/output statements. The **READ**, **WRITE**, and **PRINT** statements may be used to transmit data between a file and the variables specified in a namelist.

*Example:*

```
          CHARACTER*20 NAME
          CHARACTER*20 STREET
          CHARACTER*15 CITY
          CHARACTER*20 STATE
          CHARACTER*20 COUNTRY
          CHARACTER*10 ZIP_CODE
          INTEGER AGE
          INTEGER MARKS(10)
          NAMELIST /PERSON/ NAME, STREET, CITY, STATE,
         +                  COUNTRY, ZIP_CODE, AGE, MARKS

          OPEN( UNIT=1, FILE='PEOPLE' )
          LOOP
              READ( UNIT=1, FMT=PERSON, END=99 )
              WRITE( UNIT=6, FMT=PERSON )
          ENDLOOP
    99    CLOSE( UNIT=1 )
          END
```

The following example shows another form of a namelist-directed **READ** statement.

*Example:*

```
      CHARACTER*20 NAME
      CHARACTER*20 STREET
      CHARACTER*15 CITY
      CHARACTER*20 STATE
      CHARACTER*20 COUNTRY
      CHARACTER*10 ZIP_CODE
      INTEGER AGE
      INTEGER MARKS(10)
      NAMELIST /PERSON/ NAME, STREET, CITY, STATE,
     +                  COUNTRY, ZIP_CODE, AGE, MARKS

      READ PERSON
      PRINT PERSON
      END
```

The input data must be in a special format. The first character in each record must be blank. The second character in the first record of a group of data records must be an ampersand (&) or dollar sign ($) immediately followed by the **NAMELIST** name. The **NAMELIST** name must be followed by a blank and must not contain any imbedded blanks. This name is followed by data items separated by commas. The end of a data group is signaled by the character "&" or "$", optionally followed by the string "END". If the "&" character was used to start the group, then it must be used to end the group. If the "$" character was used to start the group, then it must be used to end the group.

The form of the data items in an input record is:

***Name = Constant***

The name may be a variable name or an array element name. The constant may be integer, real, complex, logical or character. Logical constants may be in the form "T" or ".TRUE" and "F" or ".FALSE". Character constants must be contained within apostrophes. Subscripts must be of integer type.

***ArrayName = Set of Constants***

The set of constants consists of constants of the type integer, real, complex, logical or character. The constants are separated by commas. The number of constants must be less than or equal to the number of elements in the array. Successive occurrences of the same constant may be represented in the form `r*constant,` where `r` is a non-zero integer constant specifying the number of times the constant is to occur.

The variable and array names specified in the input file must appear in the **NAMELIST** list, but the order is not important. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the **NAMELIST** list. The list can contain names of items in **COMMON** but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in name or constants. Trailing blanks after integers and exponents are treated as zeros.

*Example:*

```
&PERSON
   NAME = 'John Doe'
   STREET = '22 Main St.' CITY = 'Smallville'
   STATE = 'Texas'        COUNTRY = 'U.S.A.'
   ZIP_CODE = '78910-1203'
   MARKS = 73, 82, 3*78, 89, 2*93, 91, 88
   AGE = 23
&END
```

The form of the data items in an output record is suitable for input using a namelist-directed **READ** statement.

1. Output records are written using the ampersand character (&), not the dollar sign ($), although the dollar sign is accepted as an alternative during input. That is, the output data is preceded by "&name" and is followed by "&END".

2. All variable and array names specified in the **NAMELIST** list and their values are written out, each according to its type.

3. Character data is enclosed in apostrophes.

4. The fields for the data are made large enough to contain all the significant digits.

5. The values of a complete array are written out in columns.

For more information, see the chapters entitled "Format" on page 225 and "Input/Output" on page 215.

# 2.62 OPEN Statement

```
OPEN (oplist)
```

***where:***

***oplist*** must include the **UNIT=** specifier and may include at most one of each of the open specifiers listed below. Specifiers are separated by commas.

```
[UNIT =] u
IOSTAT = ios
ERR = s
FILE = fin
STATUS = sta
ACCESS = acc
FORM = fm
RECL = rcl
BLANK = blnk
```

As an extension to the FORTRAN 77 language, the following inquiry specifiers are also supported.

```
ACTION = act
CARRIAGECONTROL = cc
RECORDTYPE = rct
BLOCKSIZE = bl
SHARE = shr
```

The **OPEN** statement may be used to connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain specifications of a connection between a file and a unit.

***Open Specifiers***

***[UNIT =] u***

`u` is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional **UNIT=** specifier is omitted then the specifier must be the first item in the list of specifiers.

***IOSTAT = ios***

is an *input/output status specifier.* The integer variable or integer array element `ios` is defined with zero if no error condition exists or a positive integer value if an error condition exists.

***ERR = s***

is an *error specifier* and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s`.

***FILE = fin***

`fin` is a character expression whose value when trailing blanks are removed is the name of a file to be connected to the specified unit. If this specifier is omitted and the unit is not connected to a file, it becomes connected to a file determined by Open Watcom FORTRAN 77. The name established by Open Watcom FORTRAN 77 is described in the section entitled "Units" on page 219 of the chapter entitled "Input/Output"

***STATUS = sta***

`sta` is a character expression whose value when trailing blanks are removed is `'OLD'`, `'NEW'`, `'SCRATCH'`, or `'UNKNOWN'`.

***OLD*** When `OLD` is specified, a **`FILE=`** specifier must be given. The file must exist.

***NEW*** When `NEW` is specified, a **`FILE=`** specifier must be given. The file must not exist. Successful execution of the **`OPEN`** statement creates the file and changes the status to `OLD`.

***SCRATCH*** `SCRATCH` may only be specified for an unnamed file (i.e. **`FILE=`** is not allowed). When the file is closed, it is deleted.

***UNKNOWN*** If `UNKNOWN` is specified, the status is ignored. If the **`STATUS=`** specifier is omitted then `UNKNOWN` is assumed.

***ACCESS = acc***

`acc` is a character expression whose value when trailing blanks are removed is `'SEQUENTIAL'` or `'DIRECT'`. It specifies the access method for the file. If the **`ACCESS=`** specifier is omitted then `'SEQUENTIAL'` is assumed. If the file exists then the access method must be in the set of allowed access methods for the file. If the file does not exist then the file is created with a set of allowed access methods that includes the specified access method.

Open Watcom FORTRAN 77 also supports access `'APPEND'` which is a form of sequential access in which the file is positioned at the endfile record. The file must exist or the append access method must be in the set of allowed access methods for the file. In all other respects, the file is treated as if `'SEQUENTIAL'` had been specified.

***FORM = fm***

`fm` is a character expression whose value when trailing blanks are removed is `'FORMATTED'` or `'UNFORMATTED'`. It specifies that the file is being connected for formatted or unformatted input/output. If the **`FORM=`** specifier is omitted and the file is being connected for direct access then `'UNFORMATTED'` is assumed. If the **`FORM=`** specifier is omitted and the file is being connected for sequential access then `'FORMATTED'` is assumed. If the file exists then the specified form must be included in the set of allowed forms for the file. If the file does not exist then the file is created with a set of allowed forms that includes the specified form.

***RECL = rcl***

`rcl` is an integer expression whose value must be positive. It specifies the length of each record in a file being connected for direct access. If the file is being connected for direct access, this specifier must be given; otherwise it must be omitted. Open Watcom FORTRAN 77 allows the **`RECL=`** specifier for files opened for sequential access.

***BLANK = blnk***

`blnk` is a character expression whose value when trailing blanks are removed is `'NULL'` or `'ZERO'`. If `'NULL'` is specified then all blank characters in numeric formatted input fields are ignored except that an entirely blank field has a value of zero. If `'ZERO'` is specified then all blank characters other than leading blanks are treated as zeroes. If this specifier is omitted then `'NULL'` is assumed. This specifier may only be present for a file being connected for formatted input/output.

***ACTION = act***

`act` is a character expression whose value when trailing blanks are removed is `'READ'`, `'WRITE'` or `'READWRITE'`. If `'READ'` is specified, data can only be read from the file. If `'WRITE'` is specified, data can only be written to the file. If `'READWRITE'` is specified, data can both be read and written. The default is `'READWRITE'`.

***CARRIAGECONTROL = cc***

`cc` is a character expression whose value when trailing blanks are removed is `'YES'`, or `'NO'`. If `'YES'` is specified, Open Watcom FORTRAN 77 will automatically add an extra character at the beginning of each record. This character will be interpreted as a carriage control character. If `'NO'` is specified, records will be written to the file without adding a carriage control character at the beginning of the record. The default is `'NO'`.

***RECORDTYPE = rct***

`rct` is a character expression whose value when trailing blanks are removed specifies the type of record (or record structure) to be used for the file. The allowed values for `rct` depend on the system on which you are running the program. See the compiler User's Guide for a discussion of the **`RECORDTYPE=`** specifier.

***BLOCKSIZE = bl***

`bl` is an integer expression whose value specifies the internal buffer size to be used for file input/output. The allowed values for `bl` depend on the system on which you are running the program. Generally, the larger the buffer, the faster the input/output. See the compiler User's Guide for a discussion of the **`BLOCKSIZE=`** specifier.

***SHARE = shr***

`shr` is a character expression whose value when trailing blanks are removed specifies the way in which other processes can simultaneously access the file. The allowed values for `shr` depend on the system on which you are running the program. See the compiler User's Guide for a discussion of the **`SHARE=`** specifier.

*Example:*

```
      OPEN( UNIT=1, FILE='TEST', STATUS='UNKNOWN',
     +      ACCESS='SEQUENTIAL',
     +      FORM='FORMATTED', BLANK='ZERO' )
```

In the above example, the file `'TEST'`, containing FORMATTED records, is connected to unit 1. The status of the file is `'UNKNOWN'` since we do not know if it already exists. We will access the file sequentially, using formatted input/output statements. Blanks in numeric input data are to be treated as zeroes.

*Notes:*

1. If the unit is already connected to a file that exists, the execution of an **OPEN** statement for that unit is permitted.

   **(a)** If the same file is opened then only the **BLANK=** specifier may be different. The same file is opened if no **FILE=** specifier was given or if the **FILE=** specifier refers to the same file.

   **(b)** If a different file is opened then the currently connected file is automatically closed.

2. If the file to be connected to the unit does not exist, but is already preconnected to the unit, any properties specified in the **OPEN** statement are merged with and supersede those of the preconnection. For example, the **RECL=** specification will override the record length attribute defined by a preconnection of the file.

3. The same file may not be connected to two or more different units.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.63 OTHERWISE Statement

```
OTHERWISE
```

The **OTHERWISE** statement is used in conjunction with the **SELECT** statement. The **OTHERWISE** statement marks the start of a new **CASE** block which is a series of zero or more statements ending in an **END SELECT** statement.

When this statement is used and the value of a *case expression* is not found in any *case list* then control of execution is transferred to the first executable statement following the **OTHERWISE** statement.

The **CASE DEFAULT** statement may be used in place of the **OTHERWISE** statement.

*Example:*

```
SELECT CASE ( CH )
CASE ( 'a' : 'z' )
    PRINT *, 'Lower case letter'
CASE ( 'A' : 'Z' )
    PRINT *, 'Upper case letter'
CASE ( '0' : '9' )
    PRINT *, 'Digit'
OTHERWISE
    PRINT *, 'Special character'
END SELECT
```

In the above example, if the character `CH` is not a letter or digit then the **OTHERWISE** block is executed.

***Note:*** The **OTHERWISE** or **CASE DEFAULT** block must follow all other **CASE** blocks.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.64 PARAMETER Statement

```
PARAMETER (p=e [,p=e] ...)
```

***where:***

***p*** is a symbolic name.

***e*** is a constant expression. Refer to the chapter entitled "Expressions" on page 173 for more information.

p is known as a symbolic constant whose value is determined by the value of the expression e according to the rules of assignment as described in the chapter entitled "Assignment Statements" on page 187. Any symbolic constant appearing in expression e must have been previously defined in the same or a previous **PARAMETER** statement in the same program unit. A symbolic constant may not be defined more than once in a program unit.

If the symbolic name p is of type integer, real, double precision or complex then the corresponding expression e must be an arithmetic constant expression (see the chapter entitled "Expressions" on page 173). If the symbolic name p is of type character or logical then the expression e must be a character constant expression or a logical constant expression respectively (see the chapter entitled "Expressions" on page 173).

*Example:*

```
PARAMETER (PI=3.14159,BUFFER=80,PIBY2=PI/2)
PARAMETER (ERRMSG='AN ERROR HAS OCCURRED')
```

If a symbolic constant is not of default implied type, its type must be specified in an **IMPLICIT** statement or a type statement before its occurrence in a **PARAMETER** statement. Similarly, if the length of a character symbolic constant is not the default length of 1, its length must be specified in an **IMPLICIT** statement or a type statement before its occurrence in a **PARAMETER** statement.

# 2.65 PAUSE Statement

```
PAUSE [n]
```

***where:***

***n*** is a character constant or an unsigned integer constant of no more than five digits.

Open Watcom FORTRAN 77 allows `n` to be any unsigned integer constant.

Execution of a **PAUSE** statement causes a cessation of execution of the program. Execution of the program may be resumed by the program operator by pressing the terminal line entering key (e.g., ENTER or RETURN). The **PAUSE** statement may appear in any program unit.

If the Open Watcom FORTRAN 77 debugger was requested then execution of the **PAUSE** statement will cause entry into the debugger. Program execution may be resumed by issuing the debugger "go" command.

*Example:*

```
PAUSE 4341
```

The four digit number `4341` is displayed on the terminal. The program temporarily ceases execution. Execution is resumed by pressing the terminal line entering key.

*Example:*

```
PAUSE 'Ready the paper and then resume execution'
```

The character string

```
Ready the paper and then resume execution
```

is displayed on the terminal. Execution of the program may be resumed.

# 2.66 PRINT Statement

Two forms of the **PRINT** statement are supported by Open Watcom FORTRAN 77.

## 2.66.1 Standard PRINT Statement

```
PRINT f [,olist]
```

***where:***

***f*** is a format identifier.

***olist*** is an optional output list.

## 2.66.2 Extended PRINT Statement

```
PRINT, olist
```

***where:***

***olist*** is an output list.

## 2.66.3 Description of PRINT Statement

The **PRINT** statement is used to transfer data from the executing FORTRAN program to an external device or file.

*Format Identifier* - A format identifier is one of the following:

1. A statement label of a **FORMAT** statement that appears in the same program unit as the format identifier.
2. An integer variable name that has been assigned the statement label of a **FORMAT** statement that appears in the same program unit as the format identifier (see the **ASSIGN** statement).
3. An integer array name.
4. A character array name.
5. Any character expression except one involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant (see the **PARAMETER** statement).
6. An asterisk `(*),` indicating *list-directed* formatting.

   Open Watcom FORTRAN 77 supports a variation of *list-directed* formatting in which the asterisk `(*)` may be omitted. It is equivalent to

   ```
   PRINT * [,olist]
   ```

7. A **NAMELIST** name, indicating *namelist-directed* formatting.

*Output list* - An output list may contain one or more of the following:

1. A variable name.
2. An array element name.
3. A character substring name.
4. An array name except an assumed-size dummy array.
5. Any other expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbol name of a constant (since the length can be determined at compile time).
6. An implied-DO list of the form:

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

   where `dlist` is composed of one or more of items (1) through (6).

*Example:*

```
CHARACTER*5 S
COMPLEX C
S = 'Hello'
I = 123
X = 12.5
C = (12.5,4.58)
PRINT *, S, I, X, C
END
```

The above example illustrates *list-directed* formatting using the **PRINT** statement. The asterisk specifies that the variables in the output list are to be displayed in some format that is appropriate to the type of the variable (hence the term "list-directed"). The CHARACTER variable `S` is printed using a suitable `A` format descriptor. The INTEGER variable `I` is printed using a suitable `I` format descriptor. The REAL variable `X` is printed using a suitable `G` format descriptor. The COMPLEX variable `C` is printed using a suitable `G` format descriptor and is displayed with enclosing parentheses and a comma. Output from the above program would resemble the following.

```
Hello    123    12.5000000      (12.5000000,4.5799999)
```

*Example:*

```
CHARACTER*5 S
COMPLEX C
S = 'Hello'
I = 123
X = 12.5
C = (12.5,4.58)
PRINT, S, I, X, C
END
```

The above example illustrates a Open Watcom FORTRAN 77 extension of *list-directed* formatting using the **PRINT** statement. The asterisk is omitted but the results are exactly the same as in the previous example.

*Example:*

```
          PRINT 100, X, Y, Z
    100   FORMAT( 3F10.5 )
          PRINT '( 3F10.5 )', X, Y, Z
```

The above gives two examples of the **PRINT** statement. In both cases, the format conversion is identical but it was specified in different ways. When executed, the effect of both **PRINT** statements is the same.

*Example:*

```
          PRINT '(1X,100A1)', ('*',I=1,J)
```

The above example illustrates a technique for producing histograms using the implied DO-loop. Each time this statement is executed, a number of asterisks are printed, depending on the value of `J`.

*Notes:*

1. The **PRINT** statement is implicitly a formatted output statement.

2. The unit number that is *implicitly* used in the **PRINT** statement is unit number 6.

3. If no output list is specified then the effect of the **PRINT** statement is to produce one or more records whose characters are all blanks.

4. FORTRAN 77 leaves the format of output in list-directed formatting to the discretion of Open Watcom FORTRAN 77. Hence other FORTRAN compilers may produce different results. If the format of output must be consistent from one compiler to the next then list-directed formatting should *not* be used.

5. An implication of point (6) above is that nesting of implied-DO lists is permitted. For example, the output list

   ```
   ( (A(I,J), B(I,J), J = 1, 5), I = 1, 10 )
   ```

   may be broken down into the following components:

   ```
      A(I,J), B(I,J)
     (....dlist1...., J = 1, 5)
   ( .....dlist2.............., I = 1, 10 )
   ```

For more information on input/output, see the chapter entitled "Input/Output" on page 215. For more information on formatted input/output, see the chapter entitled "Format" on page 225.

# 2.67 PROGRAM Statement

```
PROGRAM pgm
```

***where:***

***pgm*** is the symbolic name of the main program.

A **PROGRAM** statement is optional in an executable program. If it does appear, it must be the first statement in the main program.

*Example:*

```
PROGRAM CALC
     .
     .
     .
CALL COMPUTE
     .
     .
     .
END
```

The main program can contain any Open Watcom FORTRAN 77 statement except a **FUNCTION**, **SUBROUTINE**, **BLOCK DATA**, **RETURN** or **ENTRY** statement. Note that a **SAVE** statement is allowed but has no effect in the main program.

# 2.68 QUIT Statement

```
QUIT    [: block-label]
```

The **QUIT** statement may be used to cause a transfer of control to the first executable statement that follows the terminal statement of the block which contains it. Examples of such terminal statements are **ADMIT**, **CASE**, **END DO**, **END LOOP**, **END WHILE**, **UNTIL**, etc. If `block-label` is present then control is transferred out of the block identified by that block label. The **QUIT** statement is an extension to the FORTRAN 77 language.

*Example:*

```
LOOP
    WRITE( UNIT=*, FMT='(A)' ) 'Enter a number'
    READ( UNIT=*, FMT='(F10.4)', IOSTAT=IOS ) X
    IF( IOS .NE. 0 ) QUIT
    IF( X .LT. 0 ) QUIT
    PRINT *, X, SQRT( X )
END LOOP
END
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.69 READ Statement

Three forms of the **READ** statement are supported by Open Watcom FORTRAN 77.

## 2.69.1 Standard READ Statement

```
READ (cilist) [ilist]
READ f [,ilist]
```

***where:***

***cilist*** is a control information list of specifiers separated by commas:

```
[UNIT =] u
[FMT =] f
REC = rn
IOSTAT = ios
ERR = s
END = s
```

***f*** is a format identifier.

***ilist*** is an optional input list.

## 2.69.2 Extended READ Statement

```
READ, ilist
```

***where:***

***ilist*** is an input list.

## 2.69.3 Description of READ Statement

The **READ** statement is used to transfer data from a device or file into the executing FORTRAN program. As shown above, Open Watcom FORTRAN 77 supports three forms of the **READ** statement.

***Control Information List and Format Identifier***

***[UNIT =] u***

`u` is an *external unit identifier* or an *internal file identifier.*

1. An external unit identifier is a non-negative integer expression or an asterisk `(*)` in which case unit 5 is assumed.

2. An internal file identifier is the name of a character variable, character array, character array element, or character substring.

If the optional **UNIT=** specifier is omitted then the unit specifier must be the first item in the list of specifiers.

***[FMT =] f***

`f` is a *format identifier*. A format identifier is one of the following:

1. A statement label of a **FORMAT** statement that appears in the same program unit as the format identifier.
2. An integer variable name that has been assigned the statement label of a **FORMAT** statement that appears in the same program unit as the format identifier (see the **ASSIGN** statement).
3. An integer array name.
4. A character array name.
5. Any character expression except one involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant (see the **PARAMETER** statement).
6. An asterisk `(*),` indicating *list-directed* formatting.

   Open Watcom FORTRAN 77 supports a third form of the **READ** statement in which the asterisk `(*)` may be omitted. This is a form of *list-directed* formatting in which unit 5 is assumed. It is equivalent to

   ```
   READ * [,ilist]
   ```

7. A **NAMELIST** name, indicating *namelist-directed* formatting.

If the optional **FMT=** specifier is omitted then the format specifier must be the second item in the list of specifiers and **UNIT=** must not be specified for the first item in the list.

***REC = rn***

`rn` is an integer expression whose value must be positive. It is the number of the record to be read when a file is connected for direct access.

***IOSTAT = ios***

is an input/output status specifier. The integer variable or integer array element `ios` is defined with zero if no error occurs, a positive integer value if an error occurs, or a negative integer value if an end-of-file occurs.

***ERR = s***

is an error specifier and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s.`

***END = s***

is an end-of-file specifier and `s` is a statement label. When an end-of-file occurs, execution is transferred to the statement labelled by `s.`

*Input list* - An input list may contain one or more of the following:

1. A variable name.

2. An array element name.

3. A character substring name.

4. An array name except an assumed-size dummy array.

5. An implied-DO list of the form:

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

where `dlist` is composed of one or more of items (1) through (5).

*Example:*

```
          READ( 5, 100 )X, Y, Z
          READ( UNIT=5, FMT=100 )X, Y, Z
100       FORMAT( 3F10.5 )
          READ( UNIT=5, FMT='( 3F10.5 )' )X, Y, Z
          READ( 5, '( 3F10.5 )' )X, Y, Z
```

The above gives four examples of formatted **READ** statements, using the first of three supported forms of the **READ** statement. In all cases, the format conversion is identical but it was specified in different ways. When executed, the effect of all **READ** statements is the same. The unit number that is *explicitly* used in this form of the **READ** statement is unit number 5. There are, in fact, many other ways in which the **READ** statement could have been written, all of which would have the same effect when executed. We have not shown the use of all the specifiers.

*Example:*

```
          READ 100, X, Y, Z
100       FORMAT( 3F10.5 )
          READ '( 3F10.5 )', X, Y, Z
```

The above gives two examples of formatted **READ** statements, using the second of three supported forms of the **READ** statement. In both cases, the format conversion is identical but it was specified in different ways. When executed, the effect of both **READ** statements is the same. The unit number that is *implicitly* used in this form of the **READ** statement is unit number 5.

*Example:*

```
          READ( 5, * )X, Y, Z
          READ( *, * )X, Y, Z
          READ( UNIT=5, FMT=* )X, Y, Z
          READ( UNIT=*, FMT=* )X, Y, Z
          READ *, X, Y, Z
          READ  , X, Y, Z
```

The above six examples of list-directed formatted input are all equivalent. Open Watcom FORTRAN 77 assumes unit 5 when the unit number identifier is an asterisk (as in the second and fourth examples). In the fifth example, the asterisk is a format identifier indicating list-directed formatting. The fifth and sixth examples are examples of the second and third forms, respectively, of the **READ** statement in which Open Watcom FORTRAN 77 assumes unit 5. When the format identifier is an asterisk or when the third form of the **READ** statement is used, we call this list-directed *list-directed* formatting.

*Example:*

```
READ( 8 )X, Y, Z
READ( UNIT=8 )X, Y, Z
```

The above gives two examples of unformatted **READ** statements. The unit number used in the example is 8. When executed, the effect of both of these statements is the same. The values of the variables `X, Y` and `Z` are read from the file connected to unit 8. The values are stored in the file in their binary form (a form quite incomprehensible to most human beings). An advantage to using this particular form of the **READ** statement is that no conversion is required between the internal binary representation of the values and their textual (human-readable) form (which means it takes less computer time to process the data).

*Notes:*

1. The **REC=** specifier may not be used when list-directed output is specified.

2. If no input list is specified then the effect of the **READ** statement is to skip one or more records in the file.

3. An implication of point (5) above is that nesting of implied-DO lists is permitted. For example, the input list

   ```
   ( (A(I,J), B(I,J), J = 1, 5), I = 1, 10 )
   ```

   may be broken down into the following components:

   ```
      A(I,J), B(I,J)
     (....dlist1....., J = 1, 5)
   ( .....dlist2..............., I = 1, 10 )
   ```

For more information on input/output, see the chapter entitled "Input/Output" on page 215. For more information on formatted input/output, see the chapter "Format" on page 225.

# 2.70 REAL Statement

The **REAL** statement is a type declaration statement and can be used to declare a name to be of type real. The implicit type of the name, whether defined by the "first letter rule" (see the chapter entitled "Names, Data Types and Constants" on page 151) or by an **IMPLICIT** statement, is either confirmed or overridden. However, once a name has been declared to be of type real, it cannot appear in another type declaration statement.

There are various forms of the **REAL** statement. The following sections describe them.

## 2.70.1 Standard REAL Statement

```
REAL name [,name] ...
```

***where:***

***name*** is a variable name, array name, array declarator, symbolic name of a constant, function name or dummy procedure name.

This form is the standard form of the **REAL** statement.

*Example:*

```
DIMENSION C(-5:5)
REAL A, B(10), C
```

In the previous example, `A` is defined to be a variable of type real and `B` and `C` are defined to be arrays of type real.

## 2.70.2 Extended REAL Statement: Length Specification

```
REAL[*len[,]] name [,name] ...
```

***where:***

***name*** is one of the following forms:

```
v[*len]

a[*len](d)

a(d)[*len]
```

***v*** is a variable name, array name, symbolic name of a constant, function name or dummy procedure name.

***a*** is an array name.

***(d)*** is that part of the array declarator defining the dimensions of the array.

***len*** is called the *length specification* and is an unsigned positive integer constant or an integer constant expression enclosed in parentheses whose value is 4 or 8.

This form of the **REAL** statement is a Open Watcom FORTRAN 77 extension to the FORTRAN 77 language. The length specification specifies the number of bytes of storage that will be allocated for the name appearing in the **REAL** statement. The default length specification is 4. A length specification of 8 specifies that the data type of the name appearing in the **REAL** statement is to be double precision.

The length specification immediately following the word `REAL` is the length specification for each entity in the statement not having its own length specification. If a length specification is not specified the default length specification is used. An entity with its own specification overrides the default length specification or the length specification immediately following the word `REAL`. Note that for an array the length specification applies to *each* element of the array.

*Example:*

```
DIMENSION C(-5:5)
REAL A, B*8(10), C*8
REAL*8 X
```

In the previous example, `X` is declared to be a variable of type double precision, `A` is declared to be a variable of type real and `B` and `C` are declared to be arrays of type double precision.

## 2.70.3 Extended REAL Statement: Data Initialization

```
REAL[*len[,]] name [/cl/] [,name[/cl/]] ...
```

***where:***

***name*** is as described in the previous section.

***len*** is as described in the previous section.

***cl*** is a list of the form:

```
k [,k] ...
```

***k*** is one of the forms:

```
c
```

`r*c` (equivalent to `r` successive appearances of `c`)

***c*** is a constant or the symbolic name of a constant

***r*** is an unsigned positive integer constant or the symbolic name of a constant.

This form of the **REAL** statement is an extension to the FORTRAN 77 language. The rules for data initialization are the same as for the **DATA** statement.

*Example:*

```
REAL A/1.2/, B(10)/10*5.0/
```

In the previous example, `A` is initialized with the real constant `1.2` and each element of the array `B` is initialized with the real constant `5.0`.

# 2.71 RECORD Statement

```
RECORD /typename/ name [,name] ...
```

***where:***

***typename*** is the name of a user-defined structure type.

***name*** is a variable name, array name, array declarator, function name or dummy procedure name.

The **RECORD** statement is used to assign a structure type to a variable.

*Example:*

```
STRUCTURE /ADDRESS/
    CHARACTER*20 STREET
    CHARACTER*15 CITY
    CHARACTER*20 STATE
    CHARACTER*20 COUNTRY
    CHARACTER*10 ZIP_CODE
END STRUCTURE

STRUCTURE /PEOPLE/
    CHARACTER*20 NAME
    RECORD /ADDRESS/ ADDR
    INTEGER*2 AGE
END STRUCTURE

RECORD /PEOPLE/ CUSTOMER

CUSTOMER%NAME = 'John Doe'
CUSTOMER%ADDR%STREET = '22 Main St.'
CUSTOMER%ADDR%CITY = 'Smallville'
CUSTOMER%ADDR%STATE = 'Texas'
CUSTOMER%ADDR%COUNTRY = 'U.S.A.'
CUSTOMER%ADDR%ZIP_CODE = '78910-1203'
CUSTOMER%AGE = 23
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.72 REMOTE BLOCK Statement

```
REMOTE BLOCK name
```

***where:***

***name*** is a valid FORTRAN symbolic name.

The **REMOTE BLOCK** statement is used to define a block of statements which may be executed by an **EXECUTE** statement. A REMOTE-block must be defined in the program unit in which it is used and is terminated by an **END BLOCK** statement. A REMOTE-block is similar in concept to a subroutine, with the advantage that shared variables do not need to be placed in a common block or passed in an argument list. When execution of the REMOTE-block is complete, control returns to the statement following the **EXECUTE** statement which invoked it.

This feature is helpful in avoiding duplication of code for a common sequence of statements required in a number of places throughout a program. It can also be an aid to writing a well structured program. This feature can be mimicked using the **ASSIGN** and assigned **GO TO** statements. However, statement numbers must be introduced which could lead to errors.

Each REMOTE-block must have a different name and it must not be a subprogram or variable name. Note that a REMOTE-block is local to the program unit in which it is defined and may not be referenced (executed) from another program unit.

Note that the nested definition of REMOTE-blocks is not permitted.

*Example:*

```
EXECUTE INCR
PRINT *, 'FIRST'
EXECUTE INCR
PRINT *, 'SECOND'
    .
    .
    .
REMOTE BLOCK INCR
    I=I+1
    PRINT *, 'I=',I
END BLOCK
```

Both **EXECUTE** statements will cause REMOTE-block `INCR` to be executed. That is, variable `I` will be incremented and its value will be printed. When the block has been executed by the first **EXECUTE** statement, control returns to the **PRINT** statement following it and the word `FIRST` is printed. Similarly, when the block is executed by the second **EXECUTE** statement, control returns to the **PRINT** statement following it and the word `SECOND` is printed.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.73 RETURN Statement

```
RETURN [e]
```

***where:***

***e*** is an integer expression.

A **RETURN** statement is used to terminate execution of a subprogram and return control to the program unit that referenced it. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 permits the use of the **RETURN** statement in the main program. When a **RETURN** statement is executed in the main program, program execution terminates in the same manner as the **STOP** or **END** statement.

The expression e is not permitted when returning from an external function subprogram (or main program); it can only be specified when returning from a subroutine subprogram.

*Example:*

```
      FUNCTION ABS( A )
      ABS = A
      IF( A .GE. 0 )RETURN
      ABS = -A
      RETURN
      END
```

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

# 2.74 REWIND Statement

```
REWIND u
REWIND (alist)
```

***where:***

***u*** is an external unit identifier.

***alist*** is a list of rewind specifiers separated by commas:

```
[UNIT =] u
IOSTAT = ios
ERR = s
```

Execution of a **REWIND** statement causes the file connected to the specified unit to be positioned at the beginning (or before the first record) of the file.

***Rewind Specifiers***

***[UNIT =] u***

u is an *external unit identifier.* An external unit identifier is a non-negative integer expression. If the optional UNIT= specifier is omitted then the specifier must be the first item in the list of specifiers.

***IOSTAT = ios***

is an *input/output status specifier.* The integer variable or integer array element ios is defined with zero if no error condition exists or a positive integer value if an error condition exists.

***ERR = s***

is an *error specifier* and s is a statement label. When an error occurs, execution is transferred to the statement labelled by s.

*Example:*

```
        LOOP
            READ( UNIT=7, END=100, FMT=200 )RECORD
            PRINT *, RECORD
        ENDLOOP
100     REWIND( UNIT=7 )
        LOOP
            READ( UNIT=7, END=101, FMT=200 )RECORD
            WRITE( UNIT=8, FMT=200 )RECORD
        ENDLOOP
101     CONTINUE
```

In the previous example, we illustrate how one might process the records in a file twice. After reaching the endfile record, a **REWIND** statement is executed and the file is read a second time.

*Notes:*

1. The unit must be connected for sequential access.
2. If the file is positioned at the beginning of the file then the **REWIND** statement has no effect.
3. It is permissible to rewind a file that does not exist but it has no effect.

For more information on input/output, see the chapter entitled "Input/Output" on page 215.

# 2.75 SAVE Statement

```
SAVE [a [,a] ...]
```

***where:***

***a*** is a named common block preceded and followed by a slash (/), a variable name or an array name.

The **SAVE** statement is used to retain the value of an entity after the execution of a **RETURN** or **END** statement in a subprogram. Upon re-entry to the subprogram, the entity will have the same value it had when exit was made from the subprogram. However, an entity belonging to a common block that has appeared in a **SAVE** statement may become redefined in another program unit.

*Notes:*

1. A name cannot appear in a **SAVE** statement more than once in the same program unit.

2. Dummy arguments, procedure names and names belonging to a common block are not permitted in a **SAVE** statement.

3. A **SAVE** statement with no list is identical to a **SAVE** statement containing all allowable names in a program unit.

4. A common block name appearing in a **SAVE** statement has the same effect of specifying all names belonging to that common block in the **SAVE** statement.

5. If a named common block is specified in a **SAVE** statement in a subprogram, it must be specified in a **SAVE** statement in every subprogram in which that common block appears. Furthermore, upon executing a **RETURN** or **END** statement, the current values of the entities in that common block are made available to the next program unit executed in which that common block appears.

6. If a named common block is specified in a **SAVE** statement in the main program unit, the current values of the entities in that common block are made available to every subprogram that specifies that common block. In this case, a **SAVE** statement has no effect in the subprogram.

In the following example, the subroutine `BLKINIT` initializes the entities of the common block `BLK` and uses a **SAVE** statement to ensure that their values are made available to subroutine `BLKPRT`.

*Example:*

```
PROGRAM MAIN
     .
     .
     .
CALL BLKINIT
CALL BLKPRT
     .
     .
     .
END

SUBROUTINE BLKINIT
COMMON /BLK/ A,B,C
SAVE /BLK/
A = 1.0
B = 2.0
C = 3.0
END

SUBROUTINE BLKPRT
COMMON /BLK/ A,B,C
SAVE /BLK/
PRINT *, A, B, C
END
```

# 2.76 SELECT Statement

```
SELECT [CASE] (e) [FROM]  [: block-label]
```

The **SELECT** statement is used in conjunction with the **CASE** and **END SELECT** statements. The form of a **SELECT** block is as follows:

```
SELECT [CASE] (e) [FROM]  [: block-label]
CASE ( case-list )
    statement (s)
CASE ( case-list )
    statement (s)
    .
    .
    .
CASE ( case-list )
    statement(s)
CASE DEFAULT
    statement(s)
END SELECT
```

***where:***

***e*** is an integer expression.

***case-list*** is a list of one or more *cases* separated by commas. A *case* is either

**(a)** a single integer, logical or character constant expression or

**(b)** an integer, logical or character constant expression followed by a colon followed by another expression or the same type. This form of a case defines a range of values consisting of all integers or characters greater than or equal to the value of the expression preceding the colon and less than or equal to the value of the expression following the colon.

The **CASE** and **FROM** keywords are optional in the **SELECT** statement. An optional block label may be specified with the **SELECT** statement.

The case expression *e* is evaluated and if the result is equal to one of the values covered by `case-list` then the control of execution is transferred to the associated **CASE** block.

*Example:*

```
SELECT CASE ( CH )
CASE ( 'a' : 'z' )
    PRINT *, 'Lower case letter'
CASE ( 'A' : 'Z' )
    PRINT *, 'Upper case letter'
CASE ( '0' : '9' )
    PRINT *, 'Digit'
CASE DEFAULT
    PRINT *, 'Special character'
END SELECT
```

In the above example, if the character `CH` is not a letter or digit then the **`CASE DEFAULT`** block is executed.

The **`CASE DEFAULT`** statement is optional. If it is present and the case expression is out of range (i.e., no **`CASE`** blocks are executed) then the **`CASE DEFAULT`** block is executed. If it is not present and the case expression is out of range then execution continues with the first executable statement following the **`END SELECT`** statement. The **`CASE DEFAULT`** block must follow all other **`CASE`** blocks.

*Example:*

```
SELECT CASE ( I )
CASE ( 1 )
    Y = Y + X
    X = X * 3.2
CASE ( 2 )
    Z = Y**2 + X
CASE ( 3 )
    Y = Y * 13. + X
    X = X - 0.213
CASE ( 4 )
    Z = X**2 + Y**2 - 3.0
    Y = Y + 1.5
    X = X * 32.0
CASE DEFAULT
    PRINT *, 'CASE is not in range'
END SELECT
PRINT *, X, Y, Z
```

In order to retain compatibility with earlier versions of WATCOM FORTRAN 77 compilers, the **`OTHERWISE`** statement may be used in place of the **`CASE DEFAULT`** statement.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.77 STOP Statement

```
STOP [n]
```

***where:***

***n*** is a character constant or an unsigned integer constant of no more than five digits.

Open Watcom FORTRAN 77 allows `n` to be any unsigned integer constant.

Execution of a **`STOP`** statement causes termination of execution of the program. A **`STOP`** statement may appear in any program unit (although good programming practice suggests that the main program is the proper place for this statement).

*Example:*

```
STOP 943
```

The three digit number `943` is displayed on the console prior to program termination.

*Example:*

```
STOP 'Finished at last'
```

The character string

```
Finished at last
```

is displayed on the console prior to program termination.

# 2.78 STRUCTURE Statement

```
STRUCTURE /typename/
```

***where:***

***typename*** is the name for a new, compound variable, data type.

The **STRUCTURE** statement is used in conjunction with the **END STRUCTURE** declarative statement. The **STRUCTURE** statement marks the start of a structure definition.

The **STRUCTURE** statement defines a new variable type, called a *structure.* It does not declare a specific program variable. The **RECORD** statement is used to declare variables and arrays to be of this particular structure type.

Structures may be composed of simple FORTRAN types or more complex structure types. This is shown in the following example.

*Example:*

```
STRUCTURE /ADDRESS/
    CHARACTER*20 STREET
    CHARACTER*20 CITY
    CHARACTER*20 STATE
    CHARACTER*20 COUNTRY
    CHARACTER*10 ZIP_CODE
END STRUCTURE

STRUCTURE /PEOPLE/
    CHARACTER*20 NAME
    RECORD /ADDRESS/ ADDR
    INTEGER*2 AGE
END STRUCTURE

RECORD /PEOPLE/ CUSTOMER
```

Element names are local to the structure in which they appear. The same element name can appear in more than one structure. Nested structures may have elements with the same name. A particular element is specified by listing the sequence of elements required to reach the desired element, separated by percent symbols (%) or periods (.).

*Example:*

```
CUSTOMER%NAME = 'John Doe'
CUSTOMER%ADDR%STREET = '22 Main St.'
CUSTOMER%ADDR%CITY = 'Smallville'
CUSTOMER%ADDR%STATE = 'Texas'
CUSTOMER%ADDR%COUNTRY = 'U.S.A.'
CUSTOMER%ADDR%ZIP_CODE = '78910-1203'
CUSTOMER%AGE = 23
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.79 SUBROUTINE Statement

```
SUBROUTINE sub [( [d [, d] ...] )]
```

***where:***

***sub*** is a symbolic name of a subroutine subprogram.

***d*** is a variable name, array name, dummy procedure name or an asterisk (*). `d` is called a *dummy argument.*

A **`SUBROUTINE`** statement is used to define the start of a subroutine subprogram.

*Example:*

```
CALL TMAX3( -1.0, 12.0, 5.0 )
END

SUBROUTINE TMAX3( ARGA, ARGB, ARGC )
    THEMAX = ARGA
    IF( ARGB .GT. THEMAX ) THEMAX = ARGB
    IF( ARGC .GT. THEMAX ) THEMAX = ARGC
    PRINT *, THEMAX
END
```

In the above example, the subroutine `TMAX3` is defined to find and print out the maximum value of three real variables.

*Notes:*

1. No dummy arguments need be specified in the **`SUBROUTINE`** statement. If such is the case, the parentheses () are optional.

For more information, see the chapter entitled "Functions and Subroutines" on page 243.

# 2.80 UNION Statement

```
UNION
```

The **UNION** statement is used in conjunction with the **END UNION** declarative statement. The **UNION** statement marks the start of a series of **MAP** structures. A **UNION** block must contain at least two **MAP** structures. A **UNION** block permits the mapping of the same storage in several different ways.

The following example maps out a 4-byte integer on an Intel 80x86-based processor.

*Example:*

```
STRUCTURE /MAPINT/
    UNION
        MAP
            INTEGER*4 LONG
        END MAP
        MAP
            INTEGER*2 LO_WORD
            INTEGER*2 HI_WORD
        END MAP
        MAP
            INTEGER*1 BYTE_0
            INTEGER*1 BYTE_1
            INTEGER*1 BYTE_2
            INTEGER*1 BYTE_3
        END MAP
    END UNION
END STRUCTURE

RECORD /MAPINT/ I

I%LONG = '01020304'x
PRINT '(2Z4)', I%LO_WORD, I%HI_WORD
END
```

For more information, see the chapter entitled "Structures, Unions and Records" on page 167.

# 2.81 UNTIL Statement

```
UNTIL (e)
```

***where:***

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

The **UNTIL** statement is used in conjunction with the structured **LOOP** or block **WHILE** statement. The **LOOP** or block **WHILE** statement marks the beginning of a sequence of statements which are to be repeated. The **UNTIL** statement marks the end of the loop. The LOOP-block or WHILE-block is executed until control is transferred out of the block or the logical expression of the **UNTIL** statement has a true value.

*Example:*

```
X = 1.0
LOOP
    PRINT *, X, SQRT( X )
    X = X + 1.0
UNTIL( X .GT. 10.0 )
```

*Example:*

```
I = 1
WHILE( I .LT. 100 )DO
    J = 4 * I * I
    K = 3 * I
    PRINT *, '4x**2 + 3x + 6 = ', J + K + 6
    I = I + 1
UNTIL( ( J + K + 6 ) .GT. 100 )
```

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.82 VOLATILE Statement

```
VOLATILE [a [,a] ...]
```

***where:***

***a*** is a variable name or an array name.

The **VOLATILE** statement is used to indicate that a variable or an element of an array may be updated concurrently by other code. A volatile variable or array element will not be cached (in a register) by the code generator. Each time a volatile variable or array element is referenced, it is loaded from memory. Each time a volatile variable or array element is updated, it is stored back into memory.

*Notes:*

1. A name cannot appear in a **VOLATILE** statement more than once in the same program unit.

2. Dummy arguments, procedure names, and common block names are not permitted in a **VOLATILE** statement.

In the following example, the subroutine `A_THREAD` waits on the `HoldThreads` semaphore. It uses the **VOLATILE** statement to ensure that the variable is re-loaded from memory each time through the loop.

*Example:*

```
SUBROUTINE A_THREAD()

STRUCTURE /RTL_CRITICAL_SECTION/
    INTEGER*4 DebugInfo
    INTEGER*4 LockCount
    INTEGER*4 RecursionCount
    INTEGER*4 OwningThread
    INTEGER*4 LockSemaphore
    INTEGER*4 Reserved
END STRUCTURE

INTEGER NumThreads
LOGICAL HoldThreads
VOLATILE HoldThreads
RECORD /RTL_CRITICAL_SECTION/ CriticalSection
COMMON NumThreads, HoldThreads, CriticalSection
INTEGER threadid


WHILE( HoldThreads )DO
    CALL Sleep( 1 )
END WHILE
PRINT '(''Hi from thread '', i4)', threadid()
CALL EnterCriticalSection( CriticalSection )
NumThreads = NumThreads - 1
CALL LeaveCriticalSection( CriticalSection )
CALL endthread()
END
```

# 2.83 Block WHILE Statement

```
WHILE (e) DO    [: block-label]
```

***where:***

***e*** is a logical expression or integer arithmetic expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

The block **WHILE** statement is used in conjunction with the structured **END WHILE** or **UNTIL** statement. The block **WHILE** statement marks the beginning of a sequence of statements which are to be repeated. The **END WHILE** or **UNTIL** statement marks the end of the WHILE-block. The WHILE-block is executed while the logical expression of the **WHILE** statement has a true value or until control is transferred out of the WHILE-block.

*Example:*

```
X = 1.0
WHILE( X .LT. 100 )DO
    PRINT *, X, SQRT( X )
    X = X + 1.0
END WHILE
```

*Example:*

```
I = 1
WHILE( I .LT. 100 )DO
    J = 4 * I * I
    K = 3 * I
    PRINT *, '4x**2 + 3x + 6 = ', J + K + 6
    I = I + 1
UNTIL( ( J + K + 6 ) .GT. 100 )
END
```

An optional block label may be specified with the **WHILE** statement.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193.

# 2.84 WHILE Statement

```
WHILE (e) stmt
```

***where:***

***e*** is a logical expression.

***stmt*** is an executable statement. Only certain executable statements are allowed. See the section entitled "Classifying Statements" on page 9 at the beginning of this chapter for a list of allowed statements.

This form of the **WHILE** statement allows an executable statement to be repeatedly executed until the logical expression *e* is false.

*Example:*

```
I = 0
WHILE( I .LE. 100 ) CALL PRTSQR( I )
END

SUBROUTINE PRTSQR( J )
PRINT *, J, J**2
J = J + 1
END
```

In the above example, the subroutine `PRTSQR` is called again and again until the value of `I` has been incremented beyond 100. Note that the subroutine increments its argument thereby guaranteeing that the program will eventually stop execution.

For more information, see the chapter entitled "Program Structure Control Statements" on page 193 Control Statements".

# 2.85 WRITE Statement

```
WRITE (cilist) [olist]
```

***where:***

***cilist*** is a control information list of specifiers separated by commas:

```
[UNIT =] u
[FMT =] f
REC = rn
IOSTAT = ios
ERR = s
```

***olist*** is an output list.

The **WRITE** statement is used to transfer data from the executing FORTRAN program to an external device or file.

***Control Information List***

***[UNIT =] u***

`u` is an *external unit identifier* or an *internal file identifier.*

1. An external unit identifier is a non-negative integer expression or an asterisk `(*)` in which case unit 6 is assumed.
2. An internal file identifier is the name of a character variable, character array, character array element, or character substring.

If the optional **UNIT=** specifier is omitted then the unit specifier must be the first item in the list of specifiers.

***[FMT =] f***

`f` is a *format identifier.* A format identifier is one of the following:

1. A statement label of a **FORMAT** statement that appears in the same program unit as the format identifier.
2. An integer variable name that has been assigned the statement label of a **FORMAT** statement that appears in the same program unit as the format identifier (see the **ASSIGN** statement).
3. An integer array name.
4. A character array name.
5. Any character expression except one involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant (see the **PARAMETER** statement).
6. An asterisk `(*),` indicating *list-directed* formatting.
7. A **NAMELIST** name, indicating *namelist-directed* formatting.

If the optional **FMT=** specifier is omitted then the format specifier must be the second item in the list of specifiers and **UNIT=** must not be specified for the first item in the list.

***REC = rn***

`rn` is an integer expression whose value must be positive. It is the number of the record to be written when a file is connected for direct access.

***IOSTAT = ios***

is an input/output status specifier. The integer variable or integer array element `ios` is defined with zero if no error condition occurs or a positive integer value if an error condition occurs.

***ERR = s***

is an error specifier and `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s`.

*Output list* - An output list may contain one or more of the following:

1. A variable name.
2. An array element name.
3. A character substring name.
4. An array name except an assumed-size dummy array.
5. Any other expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant (since the length can be determined at compile time).
6. An implied-DO list of the form:

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

   where `dlist` is composed of one or more of items (1) through (6).

*Example:*

```
          WRITE( 6, 100 )X, Y, Z
          WRITE( UNIT=6, FMT=100 )X, Y, Z
100       FORMAT( 3F10.5 )
          WRITE( UNIT=6, FMT='( 3F10.5 )' )X, Y, Z
          WRITE( 6, '( 3F10.5 )' )X, Y, Z
```

The above gives four examples of formatted **WRITE** statements. In all cases, the format conversion is identical but it was specified in different ways. When executed, the effect of all **WRITE** statements is the same. The unit number, used here, is 6. There are, in fact, many other ways in which the **WRITE** statement could have been written, all of which would have the same effect when executed. We have not shown use of all the specifiers.

*Example:*

```
          WRITE( 6, * )X, Y, Z
          WRITE( *, * )X, Y, Z
          WRITE( UNIT=6, FMT=* )X, Y, Z
          WRITE( UNIT=*, FMT=* )X, Y, Z
```

The above four examples of list-directed formatted output are all equivalent. Open Watcom FORTRAN 77 assumes unit 6 when the unit number identifier is an asterisk (as in the second and fourth examples). In the examples, the format identifier is an asterisk indicating list-directed formatting.

*Example:*

```
WRITE( 8 )X, Y, Z
WRITE( UNIT=8 )X, Y, Z
```

The above gives two examples of unformatted **WRITE** statements. The unit number used in the example is 8. When executed, the effect of both of these statements is the same. The values of the variables `X, Y` and `Z` are written to the file connected to unit 8 in their binary form (a form quite incomprehensible to most human beings). An advantage to using this particular form of the **WRITE** statement is that no conversion is required between the internal binary representation of the values and their textual (human-readable) form (which means it takes less computer time to process the data).

*Notes:*

1. If no output list is specified then the effect of the **WRITE** statement is to produce a record whose characters are all blanks.

2. The **REC=** specifier may not be used when list-directed output is specified.

3. An implication of point (6) above is that nesting of implied-DO lists is permitted. For example, the output list

   ```
   ( (A(I,J), B(I,J), J = 1, 5), I = 1, 10 )
   ```

   may be broken down into the following components:

   ```
      A(I,J), B(I,J)
     (....dlist1....., J = 1, 5)
   ( .....dlist2..............., I = 1, 10 )
   ```

For more information on input/output, see the chapter entitled "Input/Output" on page 215. For more information on formatted input/output, see the chapter entitled "Format" on page 225.

# 3 Names, Data Types and Constants

## 3.1 Symbolic Names

*Symbolic names* are names that represent variables, arrays, functions, etc. Names are formed using any of the upper-case letters A-Z and the digits 0-9, the first of which must be a letter. Symbolic names are limited to 6 characters in length. The following are examples of symbolic names.

```
AMOUNT
AGE
CUST73
```

Open Watcom FORTRAN 77 extends the allowable characters that can make up a symbolic name to include the lower-case letters a-z, the dollar sign ($) and the underscore (_). Note that the dollar sign and the underscore are treated as letters and are therefore allowed as the first letter of a symbolic name. Furthermore, Open Watcom FORTRAN 77 allows symbolic names of up to 32 characters. The following are examples of permissible symbolic names.

```
Evaluate
$Cheque
ComputeAverage
_device
IO$ERROR
student_total
```

Open Watcom FORTRAN 77 makes no distinction between upper and lower case letters. The following symbolic names are identical.

```
Account
ACCount
ACCOUNT
```

Spaces are allowed in symbolic names and are ignored. The following symbolic names are identical.

```
C R E DIT
CRE D I T
```

FORTRAN 77 allows certain keywords such as WRITE to be used as symbolic names. In Open Watcom FORTRAN 77, all *keywords* satisfy the requirements of a symbolic name. A keyword is a sequence of letters that is interpreted in a special way by Open Watcom FORTRAN 77. Whether a string of characters is interpreted as a keyword or as a symbolic name depends on the context in which it is used. In the following example, the first statement is an assignment statement assigning the value 2 to the symbolic name `DO10I`. The second statement is the beginning of a DO-loop.

*Example:*

```
DO10I=1
DO10I=1,10
```

# 3.2 Data Types

There are 6 basic data types in FORTRAN 77; logical, integer, real, double precision, complex and character. Open Watcom FORTRAN 77 provides an additional data type, namely double precision complex ( **DOUBLE COMPLEX** or **COMPLEX*16**). Open Watcom FORTRAN 77 also supports the creation of more complex user-defined data types using the **STRUCTURE** statement.

Each data type can be classified as numeric, logical or character. Each datum occupies a sequence of storage units. Numeric data and logical data occupy numeric storage units whereas character data occupy character storage units. In Open Watcom FORTRAN 77, a numeric storage unit occupies 4 bytes and a character storage unit occupies 1 byte.

The following table summarizes all data types supported by Open Watcom FORTRAN 77.

| Data Type | Size (in bytes) | Standard FORTRAN |
|---|---|---|
| LOGICAL | 4 | yes |
| LOGICAL*1 | 1 | extension |
| LOGICAL*4 | 4 | extension |
| INTEGER | 4 | yes |
| INTEGER*1 | 1 | extension |
| INTEGER*2 | 2 | extension |
| INTEGER*4 | 4 | extension |
| REAL | 4 | yes |
| REAL*4 | 4 | extension |
| REAL*8 | 8 | extension |
| DOUBLE PRECISION | 8 | yes |
| COMPLEX | 8 | yes |
| COMPLEX*8 | 8 | extension |
| DOUBLE COMPLEX | 16 | extension |
| COMPLEX*16 | 16 | extension |
| CHARACTER | 1 | yes |
| CHARACTER*n | n | yes |

Detailed information on the size and range of values supported by each of these data types is provided in the User's Guide.

# 3.3 Data Type of a Name

A name must only have one data type. Its type is specified by the appearance of that name in a type statement. If a name does not appear in any type statement then an implied type is assigned to it by the "first letter rule". A name not appearing in any type statement and beginning with any of the letters I, J, K, L, M or N is assigned the type integer. A name not appearing in any type statement and beginning with any other letter is assigned the type real. The implied type of a letter can be changed by an **IMPLICIT** statement.

The type associated with a name defines the type of the data it is to contain. For example, if A is of type integer, then the storage unit which A occupies is assumed to contain integer data. Note that the data type of an array element is the same as the data type associated with the array name.

The data type of a function name specifies the type of the result returned by the function when it is referenced. A name that identifies a specific intrinsic function has type as specified in the chapter entitled "Functions and Subroutines" on page 243. A *generic function* name has no type associated with it; its type is determined by the type of its argument(s). The appearance of a generic function in a type statement is not sufficient to remove the generic properties of that name. For example, if SIN was declared to be of type real, it could still be called with an argument of type complex. The type of an external function reference is determined in the same way as for variables and arrays. The actual type of the external function is determined implicitly by its name or explicitly by its appearance in a **`FUNCTION`** or type statement. Note that an **`IMPLICIT`** statement can affect the type of the external function being defined.

# 3.4 Constants

A *constant* can be one of arithmetic, logical or character. Each constant has a data type and value associated with it and, once established in a program, cannot be changed. Arithmetic constants consist of those constants whose data type is one of integer, real, double precision, complex or double precision complex. Logical constants consist of those constants whose data type is logical and character constants consist of those constants whose data type is character. The string of characters representing a constant determines its value and data type. The blank character is insignificant for all but character constants.

## 3.4.1 Integer Constants

An *integer constant* is formed by a non-empty string of digits preceded by an optional sign.

The following are examples of integer constants.

```
1423
+345
-34565788
```

## 3.4.2 Real Constants

We first define a *simple real constant* as follows: an optional sign followed by an integer part followed by a decimal point followed by a fractional part. The integer and fractional parts are non-empty strings of digits. Either can be omitted but not both.

A *real constant* has one of the following forms.

1. A simple real constant.
2. A simple real constant followed by an `E` followed by an optionally signed integer constant.
3. An integer constant followed by an `E` followed by an optionally signed integer constant.

The optionally signed integer constant that follows the `E` is called the *exponent.* The value of a real constant that contains an exponent is the value of the constant preceding the `E` multiplied by the power of ten determined by the exponent.

The following are examples of real constants.

```
123.764
.4352344
1423.34E12
+345.E-4
-.4565788E3
2E6
1234.
```

## 3.4.3 Double Precision Constant

A *double precision constant* has one of the following forms.

1. A simple real constant followed by a `D` followed by an optionally signed integer constant.
2. An integer constant followed by a `D` followed by an optionally signed integer constant.

The optionally signed integer constant that follows the `D` is called the *exponent.* The value of a double precision constant that contains an exponent is the value of the constant preceding the `D` multiplied by the power of ten determined by the double precision exponent. Note that the resulting approximation is of greater precision than the equivalent real constant. The approximations may be of equal precision if the approximations are exact representations. For example, `0D0` and `0E0` are double and single precision constants respectively, both representing zero with the same precision.

The following are examples of double precision constants.

```
1423.34D12
+345.D-4
-.4565788D5
2D6
```

## 3.4.4 Complex Constant

A *complex constant* consists of a left parenthesis, followed by a real or integer constant representing the real part of the complex constant, followed by a comma, followed by a real or integer constant representing the imaginary part of the complex constant, followed by a right parenthesis.

The following are examples of complex constants.

```
( 1423.34E12, 3 )
( +345, 4 )
```

## 3.4.5 Double Precision Complex Constant (Extension)

A *double precision complex constant* has the same form as a complex constant except that at least one of the real and imaginary parts *must* be a double precision constant.

The following are examples of double precision complex constants.

```
( 1423.34D12, 3 )
( +345, 4D2 )
```

## 3.4.6 Logical Constant

A *logical constant* can have one of the following forms.

1. `.TRUE.` representing the value true.
2. `.FALSE.` representing the value false.

## 3.4.7 Character Constant

A *character constant* consists of an apostrophe followed by any string of characters followed by an apostrophe. The apostrophes are not part of the datum. If an apostrophe is to appear as part of the datum it must be followed immediately by another apostrophe. Note that blanks are significant. The length of the character constant is the number of characters appearing between the delimiting apostrophes. Consecutive apostrophes in a character datum represent one character, namely the apostrophe. A character constant must not have length 0.

The following are examples of character constants.

```
'ABCDEFG1234567'
'There''s always tomorrow'
```

## 3.4.8 String Constant (Extension)

A *string constant* consists of an apostrophe followed by any string of characters followed by an apostrophe and then the letter `C` or `c`. The apostrophes are not part of the datum. The datum is stored in memory with a terminating NUL character (CHAR(0)). If an apostrophe is to appear as part of the datum it must be followed immediately by another apostrophe. Note that blanks are significant. The length of the string constant is the number of characters appearing between the delimiting apostrophes plus one for the terminating NUL character (CHAR(0)). Consecutive apostrophes in a string datum represent one character, namely the apostrophe. A string constant must not have length 0. A string constant may be used anywhere a character constant may be used.

The following are examples of string constants.

```
'Hello there'C
'There''s always tomorrow'c
'The result for %s=%d'c
```

## 3.4.9 Hollerith Constants (Extension)

A *hollerith constant* consists of a positive unsigned integer constant `n` followed by the letter `H` or `h` followed by a string of exactly `n` characters. The actual data is the `n` characters following the letter `H` or `h`. A hollerith constant is another way of representing character data.

Actually, hollerith constants are treated as character constants and can be used wherever a character constant can be used. Hollerith constants are different from character constants in that a quote is represented by two quotes in character constants and by a single quote in hollerith constants.

The following are examples of hollerith constants.

```
5HABCDEFG
10h xxxxx '44
```

## 3.4.10 Hexadecimal Constants (Extension)

Two forms of *hexadecimal constant* are supported. The first form can only be used in type declaration or **DATA** statements. The second form may be used anywhere an integer constant may be used.

The first form of hexadecimal constant consists of the letter `Z` or `z` followed by a string of hexadecimal digits. A hexadecimal digit can be any digit or one of the letters `A, B, C, D, E` or `F` (the lower case of these letters is also acceptable). The actual data is the hexadecimal digits following the letter `Z` or `z.` Hexadecimal constants of this form can only be used in type declaration statements and **DATA** statements for initializing memory with binary patterns.

The following are examples of the first form of hexadecimal constant.

```
z1234
Zac
```

The first example is equivalent to the binary pattern `0001 0010 0011 0100.` The second example is equivalent to the binary pattern `1010 1100.`

The second form of hexadecimal constant consists of an apostrophe followed by any string of hexadecimal digits followed by an apostrophe and then the letter `X` or `x.` A hexadecimal digit can be any digit or one of the letters `A, B, C, D, E` or `F` (the lower case of these letters is also acceptable). The actual data is the hexadecimal digits placed inside apostrophes.

The following are examples of the second form of hexadecimal constant.

```
'1234'x
'ac'X
```

The first example is equivalent to the binary pattern `0001 0010 0011 0100.` The second example is equivalent to the binary pattern `1010 1100.`

## 3.4.11 Octal Constants (Extension)

An *octal constant* consists of an apostrophe followed by any string of octal digits followed by an apostrophe and then the letter `O` or `o.` An octal digit can be any of the digits 0 through 7. The actual data is the octal digits placed inside apostrophes. An octal constant may be used anywhere an integer constant may be used.

The following are examples of octal constants.

```
'1234'o
'37'O
```

The first example is equivalent to the binary pattern `001 010 011 100`. The second example is equivalent to the binary pattern `011 111`.

## 3.5 Symbolic Constants

It is possible to give a constant a symbolic name. This is done through **PARAMETER** statements. For more details, see the section on the **PARAMETER** statement in the chapter entitled "FORTRAN Statements" on page 9.

# 4 Arrays

## 4.1 Introduction

An array is a non-empty collection of data. Arrays allow a convenient way of manipulating large quantities of data. An array can be referenced as an entity. In this way it is possible to conveniently pass large quantities of data between subprograms. Alternatively, it is possible to reference each element of an array individually so that data can be selectively processed. Consider the task of managing the marks of 100 students. Without arrays one would have to have a unique name for each mark. They might be M1, M2, etc. up to M100. This is clearly cumbersome. Instead, we can use an array called MARKS containing 100 elements. Now there is one name for all the marks. Each mark can be referenced by using that name followed by a subscript. Furthermore, suppose the size of the class doubled. Do we add the names M101, M102, etc. up to M200? Not if we use arrays. If the size of the class doubled, all that need be done is to define the array to contain 200 elements. It is not hard to see that programs that use arrays tend to be general in nature. Arrays also facilitate the repetitive computations that must be performed on large amounts of data in that they lend themselves to loop processing.

## 4.2 Properties of Arrays

Arrays are defined by an array declarator. The form of an *array declarator* is:

```
a( d [,d] ... )
```

***where:***

***a*** is the symbolic name of the array

***d*** is a dimension declarator.

The number of dimensions of the array is determined by the number of dimension declarators appearing in the array declarator. Allowable dimensions for arrays range from 1 to 7. A 1-dimensional array can be viewed as a vector, a 2-dimensional array as a matrix and a 3-dimensional array as a number of parallel matrices. Arrays with dimension higher than 3 are generally difficult to intuitively describe and hence examples will deal with arrays whose dimension is 1, 2 or 3.

Each dimension has a range of values. When referencing elements in that dimension, the dimension expression must fall in that range. The range of a dimension is defined in the dimension declarator. A *dimension declarator* has the following form:

```
[lo:] hi
```

***where:***

***lo*** is the lower dimension bound.

***hi*** is the upper dimension bound.

The lower and upper dimension bounds must be integer expressions and the upper dimension bound must be greater than or equal to the lower dimension bound. The upper dimension bound of the last dimension may be an asterisk `(*)`. The meaning of this will be discussed later. If the lower dimension bound is not specified then a default of 1 is assumed. The size of a dimension is defined as `hi` − lo + 1. Note that if the lower dimension bound is not specified the size of the dimension is just `hi`. The size of the array (or the number of elements in the array) is defined as the product of all the sizes of the dimensions of the array. The maximum number of elements in any dimension is limited to 65535. The maximum size of an array is limited by the amount of available memory.

Arrays are defined by the appearance of an array declarator in a **`DIMENSION`** statement, a type statement or a **`COMMON`** statement.

*Example:*

```
DIMENSION A(10), B(-5:5,-10:10)
INTEGER C(10,20)
COMMON /DATA/ X,Y(30,30),Z
```

In the previous example, B is a 2-dimensional array with 11 rows and 21 columns and has 231 elements (i.e. 11 * 21).

Each array has a data type associated with it. This data type is inherited by all elements of the array.

# 4.3 Array Elements

Each array is comprised of a sequence of array elements. An array element is referenced by following the array name with a *subscript.* Different elements of the array are referenced by simply changing the subscript. An *array element* has the following form:

```
a(s[,s]...)
```

***where:***

***a*** is the array name.

***(s[,s]...)*** is a subscript.

***s*** is a subscript expression.

Each *subscript expression* must be an integer expression and must be in the range defined by the upper and lower dimension bounds of the corresponding dimension. The number of subscript expressions must be equal to the dimension of the array.

If an array has *n* elements then there is a 1-to-1 correspondence between the elements of the array and the integers from 1 to *n.* Each subscript has a *subscript value* associated with it which determines which element of the array is being referenced. If the subscript value is *i* then the *i*th element of the array is the one referenced. The subscript value depends on the subscript expressions and on the dimensions of the array. The following table describes how to compute the subscript value.

| n | Dimension Declarator | Subscript Value | Subscript |
|---|---|---|---|
| 1 | (J1:K1) | (S1) | 1+(S1-J1) |
| 2 | (J1:K1,J2:K2) | (S1,S2) | 1+(S1-J1)<br>+(S2-J2)*D1 |
| 3 | (J1:K1,J2:K2,J3:K3) | (S1,S2,S3) | 1+(S1-J1)<br>+(S2-J2)*D1<br>+(S3-J3)*D2*D1 |
| .<br>.<br>. | .<br>.<br>. | .<br>.<br>. | .<br>.<br>. |
| n | (J1:K1,...,Jn:Kn) | (S1,...,Sn) | 1+(S1-J1)<br>+(S2-J2)*D1<br>+(S3-J3)*D2*D1<br>+<br>+(Sn-Jn)*Dn-1*Dn-2*...*D1 |

*Notes:*

1. n is the number of dimensions, 1 <= n <= 7.
2. Ji is the value of the lower bound of the *i'th* dimension.
3. Ki is the value of the upper bound of the *i'th* dimension.
4. If only the upper bound is specified, then Ji = 1
5. Si is the integer value of the *i'th* subscript expression.
6. Di = Ki-Ji+1 is the size of the *i'th* dimension. If the value of the lower bound is 1, then Di = Ki.
7. A subscript of the form (J1,...,Jn) has subscript value 1 and identifies the first element of the array. A subscript of the form (K1,...,Kn) has subscript value equal to the size of the array and identifies the last element of the array.

# 4.4 Classifying Array Declarators by Dimension Declarator

Array declarators can be classified according to the characteristics of the dimension declarator. The following sections discuss the three classifications.

## 4.4.1 Constant Array Declarator

A *constant array declarator* is one in which each of the dimension bound expressions is an integer constant expression. It is called a constant array declarator because the dimension bound expressions can never change. In the following example both `A(10)` and `B(−5:5)` are constant array declarators.

*Example:*

```
SUBROUTINE SQUARE( A )
DIMENSION A(10), B(−5:5)
.
.
.
END
```

## 4.4.2 Adjustable Array Declarator

An *adjustable array declarator* is one that contains at least one variable in all of its dimension bound expressions. It is called an adjustable array declarator because the dimension bound expressions can change depending on the current value of the variables in the dimension bound expressions. The array name must be a dummy argument. In the following example, `A(M,2*N)` is an adjustable array declarator. If `SQUARE` is called with `M` having value 5 and `N` having value 10, then the array `A` will be a 2-dimensional array having 5 rows and 20 columns.

*Example:*

```
SUBROUTINE SQUARE( A, M, N )
DIMENSION A(M,2*N)
.
.
.
END
```

## 4.4.3 Assumed-size Array Declarator

An *assumed-size array declarator* is a constant array declarator or an adjustable array declarator whose upper dimension bound of the last dimension is an asterisk (e.g., A(M,N,*)) or the integer value 1 (e.g., A(M,N,1)). The array name must be a dummy argument. The value of the upper bound of the last dimension is determined by the number of elements of the actual array argument and is computed as follows. First we compute the size of the dummy array. Note that this size is really an upper bound.

1. If the corresponding actual array argument is a non-character array name, the size of the dummy array is the size of the actual array.

2. If the corresponding actual array argument is a non-character array element name with a subscript value of $r$ in an array of size $x$, the size of the dummy array is $x + 1 - r$.

3. If the corresponding actual argument is a character array name, character array element or a substring character array element which begins at character *t* of an array with *c* characters then the size of the dummy array is INT(( *c* + 1 − *t* ) / *e)* where *e* is the size of an element of the dummy array.

If the assumed-size array has dimension *n* then the product of the first *n* − 1 dimensions must be less than or equal to the size of the array as determined by one of the preceding rules. The value of the assumed dimension is the largest integer such that the product of all of the dimensions is less than or equal to the size of the dummy array. In the following example, `A(4,*)` is an assumed-size array declarator.

*Example:*

```
DIMENSION B(10)
.
.
.
CALL SQUARE( B )
.
.
.
END

SUBROUTINE SQUARE( A )
DIMENSION A(4,*)
.
.
.
END
```

By rule 1, the upper bound of the size of `A` is 10. We now look for the largest integer *n* such that 4 * *n* is less than or equal to 10. Clearly, *n* is 2. `A` is therefore a 2-dimensional array with 4 rows and 2 columns.

## 4.4.4 Allocatable Array Declarator

An *allocatable array declarator* is one that contains no dimension bound expressions. It is called an allocatable array declarator because the dimension bounds are specified at run-time in an **`ALLOCATE`** statement.

*Example:*

```
DIMENSION A(:), B(:,:)
     .
     .
     .
ALLOCATE( A(N) )
ALLOCATE( B(0:4,5) )
```

In the previous example, `A(:)` is a one-dimensional allocatable array declarator and `B(:,:)` is a two-dimensional allocatable array declarator. The first **`ALLOCATE`** statement is used to allocate the array `A` with bounds `1:N`. The second **`ALLOCATE`** statement is used to allocate the array `B` with bounds `0:4` in the first dimension and `1:5` in the second dimension.

# 4.5 Classifying Array Declarators by Array Name

Array declarators can also be classified according to the characteristic of the array name. The following sections discuss the two classifications.

## 4.5.1 Actual Array Declarator

An *actual array declarator* is one in which the array name is not a dummy argument. All actual array declarators must also be constant array declarators. An actual array declarator is permitted in a **DIMENSION** statement, a type statement or a **COMMON** statement.

## 4.5.2 Dummy Array Declarator

A *dummy array declarator* is one in which the array name is a dummy argument and hence can only appear in a function or subroutine subprogram. It can be a constant, adjustable or assumed-size array declarator. A dummy array declarator can appear in a **DIMENSION** statement or a type statement but not in a **COMMON** statement. It should be noted that the array declarator for a dummy array declarator need not be the same as the array declarator of the corresponding actual array declarator. Also note that every array declarator in a main program must be a constant array declarator.

# 4.6 Use of Array Names

The appearance of an array name must always be as part of an array element name except in the following cases:

1. in a list of dummy arguments. For example, a subroutine that has as one of its arguments an array.
2. in a **COMMON** statement to define that array as belonging to a common block.
3. in a type statement either as part of an array declarator or by itself to establish the type of the array.
4. in an array declarator in a **DIMENSION**, type or **COMMON** statement.
5. in an **EQUIVALENCE** statement.
6. in a **DATA** statement.
7. in the list of actual arguments when calling an external procedure.
8. In the list of an input/output statement.
9. as a unit identifier for an internal file in an input/output statement.
10. as a format identifier in an input/output statement.
11. in a **SAVE** statement.

# 5 Character Substrings

## 5.1 Introduction

A *substring* is a contiguous portion of a character entity. The substring operation selects a substring from a character entity. The resulting substring can then be treated as a character entity in itself. Substringing also allows the replacement of substrings from character entities with other character entities.

## 5.2 Substring Names

Substrings are formed by specifying a substring name. The forms of a *substring name* are:

```
v( [ e1] : [ e2] )
a(s [,s] ...)( [ e1] : [ e2] )
```

***where:***

| | |
|---|---|
| ***v*** | is a character variable name. |
| ***a(s[,s]...)*** | is a character array element name. |
| ***e1*** | is an integer expression identifying the leftmost character of the substring. |
| ***e2*** | is an integer expression identifying the rightmost character of the substring. |

`e1` and `e2` are called *substring expressions.* They must be such that

```
1 <= e1 <= e2 <= len
```

where `len` is the length of the character entity. If `e1` is omitted, a value of 1 is assumed. If `e2` is omitted, a value of `len` is assumed. Both `e1` and `e2` may be omitted. The length of the substring is `e2 - e1 + 1`.

*Example:*

```
      CHARACTER A*8, B(4)*8, C*14
* A gets the string 'EVERYDAY'
      A = 'EVERYDAY'
* Replace 'DAY' with 'ONE' in A
      A(6:8) = 'ONE'
* B(1) gets the string 'OTHELLO'
      B(1) = 'OTHELLO'
* B(2) gets same value as B(1)
      B(2)(:) = 'OTHELLO'
* B(3) gets last 6 characters of B(1)
      B(3) = B(1)(3:8)
* B(4) gets first 4 characters of B(1)
*     concatenated with the letter 'R'
      B(4) = B(1)(1:4) // 'R'
* C gets last 6 characters of B(1)
*     concatenated with the variable A
      C = B(1)(3:) // A
* Print out the results
      PRINT *, A
      PRINT '(A8)', B
      PRINT *, C
      END
```

# 5.3 Extensions

Open Watcom FORTRAN 77 allows an external character function reference or a character statement function reference as part of the substring name (see the chapter entitled "Functions and Subroutines" on page 243. for more information).

*Example:*

```
      CHARACTER*10 F,G
      CHARACTER*10 X
*
* DEFINE CHARACTER STATEMENT FUNCTION
*
      G(X) = X
*
      PRINT *, F('0123456789')(1:5)
      PRINT *, G('0123456789')(6:10)
      END

*
* DEFINE CHARACTER EXTERNAL FUNCTION
*
      CHARACTER*(*) FUNCTION F( X )
      CHARACTER*10 X
      F = X
      END
```

# 6 Structures, Unions and Records

## 6.1 Structures and Records

As an extension to the basic FORTRAN 77 types such as INTEGER, REAL, LOGICAL, etc., Open Watcom FORTRAN 77 supports the creation of hierarchical, composite data types called *structures.* A structure is a template describing the form of a *record.* It is composed of members or fields of various types, including other structures. A structure does not reserve any storage.

For example, you could describe the structure of the COMPLEX data type using the following construction.

*Example:*

```
STRUCTURE /CMPLX/
    REAL REAL_PART
    REAL IMAG_PART
END STRUCTURE
```

Since the COMPLEX data type is an intrinsic type of FORTRAN, there is no need to do so. The **STRUCTURE** and **END STRUCTURE** statements mark the start and end of a structure definition.

There are, however, many practical examples of collections of data that may be described using a structure. Consider, for example, the contents of a data record on disk. It may contain fields such as last name, first name, and middle initial which describe the name of a customer. Each of these fields are fixed in length. A sample structure declaration might be:

```
STRUCTURE /NAME/
    CHARACTER*20 LAST_NAME
    CHARACTER*20 FIRST_NAME
    CHARACTER*1  MIDDLE_INITIAL
END STRUCTURE
```

As we stated above, a structure does not allocate storage. Instead, we have created a new type called `NAME` which may be used to describe objects. Objects of the new type are defined using the **RECORD** statement. For example, the following statements describe two objects, `STUDENT_1` and `STUDENT_2`, to be of type `NAME`.

```
RECORD /NAME/ STUDENT_1
RECORD /NAME/ STUDENT_2
```

There are other attributes of a person besides one's name that could be recorded in the record. For example, we can also store a person's date of birth and sex. First, let us define a `DATE` structure.

```
STRUCTURE /DATE/
    INTEGER*1 DAY
    INTEGER*1 MONTH
    INTEGER*2 YEAR
END STRUCTURE
```

Now we can describe a person in terms of name, date of birth, and sex.

```
STRUCTURE /PERSON/
    RECORD /NAME/ NAME
    RECORD /DATE/ BIRTH_DATE
    CHARACTER*1 SEX
END STRUCTURE

RECORD /PERSON/ STUDENT
```

Having declared `STUDENT` to be of type `PERSON`, how do we reference the component parts of `STUDENT`? The following example illustrates this.

```
STUDENT.NAME.LAST_NAME = 'Pugsley'
STUDENT.NAME.FIRST_NAME = 'Elmar'
STUDENT.NAME.MIDDLE_INITIAL = 'M'
STUDENT.BIRTH_DATE.DAY = 21
STUDENT.BIRTH_DATE.MONTH = 11
STUDENT.BIRTH_DATE.YEAR = 1959
STUDENT.SEX = 'M'
```

The object's name is specified first, followed by a "." (or "%") and the structure member name. If the structure member is itself a record then another "." (or "%") and member name is specified. This continues until the desired structure member is identified. The "." or "%" is called a *field selection operator.*

The previous example contained both a structure called `NAME (RECORD /NAME/)` and a structure member called `NAME (RECORD /NAME/ NAME)`. The structure name is enclosed within slashes ("/"). A structure name must be unique among structure names. However, the same name can also be used to name either variables or structure members (fields). Thus it is possible to have a variable named `X`, a structure named `X`, and one or more fields named `X`.

Structure, field, and variable names are all local to the program unit in which they are defined.

# 6.2 Arrays of Records

It is often the case that the individual attributes of objects are stored in separate arrays. If, for example, your application deals with 1000 objects with attributes "size", "weight", and "colour", the traditional approach is to declare three different arrays.

```
PARAMETER (MAX_ELS=1000)

REAL         SIZE(MAX_ELS)
INTEGER      WEIGHT(MAX_ELS)
CHARACTER*2 COLOUR(MAX_ELS)
```

To read or write the attributes relating to an object, you would use a statement such as:

```
READ(UNIT=3) SIZE(I), WEIGHT(I), COLOUR(I)
```

Using a simple structure, we can express the problem as follows:

```
PARAMETER (MAX_ELS=1000)

STRUCTURE /OBJECT/
    REAL        SIZE
    INTEGER     WEIGHT
    CHARACTER*2 COLOUR
END STRUCTURE

RECORD /OBJECT/ ITEM(MAX_ELS)
```

To read or write the attributes relating to an object, you would use a statement such as:

```
READ(UNIT=3) ITEM(I)
```

## 6.3 Unions

Sometimes it is useful to be able to describe parts of structures in different ways in much the same way that the **EQUIVALENCE** statement is used to describe a specific storage area in different ways. The **UNION** - **END UNION** statements are used to mark a section of a structure that will have alternate storage organizations (MAPs). The **MAP** - **END MAP** statements are used to define the start and end of an alternate storage map. Thus several **MAP** - **END MAP** pairs will appear between a **UNION** - **END UNION** section.

Consider the following example. The subroutine displays the contents of a field using different names and formats depending on a `TYPE` field.

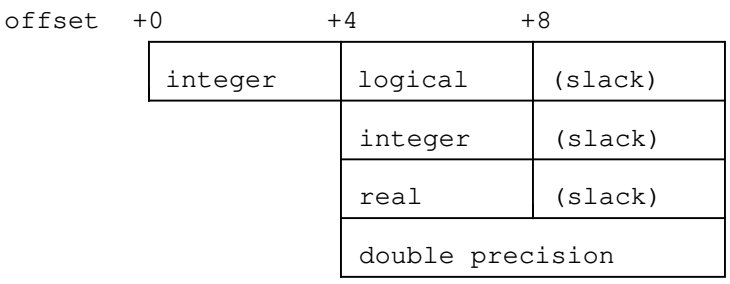
*Example:*

```
SUBROUTINE PRINT_ITEM( ITEM )
    STRUCTURE /DATA_MAP/
        INTEGER TYPE
        UNION
            MAP
                LOGICAL           LGL
            END MAP
            MAP
                INTEGER           INT
            END MAP
            MAP
                REAL              FLT
            END MAP
            MAP
                DOUBLE PRECISION  DBL
            END MAP
        END UNION
    END STRUCTURE

    RECORD /DATA_MAP/ ITEM

    IF( ITEM%TYPE .EQ. 1 ) THEN
        PRINT '(L2)', ITEM%LGL
    ELSEIF( ITEM%TYPE .EQ. 2 ) THEN
        PRINT '(I8)', ITEM%INT
    ELSEIF( ITEM%TYPE .EQ. 3 ) THEN
        PRINT '(E12.5)', ITEM%FLT
    ELSEIF( ITEM%TYPE .EQ. 4 ) THEN
        PRINT '(D12.5)', ITEM%DBL
    ENDIF
END
```

The organization of the record in memory is as follows:




The first 4 bytes of storage are occupied by `TYPE`. The next 4 to 8 bytes of storage are occupied by either `LGL`, `INT`, `FLT`, or `DBL` depending on the interpretation of the contents of the variable `TYPE`. The size of the record `ITEM` is a total of 12 bytes. Based on the conventions of the above program example, only 8 bytes of the record `ITEM` are used when `TYPE` is 1, 2, or 3. When `TYPE` is 4 then 12 bytes of the record are used.

The following example maps out a 4-byte integer on an Intel 80x86-based processor.

*Example:*

```
STRUCTURE /MAPINT/
    UNION
        MAP
            INTEGER*4 LONG
        END MAP
        MAP
            INTEGER*2 LO_WORD
            INTEGER*2 HI_WORD
        END MAP
        MAP
            INTEGER*1 BYTE_0
            INTEGER*1 BYTE_1
            INTEGER*1 BYTE_2
            INTEGER*1 BYTE_3
        END MAP
    END UNION
END STRUCTURE

RECORD /MAPINT/ I

I%LONG = '01020304'x
PRINT '(Z8)', I%LONG
PRINT '(Z4,1X,Z4)', I%LO_WORD, I%HI_WORD
PRINT '(Z2,3(1X,Z2))', I%BYTE_0, I%BYTE_1,
$                      I%BYTE_2, I%BYTE_3
END
```

The above example produces the following output:

```
01020304
0304 0102
04 03 02 01
```

# 7 Expressions

The following topics are discussed in this chapter.

- Arithmetic Expressions
- Character Expressions
- Relational Expressions
- Logical Expressions
- Evaluating Expressions
- Constant Expressions

## 7.1 Arithmetic Expressions

Arithmetic expressions are used to describe computations involving operands with numeric data type, arithmetic operators and left and right parentheses. The result of the computation is of numeric data type.

### 7.1.1 Arithmetic Operators

The following table lists the arithmetic operators and the operation they perform.

| Operator | Arithmetic Operation |
|---|---|
| ** | Exponentiation |
| / | Division |
| * | Multiplication |
| – | Subtraction or Negation |
| + | Addition or Identity |

Some operators can be either binary or unary. A *binary operator* is one that requires two operands. A *unary operator* is one that requires one operand. Each of the operators **, /, and * are binary operators. The operators + and – can either be binary or unary operators. The following table describes how each operator is used with their operands.

| Operator | Arithmetic Operation |
|---|---|
| x ** y | x is raised to the power y |
| x / y | x is divided by y |
| x * y | x is multiplied by y |
| x – y | y is subtracted from x |
| x + y | y is added to x |
| – x | x is negated |
| + x | identity |

Arithmetic expressions can contain more than one operator. It is thus necessary to define rules of evaluation for such expressions. A *precedence relation* is defined between operators. This relation defines the order in which operands are combined and hence describes the evaluation sequence of an arithmetic expression. Operands of higher precedence operators are combined using that operator to form an operand for an operator of lower precedence. The following rules define the precedence relation among arithmetic operators.

1. Exponentiation (**) has highest precedence.
2. Multiplication (*) and division (/) have equal precedence but have lower precedence than exponentiation.
3. Addition (+) and subtraction (–) have equal precedence but have lower precedence than multiplication and division.

For example, to evaluate the expression

```
A–B**4
```

B is raised to the exponent 4 first and the result is then subtracted from A.

Parentheses can be used to alter the evaluation sequence of an arithmetic expression. When a left parenthesis is encountered, the entire expression enclosed in parentheses is evaluated. Consider the following expression.

```
3*(4+5)
```

We first evaluate the expression in the parentheses, the result being 9. We now multiply the result by 3 giving a final result of 27. Now suppose we remove the parentheses. According to the precedence rules, * has precedence over + so we perform the multiplication before the addition. The result in this case is 17.

## 7.1.2 Rules for Forming Standard Arithmetic Expressions

The building blocks for arithmetic expressions are called *arithmetic primaries.* They are one of the following:

1. unsigned arithmetic constant
2. arithmetic symbolic constant
3. arithmetic variable reference
4. arithmetic array element reference
5. arithmetic function reference
6. ( arithmetic expression )

A grammar for forming arithmetic expressions can be described which reflects the precedence relation among arithmetic operators.

Exponentiation has highest precedence. We define a *factor* as:

1. primary
2. primary ** factor

A factor is simply a sequence of primaries, each separated by the exponentiation operator. Rule (2) specifies that the primaries involving exponentiation operators are combined from right to left when evaluating a factor.

Next in the precedence hierarchy are the multiplication and division operators. We define a *term* as:

1. factor
2. term / factor
3. term * factor

A term is simply a sequence of factors, each separated by a multiplication operator or a division operator. Rules (2) and (3) imply that in such a sequence, factors are combined from left to right when evaluating a term. Factors can be interpreted as the result obtained from evaluating them. This implies that all factors are evaluated before any of the multiplication or division operands are combined. This interpretation is consistent with the precedence relation between the exponentiation operator and the division and multiplication operators.

An *arithmetic expression* can now be defined as follows.

1. term
2. + term
3. – term
4. arithmetic expression + term
5. arithmetic expression – term

An arithmetic expression is simply a sequence of terms, each separated by an addition operator or a subtraction operator. Rules (4) and (5) imply that terms are evaluated from left to right. Rules (2) and (3) imply that only the first term of an arithmetic expression can be preceded by a unary + or – operator. Terms can be interpreted in the same way as factors were interpreted in the definition of terms.

Note that consecutive operators are not permitted. For example, the expression

```
A+-B
```

is illegal. However, expressions of the form

```
A+(-B)
```

are allowed.

## 7.1.3 Arithmetic Constant Expression

An *arithmetic constant expression* is an arithmetic expression in which all primaries are one of the following.

1. arithmetic constant
2. symbolic arithmetic constant
3. ( arithmetic constant expression )

There is a further restriction with the exponentiation operator; the exponent must be of type INTEGER.

As an extension to the FORTRAN 77 language, Open Watcom FORTRAN 77 supports the use of the intrinsic function **ISIZEOF** in an arithmetic constant expression.

*Example:*

```
          PARAMETER (INTSIZ = ISIZEOF(INTEGER))
```

An *integer constant expression* is an arithmetic constant expression in which all constants and symbolic constants are of type INTEGER.

*Example:*

```
          123
          -753+2
          -(12*13)
```

A *real constant expression* is an arithmetic constant expression in which at least one constant or symbolic constant is of type REAL and all other constants or symbolic constants are of type REAL or INTEGER.

*Example:*

```
          123.
          -753+2.0
          -(13E0*12)
```

A *double precision constant expression* is an arithmetic constant expression in which at least one constant or symbolic constant is of type DOUBLE PRECISION and all other constants or symbolic constants are of type DOUBLE PRECISION, REAL or INTEGER.

*Example:*

```
          123.4D0
          -753D0*2+.5
          -(12D0*12.2)
```

A *complex constant expression* is an arithmetic constant expression in which at least one constant or symbolic constant is of type COMPLEX and all other constants or symbolic constants are of type COMPLEX, REAL or INTEGER.

*Example:*

```
(123,0)
(-753,12.3)*2
-(12,-12.4)-(1.0,.2)
```

A *double precision complex constant expression* is an arithmetic constant expression in which at least one constant or symbolic constant is of type COMPLEX*16 and all other constants or symbolic constants are of type COMPLEX*16, DOUBLE PRECISION, REAL or INTEGER. If there are no constants or symbolic constants of type COMPLEX*16 in a constant expression, the type of the constant expression will be COMPLEX*16 if it contains at least one constant or symbolic constant of type COMPLEX and at least one constant or symbolic constant of type DOUBLE PRECISION. Open Watcom FORTRAN 77 supports this

*Example:*

```
(123,0D0)
(-753,12.3D0)*2
-(12D0,-12.4)-(1.0,.2)
```

## 7.1.4 Data Type of Arithmetic Expressions

Evaluating an arithmetic expression produces a result which has a type. The type of the result is determined by the type of its operands. The following table describes the rules for determining the type of arithmetic expressions. The letters I, R, D, C and Z stand for INTEGER, REAL, DOUBLE PRECISION, COMPLEX and COMPLEX*16 respectively. An entry in the table represents the data type of the result when the operands are of the type indicated by the row and column in which the entry belongs. The column represents the type of the operand to the right of the operator, and the row represents the type of the operand to the left of the operator. The table is valid for all of the arithmetic operators.

| op | I*1 | I*2 | I*4 | R | D | C | Z |
|---|---|---|---|---|---|---|---|
| I*1 | I*1 | I*2 | I*4 | R | D | C | Z |
| I*2 | I*2 | I*2 | I*4 | R | D | C | Z |
| I*4 | I*4 | I*4 | I*4 | R | D | C | Z |
| R | R | R | R | R | D | C | Z |
| D | D | D | D | D | D | Z | Z |
| C | C | C | C | C | Z | C | Z |
| Z | Z | Z | Z | Z | Z | Z | Z |

*Notes:*

1. I*1 represents the `INTEGER*1` data type, I*2 represents the `INTEGER*2` data type, and I*4 represents the `INTEGER` or `INTEGER*4` data type.

2. The data type of the result obtained by dividing an integer datum by an integer datum is also of type INTEGER even though the mathematical result may not be an integer. This result is called the *integer quotient* and is defined as the integer part of the mathematical quotient.

3. Open Watcom FORTRAN 77 supports the double precision complex data type (COMPLEX*16) as an extension of the FORTRAN 77 language. Combining an operand of type DOUBLE PRECISION with an operand of type COMPLEX yields a result of type COMPLEX*16.

# 7.2 Character Expressions

Character expressions are used to describe computations involving operands of type CHARACTER, the concatenation operator (//) and left and right parentheses. The result of the computation is of type CHARACTER.

## 7.2.1 Character Operators

There is only one character operator, namely the concatenation operator (//). It requires two operands of type CHARACTER. If `x` is the left operand and `y` is the right operand, then the result is `y` concatenated to `x`. The length of the result is the sum of the lengths of the two operands. For example, the result of

```
'AAAAA'//'BBB'
```

is the string `AAAAABBB`.

## 7.2.2 Rules for Forming Character Expressions

The building blocks for character expressions are called *character primaries.* They are one of the following.

1. character constant
2. character symbolic constant
3. character variable reference
4. character array element reference
5. character substring reference
6. character function reference
7. ( character expression )

*Character expressions* are defined as follows:

1. character primary
2. character expression // character primary

A character expression is simply a sequence of character primaries, each separated by the concatenation operator (//). Rule 2 implies that character primaries are combined from left to right. Except in a character assignment statement, the operands in a character expression must not contain operands whose length specification is (*) unless the operand is a symbolic constant.

Note that, unlike arithmetic expressions, parentheses have no effect on the result of evaluating a character expression. For example, the result of the expressions

```
'A'//'B'//'C'
```

and

```
'A'//('B'//'C')
```

is identically the string `ABC`.

### 7.2.3 Character Constant Expressions

A *character constant expression* is a character expression in which all primaries are one of the following.

1. character constant
2. symbolic character constant
3. ( character constant expression )

As an extension to the FORTRAN 77 language, Open Watcom FORTRAN 77 supports the use of the intrinsic function **CHAR** in a character constant expression.

*Example:*

```
          CHARACTER*6 HELLO, WORLD
          PARAMETER (HELLO = 'Hello'//CHAR(0))
          PARAMETER (WORLD = 'world'//CHAR(7))
          PRINT *, HELLO, WORLD
          END
```

## 7.3 Relational Expressions

A relational expression is used to compare two arithmetic expressions or two character expressions. It is not possible to compare a character expression to an arithmetic expression. Evaluation of a relational expression produces a result of type logical.

### 7.3.1 Relational Operators

The following table lists the *relational operators* and the operation they perform.

| Operator | Relational Operation |
|---|---|
| .LT. | Less than |
| .LE. | Less than or equal |
| .EQ. | Equal |
| .NE. | Not equal |
| .GT. | Greater than |
| .GE. | Greater than or equal |

### 7.3.2 Form of a Relational Expression

The form of a *relational expression* is as follows.

```
e1 relop e2
```

***where:***

***relop*** is a relational operator.

***e1, e2*** are both arithmetic expressions or both character expressions.

### 7.3.2.1 Arithmetic Relational Expressions

An *arithmetic relational expression* is a relational expression in which `e1` and `e2` are both arithmetic expressions. An arithmetic relational expression has a value of true if the operands satisfy the relation specified by the relational operator and false otherwise.

A complex operand is only permitted when using either the .EQ. or .NE. relational operators. Open Watcom FORTRAN 77 allows operands of type COMPLEX*16.

### 7.3.2.2 Character Relational Expressions

*Character relational expressions* are relational expressions whose operands are of type CHARACTER. The value of a relation between character strings is established by using the *collating sequence* of the processor character set. The collating sequence is an ordering of the characters in the processor character set. Note, for example, that the EBCDIC character set has a different collating sequence than that of the ASCII character set. For example, `e1` is greater than `e2` if the value of `e1` follows the value of `e2` in the processor collating sequence. The value of a character relational expression depends on the collating sequence. In the case of the .NE. and .EQ. operators, the collating sequence has no effect.

*Example:*

```
IF( 'A' .LT. 'a' )THEN
    PRINT *, 'The processor character set'
    PRINT *, 'appears to be ASCII'
ELSE
    PRINT *, 'The processor character set'
    PRINT *, 'appears to be EBCDIC'
END IF
END
```

The above example is a crude test for determining the character set used on your processor.

It is possible to have operands of unequal length. In this case, the character string of smaller length is treated as if blanks were padded to the right of it to the length of the larger string. The relational operator is then applied.

# 7.4 Logical Expressions

Logical expressions are used to describe computations involving operands whose type is LOGICAL or INTEGER , logical operators and left and right parentheses. The result of the computation is of type LOGICAL unless both operands are of type INTEGER in which case the result of the computation is of type INTEGER.

## 7.4.1 Logical Operators

The following table lists the *logical operators* and the operation they perform.

| Operator | Logical Operation |
|---|---|
| .NOT. | Logical negation |
| .AND. | Logical conjunction |
| .OR. | Logical inclusive disjunction |
| .EQV. | Logical equivalence |
| .NEQV. | Logical non-equivalence |
| .XOR. | Exclusive or |

The logical operator .NOT. is a unary operator; all other logical operators are binary. The following tables describe the result of each operator when it is used with logical operands.

| x | .NOT. x |
|---|---|
| true | false |
| false | true |

| x | y | x .AND. y |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

| x | y | x .OR. y |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

| x | y | x .EQV. y |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | true |

| x | y | x .NEQV. y<br>x .XOR. y |
|---|---|---|
| true | true | false |
| true | false | true |
| false | true | true |
| false | false | false |

Note that the operators .NEQV. and .XOR. perform the same logical operation.

The following tables describe the result of the logical operators when they are used with integer operands. These operators apply to bits in the operand(s), hence we show only the result of operations on individual bits. The way to read the entries in the following tables is:

1. If the bit in "x" is 0 then the corresponding bit in ".NOT.x" is 1, and so on.

2. If the bit in "x" is 1 and the corresponding bit in "y" is 1 then the corresponding bit in "x.AND.y" is 1, and so on.

| x | .NOT. x |
|---|---|
| 0 | 1 |
| 1 | 0 |

| x | y | x .AND. y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

| x | y | x .OR. y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

| x | y | x .EQV. y |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

| x | y | x .NEQV. y<br>x .XOR. y |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

Note that the operators .NEQV. and .XOR. perform the same mathematical operation on bits.

As is the case with arithmetic operators, we must define rules in order to evaluate logical expressions. Again we define rules of precedence for logical operators which dictate the evaluation sequence of logical expressions. The following lists the logical operators in order of precedence.

1. .NOT. (highest precedence)
2. .AND.
3. .OR.
4. .EQV., .NEQV. and .XOR. (lowest precedence)

For example, in the expression

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

the .AND. operator has higher precedence than the .OR. operator so `B` and `C` are combined first using the .AND. operator. The result is then combined with `A` using the .OR. operator.

Parentheses can be used to alter the sequence of evaluation of logical expressions. If in the previous example we had written

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

then `A` and `B` would have been combined first.

## 7.4.2 Rules for Forming Logical Expressions

Logical primaries are the building blocks for logical expressions. They are one of the following.

1. logical or integer constant
2. symbolic logical or integer constant
3. logical or integer variable reference
4. logical or integer array element reference
5. logical or integer function reference
6. relational expression
7. ( logical or integer expression )

As was done with arithmetic expressions, a grammar can be defined which dictates the precedence relation among logical operators.

The .NOT. logical operator has highest precedence. We define a *logical factor* as:

1. logical primary
2. .NOT. logical primary

Next in the precedence hierarchy is the .AND. operator. We define a *logical term* as:

1. logical factor
2. logical term .AND. logical factor

A logical term is simply a sequence of logical factors, each separated by the .AND. operator. Rule (2) specifies that the logical factors are combined from left to right.

Next is the .OR. operator. We define a *logical disjunct* as:

1. logical term
2. logical disjunct .OR. logical term

A logical disjunct is simply a sequence of logical terms each separated by the .OR. operator. Rule (2) specifies that the logical terms are combined from left to right.

A *logical expression* can now be defined as follows.

1. logical disjunct
2. logical expression .EQV. logical disjunct
3. logical expression .NEQV. logical disjunct or logical expression .XOR. logical disjunct

A logical expression is therefore a sequence of logical disjuncts, each separated by the .EQV. operator or the .NEQV. or .XOR. operator. Rules (2) and (3) indicate that logical disjuncts are combined from left to right.

Consider the following example.

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

Since the .NOT. operator has highest precedence we first logically negate `B`. The result is then combined with `C` using the .AND. operator. That result is then combined with `A` using the .OR. operator to form the final result.

### 7.4.3 Logical Constant Expressions

A *logical constant expression* is a logical expression in which each primary is one of the following:

1. logical constant
2. symbolic logical constant
3. a relational expression in which each primary is a constant expression
4. ( logical constant expression )

The following are examples of a logical constant expression (assume that `A`, `B`, `C` and `D` are arithmetic constants appearing in **PARAMETER** statements).

```
.TRUE. .AND. .NOT. .FALSE.
'A' .LT. 'a'
A * B .GT. C * D
```

## 7.5 Evaluating Expressions

Four different types of operators have been discussed; arithmetic, character, relational and logical. It is possible to form an expression which contains all of these operators. Consider the following example.

```
A+B .LE. C .AND. X // Y .EQ. Z .AND. L
```

where `A`, `B` and `C` are of numeric type, `X`, `Y` and `Z` are of type CHARACTER and `L` is of type LOGICAL. In this expression, + is an arithmetic operator, // is a character operator, .EQ. is a relational operator and .AND. is a logical operator. Since we can mix these four types of operators, it is necessary to define a precedence among these four classes of operators. The following defines this precedence of operators.

1. arithmetic operators (highest precedence)
2. character operators
3. relational operators
4. logical operators (lowest precedence)

With this precedence any expression can now be evaluated without ambiguity.

## 7.6 Constant Expressions

A *constant expression* is an arithmetic constant expression, a character constant expression or a logical constant expression.

# 8 Assignment Statements

## 8.1 Introduction

Assignment statements are used to define entities. There are four different types of assignment.

1. Arithmetic
2. Logical
3. Statement label (ASSIGN)
4. Character

## 8.2 Arithmetic Assignment

The form of an *arithmetic assignment statement* is

```
v = e
```

***where:***

***v*** is a variable name or array element name of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX or double precision complex (COMPLEX*16).

***e*** is an arithmetic expression.

The following are examples of arithmetic assignment statements.

```
Y = X**2 + 4.0*X + 3.0
Z(10) = 4.3*(X+Y)
```

Executing an arithmetic assignment statement causes the evaluation of the arithmetic expression `e`, converting the type of the expression `e` to the type of `v`, and defining `v` with the result.

If `v` is of type INTEGER*1 or INTEGER*2, then the value of the expression `e` is first converted to type INTEGER. The resulting integer is then assigned to `v` in the following way.

1. If `v` is of type INTEGER*2 and the value of `e` is such that −32768 <= `e` <= 32767, `v` will be assigned the value of `e`. Otherwise, `v` will be undefined.
2. If `v` is of type INTEGER*1 and the value of `e` is such that −128 <= `e` <= 127, `v` will be assigned the value of `e`. Otherwise, `v` will be undefined.

# 8.3 Logical Assignment

The form of a *logical assignment statement* is

```
v = e
```

***where:***

***v*** is a variable name or array element name of type LOGICAL.

***e*** is a logical expression.

The following are examples of logical assignment statements.

```
LOG1 = .TRUE.
LOG2 = (X.GT.Y) .AND. (X.LT.Z)
LOG3(2) = LOG2 .EQV. LOG1
```

Executing a logical assignment statement causes the evaluation of the logical expression `e`, and defining `v` with the result. Note that the type of `v` and `e` must be LOGICAL.

# 8.4 Statement Label Assignment

The form of a *statement label assignment* is

```
ASSIGN s to i
```

***where:***

***s*** is a statement label.

***i*** is the name of an integer variable.

The following is an example of a statement label assignment statement.

```
ASSIGN 10 TO I
```

The result of executing an **ASSIGN** statement causes the integer variable `i` to be defined with the value of the statement label `s`. `s` must be the statement label of an executable statement or a format statement in the same program unit in which the **ASSIGN** statement appears. It is possible to change the value of `i` by executing another **ASSIGN** statement.

During execution when `i` is used in an assigned **GO TO** statement, an **ASSIGN** statement which defines `i` must have been executed prior to the execution of the assigned **GO TO** statement.

While the variable i is defined with a statement label, it should not be used in any other way other than in an assigned **GO TO** statement. Consider the following example.

*Example:*

```
10    ASSIGN 10 TO I
* Illegal use of an ASSIGNed variable
      PRINT *, I
```

The output produced by the **PRINT** statement is *not* the integer 10. Its value is undefined and should be treated that way.

# 8.5 Character Assignment

The form of a *character assignment statement* is

```
v = e
```

***where:***

***v*** is a character variable name, character array element, or character substring.

***e*** is a character expression.

The following are examples of character assignment statements.

```
CHARACTER*20 C,D(5)
C='ABCDEF'
C(3:5)='XYZ'
D(5)(14:15)='12'
```

Executing a character assignment statement causes the evaluation of the character expression e and the definition of v with the result.

None of the character positions defined in v may be referenced in e. The following example is illegal since the 4th and 5th character positions of A appear on the left and right hand side of the equal sign.

*Example:*

```
* Illegal character assignment.
      CHARACTER*10 A,B*5
      A(2:6) = A(4:5) // B
```

The length of v and e may be different. If the length of v is less than the length of e then the assignment has the effect of truncating e from the right to the length of v. If the length of v is greater than the length of e, the value assigned to v is the value of e padded on the right with blanks to the length of v.

# 8.6 Extended Assignment Statement

Open Watcom FORTRAN 77 supports an extension to the FORTRAN 77 assignment statement, namely the *extended assignment statement* shown here:

```
v  =  v  =  v  = ... =  v  = e
1     2     3           n
```

***where:***

***v'i*** must be one of the following:

1. Variable names or array element names of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX or double precision complex (COMPLEX*16).
2. Variable names or array element names of type LOGICAL.
3. Character variable names, character array elements, or character substrings.

***e*** must be one of the following and must follow the rules of the arithmetic, logical or character assignment statements:

1. An arithmetic expression.
2. A logical expression.
3. A character expression.

The extended assignment statement is equivalent to the following individual statements.

```
v     =    e
 n
v     =    v
 n-1        n
      .
      .
      .
v     =    v
 2          3
v     =    v
 1          2
```

When using an extended assignment statement involving variables of mixed type, it is important to understand the exact way in which the assignments are performed. Assignment of each variable is made using the value of the variable to its immediate right, starting with the rightmost variable which is assigned the value of the expression. To help make this clear, consider the following program.

*Example:*

```
CHARACTER C1*10, C2*5, C3*7
LOGICAL L1, L2, L3
INTEGER*2 K, L
I = S = J = T = 1.25
PRINT *, I, S, J, T
I = K = J = L = 70000
PRINT *, I, K, J, L
C1 = C2 = C3 = 'ABCDEFGHIJKL'
PRINT *, C1, C2, C3
L1 = L2 = L3 = .TRUE.
PRINT *, L1, L2, L3
END
```

The output from this program would be:

```
   1 1.0000000          1 1.2500000
 4464       4464       4464       4464
ABCDE      ABCDEABCDEFG
   T           T          T
```

Note that variables `K` and `L` are of type INTEGER*2 and cannot contain any value greater than 32767. Truncation resulted and this value (4464) was propagated to the left.

# 9 Program Structure Control Statements

## 9.1 Introduction

The use of structured programming statements has been found to encourage better programming and design practices among beginners, and aids the more experienced programmer in writing error-free programs.

The format of these statements and their blocks is illustrated below. Following this, the use and meaning of each statement is described and illustrated with examples. In each of these illustrations, the blocks are denoted by `statement(s)` and are delimited by control statements.

In the descriptions, `logical-expression` can also be an integer expression, in which case the result of the integer expression is compared for inequality to the integer value 0.

*Example:*

```
        IF( LEN - 1 )THEN
```

In the preceding example, the expression `LEN - 1` is interpreted as `LEN - 1 .NE.  0.`

## 9.2 IF - ELSE - END IF

The **ELSE** portion of this construct is optional, thus there are two possible formats.

```
(a) IF( logical-expression )THEN    [: block-label]
        statement(s)
    END IF

(b) IF( logical-expression )THEN    [: block-label]
        statement(s)
    ELSE
        statement(s)
    END IF
```

This construct is an enhancement of the FORTRAN logical **IF** statement. If the value of the parenthesized logical expression is true in (a), the block of statements following the **IF** statement is executed, after which control passes to the statement following the **END IF** statement; otherwise, control will pass directly to the statement following the **END IF** statement. When the **ELSE** statement is used and the logical expression is true, the block of statements between the **IF** and the **ELSE** statements is executed and then control passes to the statement following the **END IF** statement; otherwise the block of statements following **ELSE** statement is executed and then control passes to the statement following the **END IF** statement.

An optional block label may be specified with the **IF** statement (see the **CYCLE**, **EXIT** or **QUIT** statement for more information).

Examples follow which illustrate the use of the two formats.

*Example:*

```
IF( I .EQ. 0 )THEN
    PRINT *, 'I IS ZERO'
    I = 1
END IF
```

If variable `I` is zero when the **`IF`** statement is executed, the string `I IS ZERO` will be printed, variable `I` will be assigned the value 1, and the statement following the **`END IF`** will be executed. If variable `I` is not zero when the **`IF`** statement is executed, control will pass to the statement following the **`END IF`** statement.

*Example:*

```
IF( A .GT. B )THEN
    PRINT *, 'A GREATER THAN B'
    A = A - B
ELSE
    PRINT *, 'A NOT GREATER THAN B'
END IF
```

If the value of variable `A` is greater than the value of variable `B` when this **`IF`** statement is executed, the string `A GREATER THAN B` will be printed and variable `A` will be assigned the value of the expression `A - B.` Control will then pass to the statement following the **`END IF`** statement.

If the value of variable `A` is not greater than the value of variable `B` when the **`IF`** statement is executed, the string `A NOT GREATER THAN B` will be printed and control will pass to the statement following the **`END IF`** statement.

# 9.3 ELSE IF

A further enhancement of the IF-THEN-ELSE construct is the **`ELSE IF`** statement which may be used in the following two formats:

```
(a) IF( logical-expression-1 )THEN    [: block-label]
        statement(s)
    ELSE IF( logical-expression-2 )THEN
        statement(s)
    ...
    END IF

(b) IF( logical-expression-1 )THEN    [: block-label]
        statement(s)
    ELSE IF( logical-expression-2 )THEN
        statement(s)
    ...
    ELSE
        statement(s)
    END IF
```

The presence of the "..." in the above formats indicates that the **ELSE IF** statement may be repeated as often as desired. If the value of `logical-expression-1` is true in case (a), the block of statements following the **IF** statement up to the first **ELSE IF** statement is executed, after which control passes to the statement following the **END IF** statement; otherwise, control will pass to the first **ELSE IF** statement. If the value of `logical-expression-2` is true, the block of statements following the first **ELSE IF** statement up to the next **ELSE IF** statement or **END IF** statement is executed, after which control passes to the statement following the **END IF** statement; otherwise, control will pass to the next **ELSE IF** statement, if there is one, or directly to the statement following the **END IF** statement. When the **ELSE** statement is used, as in case (b), and the values of all the logical expressions in the **IF** and **ELSE IF** statements are false, the block of statements following the **ELSE** statement is executed and then control passes to the statement following the **END IF** statement. An optional block label may be specified with the **IF** statement (see the **CYCLE**, **EXIT** or **QUIT** statement for more information).

Examples follow which illustrate the use of the two formats.

*Example:*

```
IF( I .EQ. 0 )THEN
    PRINT *, 'I IS ZERO'
ELSE IF( I .GT. 0 )THEN
    PRINT *, 'I IS GREATER THAN ZERO'
END IF
```

If variable `I` is zero when the **IF** statement is executed, the string `I IS ZERO` will be printed and the statement following the **END IF** statement will be executed. If variable `I` is not zero when the **IF** statement is executed, control will pass to the **ELSE IF** statement. If variable `I` is greater than zero, the string `I IS GREATER THAN ZERO` will be printed and the statement following the **END IF** statement will be executed. If variable `I` is less than zero then nothing would be printed and control passes from the **ELSE IF** statement to the statement following the **END IF** statement.

*Example:*

```
IF( A .GT. B )THEN
    PRINT *, 'A GREATER THAN B'
    A = A - B
ELSE IF( A .LT. B )THEN
    PRINT *, 'A LESS THAN B'
    A = B - A
ELSE
    PRINT *, 'A EQUAL TO B'
    A = 0.0
END IF
```

If the value of variable `A` is greater than the value of variable `B` when the **IF** statement is executed, the string `A GREATER THAN B` will be printed and variable `A` will be assigned the value of the expression `A - B.` Control will then pass to the statement following the **END IF** statement.

If the value of variable `A` is not greater than the value of variable `B` when the **IF** statement is executed, control passes to the **ELSE IF** statement. If the value of variable `A` is less than the value of variable `B`, the string `A LESS THAN B` will be printed and variable `A` will be assigned the value of the expression `B - A.` Control will then pass to the statement following the **END IF** statement.

If the value of variable `A` is not less than the value of variable `B` when the **ELSE IF** statement is executed, the string `A EQUAL TO B` will be printed and variable `A` will be assigned the value zero. Control will pass to the statement following the **END IF** statement.

# 9.4 DO - END DO

```
DO init-expr,end-value[,inc-value] [: block-label]
    statement(s)
END DO
```

This extension to FORTRAN 77 allows the creation of DO-loops without the introduction of statement numbers. An optional block label may be specified (see the **CYCLE**, **EXIT** or **QUIT** statement for more information). The **END DO** statement is used to indicate the end of the range of its corresponding **DO** statement. A statement number may not be specified in the corresponding **DO** statement. Nested DO-loops of this form require separate **END DO** statements to terminate the range of the corresponding **DO** statement. Since a statement number may appear on the **END DO** statement, the number may be used to terminate outer DO-loops. This is not a recommended practice (a **CONTINUE** statement or a structured **DO** statement should be used). A transfer of control from within the DO-loop to a statement number on the **END DO** statement is treated in the same manner as if the word **CONTINUE** had been used instead of **END DO**.

Some examples follow.

*Example:*

```
        DO I = 1, 3
            DO J = 1, 5
                PRINT *, MATRIX( I, J )
            END DO
        END DO
```

The above is equivalent to the following example which uses statement numbers.

*Example:*

```
        DO 10 I = 1, 3
            DO 20 J = 1, 5
                PRINT *, MATRIX( I, J )
20          CONTINUE
10      CONTINUE
```

The next example demonstrates the use of a **GO TO** statement to control execution of all or part of a DO-loop.

*Example:*

```
        DO I = 1, 3
            DO J = 1, 5
                PRINT *, 'INNER LOOP - J=', J
                IF( J .LE. 3 )GO TO 20
                PRINT *, 'J > 3'
20          END DO
            PRINT *, 'OUTER LOOP - J=', J
        END DO
```

A result of this example is that the character string `J > 3` is printed 6 times (i.e., twice for each iteration of the outer loop). Of course there is a much better way of coding this algorithm using the IF-END IF construct. The example is included to illustrate the behaviour of transfers of control to an **END DO** statement. The following example is an equivalent algorithm to the one above but the intent is much clearer.

*Example:*

```
      DO I = 1, 3
          DO J = 1, 5
              PRINT *, 'INNER LOOP - J=', J
              IF( J .GT. 3 )THEN
                  PRINT *, 'J > 3'
              END IF
          END DO
          PRINT *, 'OUTER LOOP - J=', J
      END DO
```

# 9.5 DO WHILE - END DO

```
DO WHILE (e) [: block-label]
    statement(s)
END DO
```

This extension to FORTRAN 77 allows the creation of DO-loops without iterative techniques. Instead, the DO-loop is executed while the parenthesized expression is true. The logical expression is evaluated before entry to the DO-loop. If the value is false, control is transferred to the statement following the **END DO** statement. If the logical expression if true, the statements of the DO-loop are executed. When the **END DO** statement is reached, the expression is re-evaluated and program control proceeds as previously described. An optional block label may be specified (see the **CYCLE**, **EXIT** or **QUIT** statement for more information).

An optional statement number can be specified after the **DO** keyword. When the **END DO** statement is used to indicate the end of the range of its corresponding **DO WHILE** statement, a statement number may not be specified.

Some examples follow.

*Example:*

```
      I = 1
      DO WHILE( I .LE. 3 )
          J = 1
          DO WHILE( J .LE. 5 )
              PRINT *, MATRIX( I, J )
          END DO
      END DO
```

The above is equivalent to the following example which uses statement numbers.

*Example:*

```
      I = 1
      DO 10 WHILE( I .LE. 3 )
          J = 1
          DO 20 WHILE( J .LE. 5 )
              PRINT *, MATRIX( I, J )
20        CONTINUE
10    CONTINUE
```

The next example demonstrates the use of a **GO TO** statement to control execution of all or part of a DO-loop.

*Example:*

```
          I = 1
          DO WHILE( I .LE. 3 )
              J = 1
              DO WHILE( J .LE. 5 )
                  PRINT *, 'INNER LOOP - J=', J
                  IF( J .LE. 3 )GO TO 20
                  PRINT *, 'J > 3'
    20        END DO
              PRINT *, 'OUTER LOOP - J=', J
          END DO
```

A result of this example is that the character string `J > 3` is printed 6 times (i.e., twice for each iteration of the outer loop). Of course there is a much better way of coding this algorithm using the IF-END IF construct. The example is included to illustrate the behaviour of transfers of control to an **END DO** statement. The following example is an equivalent algorithm to the one above but the intent is much clearer.

*Example:*

```
          I = 1
          DO WHILE( I .LE. 3 )
              J = 1
              DO WHILE( J .LE. 5 )
                  PRINT *, 'INNER LOOP - J=', J
                  IF( J .GT. 3 )THEN
                      PRINT *, 'J > 3'
                  END IF
              END DO
              PRINT *, 'OUTER LOOP - J=', J
          END DO
```

# 9.6 LOOP - END LOOP

```
LOOP    [: block-label]
    statement(s)
END LOOP
```

This extension to FORTRAN 77 causes the statements between the **LOOP** and **END LOOP** statements to be repeated until control is transferred out of the loop, usually by an **EXIT** or **QUIT** statement. An optional block label may be specified (see the **CYCLE**, **EXIT** or **QUIT** statement for more information). An example follows:

*Example:*

```
LOOP
    READ *, X
    IF( X .EQ. 99.0 )EXIT
    PRINT *, X
END LOOP
```

The above statements cause values to be read and printed, one to a line, until the value 99.0 is read. When variable X has this value, the logical expression in the **IF** statement evaluates as true and the **EXIT** statement causes a transfer of control to the statement following the **END LOOP** statement. The **EXIT** statement is discussed in more detail in a later section.

# 9.7 WHILE - END WHILE

```
WHILE( logical-expression )DO      [: block-label]
    statement(s)
END WHILE
```

This extension to FORTRAN 77 causes its block of code to be executed repeatedly while the parenthesized logical expression is true. The logical expression is evaluated before entry to the block. If the value is false, control passes to the statement following the **END WHILE** statement. If the logical expression is true, the statements of the block are executed. When the **END WHILE** statement is reached, the **WHILE** logical expression is re-evaluated and the above program control decisions are repeated. An optional block label may be specified (see the **CYCLE**, **EXIT** or **QUIT** statement for more information). An example follows:

*Example:*

```
WHILE( J .GT. 0 )DO
    A(J) = B(I + J)
    J = J - 1
END WHILE
```

If variable J is zero or negative when the **WHILE** statement is executed, the **WHILE** block of code will be by-passed and the statement following the **END WHILE** statement will be executed.

If variable J is greater than zero when the **WHILE** statement is executed, the **WHILE** block will be executed repeatedly until J becomes equal to zero. The effect of this loop will be to assign values to elements of array A from array B, starting with the element of A corresponding to the initial value of variable J and working backwards down the array to element 1.

# 9.8 WHILE - Executable-statement

```
WHILE( logical-expression )stmt
```

***where:***

***stmt*** is an executable statement. Only certain executable statements are are allowed. See the section entitled "Classifying Statements" on page 9 in the chapter entitled "FORTRAN Statements" for a list of allowable statements.

This control statement is another form of the WHILE construct.

*Example:*

```
WHILE( I .GT. 0 )EXECUTE A
```

When this statement is executed, if the logical expression is not true, control passes to the next statement. If the expression is true, REMOTE-block `A` (assumed to be defined elsewhere in the program unit) is executed, and the logical expression is re-evaluated. This is repeated until the logical expression, when evaluated, is false; control then passes to the next statement.

# 9.9 UNTIL

```
LOOP    [: block-label]
    statement(s)
UNTIL( logical-expression )
```

or

```
WHILE( logical-expression )DO      [: block-label]
    statement(s)
UNTIL( logical-expression )
```

The **UNTIL** statement, an extension to FORTRAN 77, may be combined with either a **LOOP** or **WHILE** statement by replacing the **END LOOP** or **END WHILE** statement. It provides a way of specifying a condition to be tested at the end of each iteration of a loop, which will determine whether or not the loop is repeated. After all of the statements in the block have been executed, the logical expression in the **UNTIL** statement is evaluated. If the result of the condition is false, the loop is repeated; otherwise, control passes to the statement following the **UNTIL** statement.

In the following example, the statements between the **LOOP** and the **UNTIL** statements are executed until the value of variable `X` is greater than 10.0.

*Example:*

```
X = 1.0
LOOP
    PRINT *, X, SQRT( X )
    X = X + 1.0
UNTIL( X .GT. 10.0 )
```

# 9.10 SELECT - END SELECT

```
SELECT [CASE] (e) [FROM]  [: block-label]
CASE ( case-list )
    statement (s)
CASE ( case-list )
    statement (s)
    .
    .
    .
CASE DEFAULT
    statement(s)
END SELECT
```

***where:***

***case-list*** is a list of one or more *cases* separated by commas. A *case* is either

**(a)** a single integer, logical or character constant expression or

**(b)** an integer, logical or character constant expression followed by a colon followed by another expression or the same type. This form of a case defines a range of values consisting of all integers or characters greater than or equal to the value of the expression preceding the colon and less than or equal to the value of the expression following the colon.

The **SELECT** construct, an extension to FORTRAN 77, is similar in concept to the FORTRAN computed **GO TO** statement. It allows one of a number of blocks of code (case blocks) to be selected for execution by means of an integer expression in the **SELECT** statement.

The **SELECT** statement keywords, **CASE** and **FROM**, are optional. The **SELECT** statement may contain a block label (see the **CYCLE**, **EXIT** or **QUIT** statements for more information).

Each block must be started with a **CASE** statement; however, the last block may begin with a **CASE DEFAULT** statement. The **CASE DEFAULT** block is optional. In order to retain compatibility with earlier versions of WATCOM FORTRAN 77 compilers, the **OTHERWISE** statement may be used in place of the **CASE DEFAULT** statement. The last block is ended by the **END SELECT** statement. The number of case blocks is optional, from one to many; however, it is recommended that the SELECT construct not be used for fewer than 3 case blocks. The conditional execution of one or two blocks of code is handled more efficiently by the IF-THEN-ELSE construct.

A particular case value or range of values must not be contained in more than one CASE-block. For example, the following is illegal:

*Example:*

```
    * Illegal SELECT block - case value in more
    * than one CASE block.
          SELECT CASE ( I - 3 )
          CASE (1,3,7:10)
              statement(s)
          CASE (5,6,8)
              statement(s)
          CASE (-3:-2+4)
              statement(s)
          END SELECT
```

The second CASE-block includes 8 which is already handled by the first CASE-block. As well, the third CASE-block handles cases –3, –2, –1, 0, 1, 2 but the first CASE-block also handles case 1. Thus the second and third CASE-ranges are in error.

When the **SELECT** statement case expression is evaluated as *i,* the case block whose range contains *i* is executed and control passes to the statement following the **END SELECT** statement. If no range contains *i,* control is transferred to the statement following the **CASE DEFAULT** statement, if one is specified. If the **CASE DEFAULT** block is omitted and the case expression is out of range when the **SELECT** statement is executed (that is, none of the CASE-blocks handles the particular expression value), control is passed to the statement following the **END SELECT** statement and none of the CASE-blocks is executed.

*Example:*

```
          SELECT CASE ( I )
          CASE (1)
              Y = Y + X
              X = X * 3.2
          CASE (2)
              Z = Y**2 + X
              PRINT *, X, Y, Z
          CASE (3)
              Y = Y * 13. + X
              X = X - 0.213
          CASE (4)
              Z = X**2 + Y**2 - 3.0
              Y = Y + 1.5
              X = X * 32.0
              PRINT *, 'CASE 4', X, Y, Z
          END SELECT
```

This example will execute in the manner described below for each of the possible values of variable `I`.

**(i)** `I` is zero or negative:
- control will pass to the statement after the **END SELECT** statement

**(ii)** `I = 1:`
- the value of `X` will be added to `Y`
- `X` will be multiplied by 3.2
- control will pass to the statement after the **END SELECT** statement

**(iii)** `I = 2:`
- `Z` will be assigned the value of the expression `Y**2 + X`
- the values of `X`, `Y` and `Z` will be printed
- control will pass to the statement after the **END SELECT** statement

**(iv)** `I = 3:`
- `Y` will be assigned the value of the expression `Y * 13.  + X`
- 0.213 will be subtracted from `X`
- control will pass to the statement after the **`END SELECT`** statement

**(v)** `I = 4:`
- `Z`, `Y` and `X` will be assigned new values
- the string `CASE 4,` followed by the values of `X`, `Y` and `Z` will be printed
- control will pass to the statement after the **`END SELECT`** statement

**(vi)** `I = 5, 6, ...:`
- control will pass to the statement after the **`END SELECT`** statement

**`CASE DEFAULT`** allows a block of code to be specified for execution when the **`SELECT`** expression is out of range. It must follow all CASE-blocks and thus is ended by the **`END SELECT`** statement. The **`CASE DEFAULT`** statement terminates the previous and last CASE-block. Note that only one **`CASE DEFAULT`** block may be specified in a SELECT construct.

If a **`CASE DEFAULT`** block were included in the above example, it would be executed in cases (i) and (vi) of the description. After a **`CASE DEFAULT`** block is executed, control then passes to the statement after the **`END SELECT`** statement.

Empty or null case blocks are permitted (that is, two **`CASE`** statements with no statements between). The net result of executing a null CASE-block is to effectively bypass the SELECT construct.

*Example:*

```
SELECT CASE ( I * 4 - J )
CASE (-10 : -5)
    PRINT *,'First case:'
    PRINT *,'-10 <= I*4-J <= -5'
CASE (-4 : 2)
    PRINT *,'Second case:'
    PRINT *,'-4 <= I*4-J <= 2'
CASE (3, 5, 7)
    PRINT *,'Third case:'
    PRINT *,'I*4-J is one of 3, 5 or 7'
CASE (4, 6, 8:10)
    PRINT *,'Fourth case:'
    PRINT *,'I*4-J is one of 4, 6, 8, 9 or 10'
CASE DEFAULT
    PRINT *,'All other cases:'
    PRINT *,'I*4-J < -10 or I*4-J > 10'
END SELECT
```

This example will execute in the manner described below for each of the possible values of expression `I*4-J.`

**(i)** expression < –10
- control will pass to the statement after the **`CASE DEFAULT`** statement
- the string `All other cases:` will be printed
- the string `I*4-J < -10 or I*4-J > 10` will be printed

**(ii)** –10 <= expression <= –5:
- control will pass to the statement after the first **`CASE`** statement
- the string `First case:` will be printed

- the string `-10 <= I*J-4 <= -5` will be printed
- control will pass to the statement after the **END SELECT** statement

**(iii)** –4 <= expression <= 2:
- control will pass to the statement after the second **CASE** statement
- the string `Second case:` will be printed
- the string `-4 <= I*J-4 <= 2` will be printed
- control will pass to the statement after the **END SELECT** statement

**(iv)** expression = 3, 5 or 7:
- control will pass to the statement after the third **CASE** statement
- the string `Third case:` will be printed
- the string `I*J-4 is one of 3, 5 or 7` will be printed
- control will pass to the statement after the **END SELECT** statement

**(v)** expression = 4, 6, 8, 9 or 10:
- control will pass to the statement after the fourth **CASE** statement
- the string `Fourth case:` will be printed
- the string `I*J-4 is one of 4, 6, 8, 9 or 10` will be printed.
- control will pass to the statement after the **END SELECT** statement

**(vi)** expression > 10:
- control will pass to the statement after the **CASE DEFAULT** statement
- the string `All other cases:` will be printed
- the string `I*4-J < -10 or I*4-J > 10` will be printed

# 9.11 EXECUTE and REMOTE BLOCK

```
EXECUTE   name
     .
     .
     .
REMOTE BLOCK name
    statement(s)
END BLOCK
```

***where:***

***name*** is a valid FORTRAN symbolic name.

The **EXECUTE** statement, an extension to FORTRAN 77, allows a named block of code to be executed. The named block of code may be defined anywhere in the same program unit and is delimited by the **REMOTE BLOCK** and **END BLOCK** statements. Executing a REMOTE-block is similar in concept to calling a subroutine, with the advantage that shared variables do not need to be placed in a common block or passed in an argument list. In addition there is less overhead involved in executing a REMOTE-block than in calling a subroutine (in both amount of object code and execution time). When execution of the REMOTE-block is complete, control returns to the statement following the **EXECUTE** statement which invoked it.

This feature is helpful in avoiding duplication of code for a function (such as I/O) required in a number of places throughout a program. It can also be an aid to writing a well-structured program.

Each REMOTE-block within the same program unit must have a different name and it must not be a subprogram or variable name. Note that a REMOTE-block is local to the program unit in which it is defined and may not be referenced (executed) from another program unit.

REMOTE-blocks may be defined anywhere in the program unit except as follows.

1. They must follow all specification statements.
2. They must not be defined within a control structure.

If a **REMOTE BLOCK** statement is encountered during execution, control is transferred to the statement following the corresponding **END BLOCK** statement.

Note that the nested definition of REMOTE-blocks is not permitted.

*Example:*

```
EXECUTE A
PRINT *, 'FIRST'
    .
    .
    .
EXECUTE A
PRINT *, 'SECOND'
    .
    .
    .
REMOTE BLOCK A
    I = I + 1
    PRINT *, 'I=', I
END BLOCK
```

Both **EXECUTE** statements will cause REMOTE-block `A` to be executed. That is, variable `I` will be incremented and its value will be printed. When the block has been executed by the first **EXECUTE** statement, control returns to the **PRINT** statement following it and the word `FIRST` is printed. Similarly, when the block is executed by the second **EXECUTE** statement, control returns to the **PRINT** statement following it and the word `SECOND` is printed.

REMOTE-blocks may be executed from other REMOTE-blocks. For example, REMOTE-block `A` might contain the statement `EXECUTE B,` where `B` is a REMOTE-block defined elsewhere in the program unit. The execution of REMOTE-blocks from other REMOTE-blocks may take place to any level; however, the recursive execution of REMOTE-blocks is not permitted, either directly or through a chain of **EXECUTE** statements. Attempts to execute REMOTE-blocks recursively are detected as errors at execution time.

# 9.12 GUESS-ADMIT-END GUESS

```
GUESS    [: block-label]
    statement(s)
ADMIT
    statement(s)
ADMIT
    statement(s)
    .
    .
    .
ADMIT
    statement(s)
END GUESS
```

The GUESS-ADMIT-END GUESS structure is a rejection mechanism which is useful when sets of statements are to be conditionally chosen for execution, but not all of the conditions required to make a selection are available beforehand. It is an extension to FORTRAN 77. The sets of statements to be chosen may be thought of as alternatives, the first alternative being statements immediately after the **GUESS** statement. Execution begins with the statements in the first alternative. If a condition is detected which indicates that the first alternative was the wrong choice, a **QUIT** statement may be executed to cause control to be passed to the statements after the **ADMIT** statement (i.e., the second alternative). A **QUIT** statement within the second alternative passes control to the third alternative, etc. A **QUIT** statement within the last alternative passes control to the statement after the **END GUESS** statement. If an alternative completes execution without encountering a **QUIT** statement (i.e., all statements are executed up to the next **ADMIT** statement) then control is passed to the statement after the **END GUESS** statement. An optional block label may be specified following the keyword **GUESS** (see the **QUIT** statement for more information).

In the following example, two sets of codes and numbers are read in and some simple sequence checking is performed. If a sequence error is detected an error message is printed and processing terminates; otherwise the numbers are processed and another pair of numbers is read.

*Example:*

```
LOOP : PRLOOP
    GUESS
        LINE = LINE + 1
        READ *, ICODE, X
        AT END, QUIT :PRLOOP
        IF( ICODE .NE. 1 )QUIT
        LINE = LINE + 1
        READ *, ICODE, Y
        AT END, QUIT
        IF( ICODE .NE. 2 )QUIT
        PRINT *, X, Y
        CALL PROCES(X, Y)
    ADMIT
        PRINT *, 'INVALID SEQUENCE: LINE =', LINE
        QUIT :PRLOOP
    END GUESS
END LOOP
```

The above example attempts to read a code and number. If an end of file occurs then the loop is terminated by the **QUIT** statement.

If the code is not 1 then we did not get what we expected and an error situation has arisen. Control is passed to the statement following the **ADMIT** statement. An error message is printed and the loop is terminated by the **QUIT** statement.

If the code is 1 then a second code and number are read. If an end of file occurs then we are missing a set of data and an error situation has arisen. Control is passed to the statement following the **ADMIT** statement. An error message is printed and the loop is terminated by the **QUIT** statement. Similarly if the expected code is not 2 an error situation has arisen. Control is passed to the statement following the **ADMIT** statement. An error message is printed and the loop is terminated by the **QUIT** statement.

If the second code is 2, the values of variables `X` and `Y` are printed. A subroutine is then called to process the data. Control resumes at the statement following the **END GUESS** statement. Since this statement is an **END LOOP**, control is transferred to the beginning of the loop.

The above example illustrates the point that all the information required to make a choice (in this case between a valid set of data and an invalid set) is not available from the beginning. In this case we make an assumption that the data values are correct (our hypothesis) and then test the assumption at various points in the algorithm. If any of the tests fail we reject the hypothesis (and, perhaps, select a new hypothesis).

It should be noted that no alternative selection need be coded (i.e., we need not use any ADMIT-blocks). This is illustrated in the following example.

*Example:*

```
GUESS
    X=SQRT( X )
    IF( X .LT. EPS )QUIT
    X=Y+SQRT(Y)
    IF( X .LT. EPS )QUIT
    CALL INTGRT( X, Y )
END GUESS
```

It might be noted that the IF-ELSE-END IF construct is simply a specific instance of the more general GUESS-ADMIT-END GUESS construct wherein the data values are known beforehand (as could be illustrated using the previous example).

# 9.13 QUIT

```
QUIT    [ : block-label]
```

The **QUIT** statement may be used to transfer control to the first executable statement following the terminal statement of the block in which it is contained.

When transferring out of a loop, control is passed to the statement following the **END DO**, **END WHILE**, **END LOOP** or **UNTIL** statement.

When transferring out of a GUESS block, control is passed to the statement after the next **ADMIT** or **END GUESS** statement.

When transferring out of an IF-block or SELECT-block, control is passed to the statement after the corresponding **END IF** or **END SELECT** statement.

When transferring out of a REMOTE-block, control passes to the statement following the **EXECUTE** statement that invoked the REMOTE-block.

If no block label is specified in the **QUIT** statement, control is transferred from the immediately enclosing structure. If several structures or DO-loops are nested, it is possible to exit from any one of them by specifying the block label of the corresponding block structure.

The **QUIT** statement is most commonly used as the statement in a logical **IF** or **AT END** statement but may also be used to cause an unconditional transfer of control. (The **AT END** statement is described in a subsequent section).

Examples of the **QUIT** statement with and without a block label follow.

*Example:*

```
CHARACTER CH
READ *, CH
GUESS
    IF( CH .LT. 'a' )QUIT
    IF( CH .GT. 'z' )QUIT
    PRINT *, 'Lower case letter'
ADMIT
    IF( CH .LT. 'A' )QUIT
    IF( CH .GT. 'Z' )QUIT
    PRINT *, 'Upper case letter'
ADMIT
    IF( CH .LT. '0' )QUIT
    IF( CH .GT. '9' )QUIT
    PRINT *, 'Digit'
ADMIT
    PRINT *, 'Special character'
END GUESS
END
```

The above statements read and print values until an end of file occurs. At that point control is passed to the **QUIT** statement, as specified by the **AT END** statement. The **QUIT** statement causes control to continue with the statement after the **END LOOP** statement.

*Example:*

```
          CHARACTER RECORD(80)
          LOOP : RDREC
              READ(5,100) RECORD
              AT END, STOP
              DO I = 1, 80
                  IF( RECORD(I) .LT. '0'
         +              .OR. RECORD(I) .GT. '9' )QUIT : RDREC
              END DO
              WRITE(6,101) RECORD
          END LOOP
          PRINT *, 'INVALID RECORD'
```

The above example reads in records and verifies that they contain only numeric data. The **QUIT** statement is within two levels of nesting: the DO-loop and the LOOP-END LOOP structure. If a non-numeric character is found, the `QUIT : RDREC` statement will cause control to be passed to the **PRINT** statement after the **END LOOP** statement.

# 9.14 EXIT

```
EXIT    [ : block-label]
```

The **EXIT** statement is used to transfer control:

1. from within a loop (DO, DO WHILE, WHILE or LOOP) to the statement following the loop,
2. from within a GUESS or ADMIT block to the statement following the **ENDGUESS** statement, or
3. from within a remote block to the statement following the **EXECUTE** statement that invoked the remote block.

When transferring out of a loop, control is passed to the statement following the **END DO**, **END WHILE**, **END LOOP** or **UNTIL** statement.

When transferring out of a GUESS block, control is passed to the statement after the corresponding **END GUESS** statement.

When transferring out of a REMOTE-block, control passes to the statement following the **EXECUTE** statement that invoked the REMOTE-block.

If no block label is specified in the **EXIT** statement, control is transferred from the immediately enclosing structure. If several structures or DO-loops are nested, it is possible to exit from any one of them by specifying the block label of the corresponding block structure.

The **EXIT** statement is most commonly used as the statement in a logical **IF** or **AT END** statement but may also be used to cause an unconditional transfer of control. (The **AT END** statement is described in a subsequent section).

Examples of the **EXIT** statement with and without a block label follow.

*Example:*

```
LOOP
    READ *, X
    AT END, EXIT
    PRINT *, X
END LOOP
```

The above statements read and print values until an end of file occurs. At that point control is passed to the **EXIT** statement, as specified by the **AT END** statement. The **EXIT** statement causes control to continue with the statement after the **END LOOP** statement.

*Example:*

```
CHARACTER RECORD(80)
LOOP : RDREC
    READ(5,100) RECORD
    AT END, STOP
    DO I = 1, 80
        IF( RECORD(I) .LT. '0'
+             .OR. RECORD(I) .GT. '9' )EXIT : RDREC
    END DO
    WRITE(6,101) RECORD
END LOOP
PRINT *, 'INVALID RECORD'
```

The above example reads in records and verifies that they contain only numeric data. The **EXIT** statement is within two levels of nesting: the DO-loop and the LOOP-END LOOP structure. If a non-numeric character is found, the `EXIT : RDREC` statement will cause control to be passed to the **PRINT** statement after the **END LOOP** statement.

# 9.15 CYCLE

```
CYCLE   [ : block-label]
```

The **CYCLE** statement is used to cause a transfer of control from within a loop to the terminal statement of a corresponding **DO**, **DO WHILE**, **WHILE** or **LOOP** statement. If `block-label` is present then control is transferred to the terminal statement of the block identified by that block label.

If no block label is specified in the **CYCLE** statement, control is transferred to the terminal statement of the immediately enclosing loop structure. If several loop structures are nested, it is possible to cycle to the terminal statement of any one of them by specifying the block label of the corresponding block structure.

The **CYCLE** statement is most commonly used as the statement in a logical **IF** statement but may also be used to cause an unconditional transfer of control.

Examples of the **CYCLE** statement with and without a block label follow.

*Example:*

```
LOOP
    WRITE( UNIT=*, FMT='(A)' ) 'Enter a number'
    READ( UNIT=*, FMT='(F10.4)', IOSTAT=IOS ) X
    IF( IOS .NE. 0 ) CYCLE
    IF( X .LT. 0 ) EXIT
    PRINT *, X, SQRT( X )
END LOOP
```

The above statements read and print values until a negative integer value is entered. If an input error occurs, the input operation (READ) is retried using the **CYCLE** statement. The **CYCLE** statement causes control to resume at the **END LOOP** statement which then immediately transfers control to the statement following the **LOOP** statement.

*Example:*

```
CHARACTER RECORD(80)
LOOP : RDREC
    READ(5,100) RECORD
    AT END, STOP
    DO I = 1, 80
        IF( RECORD(I) .LT. '0'
+             .OR. RECORD(I) .GT. '9' )THEN
            PRINT *, 'INVALID RECORD'
            CYCLE : RDREC
        ENDIF
    END DO
    WRITE(6,101) RECORD
END LOOP
```

The above example reads in records and verifies that they contain only numeric data. If the record does not, the input operation is tried again. The **CYCLE** statement is within three levels of nesting: the IF, the DO-loop, and the LOOP-END LOOP structure. If a non-numeric character is found, the `CYCLE : RDREC` statement will cause control to be passed to the **READ** statement that follows the **LOOP** statement.

# 9.16 AT END

```
(READ statement)
AT END DO [: block-label ]
    statement(s)
END AT END
```

or

```
(READ statement)
AT END, statement
```

The **AT END** control statement, an extension to FORTRAN 77, is an extension of the **END=** option of the FORTRAN **READ** statement for sequential files. It allows a statement or a block of code following the

**READ** statement to be executed when an end of file condition is encountered during the **READ** and to be by-passed immediately following a **READ** statement. It is not valid to use this control statement with direct-access or memory-to-memory reads. Clearly, it is not valid to use this statement when **END=** is specified in the **READ** statement.

*Example:*

```
READ(7, *) I, X
AT END DO
    PRINT *, 'END-OF-FILE ENCOUNTERED'
    EOFSW = .TRUE.
END AT END
```

If the **READ** statement is executed without encountering end of file, control passes to the statement following the **END AT END** statement. If an end of file condition occurs during the read, the string, `END-OF-FILE ENCOUNTERED` is printed, logical variable `EOFSW` is assigned the value `.TRUE.` and control passes to the statement following the **END AT END** statement.

*Example:*

```
READ(7, *) X
AT END, EOFSW = .TRUE.
```

If an end of file is not encountered by the **READ** statement, control passes to the statement following the **AT END** statement. If an end-of-file condition occurs, variable `EOFSW` is set to `.TRUE.` and control then passes to the statement following the **AT END** statement. Note that the use of the second form of the **AT END** statement requires the use of a comma (,) between the `AT END` word and the executable statement. This is necessary to distinguish the case where the executable statement is an assignment statement. The executable statement may be any statement that is also allowed as the operand of a logical **IF** statement.

# 9.17 Notes on Structured Programming Statements

In addition to the definitions and examples of these constructs, the following points should be noted:

**(i)** Any of the new control statements with their blocks may be used within the block of any other statement. For example, a WHILE-block may contain another WHILE or an IF-THEN-ELSE block. Blocks may be nested in this manner to any level within storage limitations. An important exception to this rule is the REMOTE-block A REMOTE-block may contain other types of blocks (nested to any level); however, another REMOTE-block may not be defined within it. Furthermore, REMOTE-blocks may not be defined within another control structure. The following example is illegal.

*Example:*

```
    * Illegal definition of a REMOTE-block.
          IF( I .EQ. 3 )then
              REMOTE BLOCK A
                  .
                  .
                  .
              END BLOCK
          END IF
```

**(ii)** When nesting blocks, the inner blocks must always be completed with the appropriate block-terminating **END** statement before the outer blocks are terminated. Similarly, when nesting blocks with DO-loops, a DO-loop started within a block must be completed before

the block is completed. A block started within a DO-loop must be terminated before the DO-loop is completed. Indenting the statements of each new block, as shown in the examples, is helpful in avoiding invalid nesting and helps to make the structure of the program visually obvious.

**(iii)** The normal flow of control of the new programming constructs described earlier may be altered with standard FORTRAN control statements. For example, the program may exit from a block using a **GO TO**, **STOP**, **RETURN** or arithmetic **IF** statement. However, a block may not be entered in the middle through use of any control statement such as **GO TO** or the arithmetic **IF**.

Consider the following example.

*Example:*

```
          GO TO 20
    10    IF( X .GT. Y )THEN
              CALL REDUCE( X, Y )
    20        X = X - 1
          ELSE
              CALL SCALE( X )
          END IF
```

This is an example of an illegal attempt to transfer execution into the middle of an IF-block. The statement `X = X - 1` is contained within the IF-block and may only be transferred to from within the block.

*Example:*

```
          IF( X .GT. Y )THEN
    20        CALL REDUCE( X, Y )
              X = X - 1
              IF( X .GT. 0 )GO TO 20
          ELSE
              CALL SCALE( X )
          END IF
```

This last example demonstrates a legal transfer of control within an IF-block. However, we have seen better ways to express the loop with this IF-block.

*Example:*

```
          IF( X .GT. Y )THEN
              LOOP
                  CALL REDUCE( X, Y )
                  X = X - 1
              UNTIL( X .LE. 0 )
          ELSE
              CALL SCALE( X )
          END IF
```

**(iv)** Many control structure statements cannot be branched to using a **GO TO** statement. For a list of these statements, see the section entitled "Classifying Statements" on page 9 in the chapter entitled "FORTRAN Statements"

**(v)** Many control structure statements cannot be the object statement of a logical **IF** statement, or be the last statement of a DO-loop. For a list of these statements, see the section entitled "Classifying Statements" on page 9 in the chapter entitled "FORTRAN Statements"

# 10 Input/Output

## 10.1 Introduction

FORTRAN 77 provides a means of communicating information or *data* between a FORTRAN program and the computing environment. The computing environment may include a number of devices which are capable of the recording, retrieval, display, and input of data. Disk and magnetic tape devices are capable of storing large amounts of data. Other devices such as printers and display terminals can be used to present a visual (i.e., human-readable) representation of the data. Yet other devices such as terminal keyboards and card-readers make possible the entry of new data into the computing environment.

For the purposes of our discussion, data is any information which can be processed by an executing FORTRAN program. Some examples of data are names, addresses, telephone numbers, credit card balances, flight trajectories, bus schedules, athletic records, etc. In computing, such information is usually well-organized in order to make it useful for processing.

To use an example, consider the entries in a telephone book. There are essentially three pieces of data listed for each entry; a name, an address, and a number.

```
Smith J 32 Arthur St--------------------------555-3208
Smith JW 512 King St--------------------------555-9229
Smith Jack 255-113 Queen St N-----------------555-0572
```

Each entry is a *record.* The organization of the book is clear. The name is always listed first, the address second, and the number last. The records are *sorted,* for our convenience, by name (within each city or geographical location). The length of each record is the same. This *fixed length* does sometimes lead to problems since entries which have a long name or address won't fit in a record. The phone company solved this by continuing the information in subsequent records. We might have solved this problem by increasing the length of a record with the disadvantage of wasting a lot of printing space. Alternatively, we could have used a *variable length* record. This solves the problem of wasted space but creates a severe problem when trying to display the records in nice orderly columns. The telephone book itself is a collection of records or a *file.*

We have introduced much of the terminology of data processing: "data", "records", "fixed and variable record sizes", "files", "sorted", etc.

## 10.2 Reading and Writing

FORTRAN provides a mechanism called "reading" for transferring data into the environment of an executing program. The **`READ`** statement is used to do this. Similarly "writing" is the mechanism for transferring data out of an executing program. The **`WRITE`** and **`PRINT`** statements are used to do this. Other statements provide additional functions such as positioning to a certain record in a file, establishing which files are to be processed by the program, or making inquiries about files.

# 10.3 Records

FORTRAN distinguishes between three kinds of records, namely:

1. Formatted
2. Unformatted
3. Endfile

We shall describe each of these in the following sections.

## 10.3.1 Formatted Record

A formatted record consists of characters. The length of a formatted record is determined by the number of characters in it. A formatted record may contain no characters at all and thus has zero length. Formatted records are read or written using *formatted input/output* statements. An excellent example of a file consisting of formatted records is our telephone book example.

## 10.3.2 Unformatted Record

An unformatted record consists of values such as integers, real numbers, complex numbers, etc. It may also consist of characters. Essentially, these values have the same representation in a record as they have in the computer's memory. The length of an unformatted record depends on the amount of storage required to represent these values in the computer's memory. For example, on this computer an integer value is stored using 4 bytes of memory (a byte is a grouping of 8 binary digits). Thus, integer values in unformatted records also require 4 bytes of storage. For example, 3 integer values stored in an unformatted record would require 12 bytes of storage. Unformatted records are read or written using *unformatted input/output* statements.

To illustrate the difference between a formatted and unformatted record consider the following example.

*Example:*

```
          INTEGER NUMBER
          NUMBER=12345
          PRINT 100, NUMBER
100       FORMAT(1X,I5)
          WRITE( UNIT=7 ) NUMBER
```

If you print the variable `NUMBER` on a printer, it requires 5 character positions. If you write it to a file using an unformatted **`WRITE`** statement, it only requires 4 bytes or character positions in the record. Note that a character is conveniently represented in one byte of storage, hence we sometimes use the term "byte" or "character" interchangeably when talking about the size of variables.

## 10.3.3 Endfile Record

An endfile record is a special record that follows all other records in a file. Simply stated, an endfile record occurs at the end of a file. Actually, an endfile record is a conceptual thing. It has no length. When the end of a file is reached (i.e., an attempt to read a record results in the endfile record being read), an "end-of-file" condition exists. There are no more records following the endfile record. There is only one endfile record so it is strictly illegal to attempt to read another record after the endfile record has been read (i.e., when the end-of-file condition exists).

# 10.4 Files

Earlier we described the notion of a file as a collection of records. In FORTRAN, there are two kinds of files:

1. External
2. Internal

## 10.4.1 External Files

External files are files that exist or can be created upon external media such as disks, printers, terminal displays, etc. A file may exist before the execution of a FORTRAN program. It may be brought into existence or "created" during execution. It may also be deleted and therefore not exist after the execution of a FORTRAN program.

All input/output statements may refer to files that exist. In addition, the **INQUIRE**, **OPEN**, **CLOSE**, **WRITE**, **PRINT**, and **ENDFILE** statements may refer to files that do not exist (and in so doing, may very well cause the file to be created).

***Properties of External Files***

***Name*** In FORTRAN, a file may or may not have a name. If it does have a name then, not surprisingly, it is called a *named* file. All files in Open Watcom FORTRAN 77 have names and so it may seem odd to introduce this notion. However, we do since the **INQUIRE** statement lets you find out if a file is named and, if so, what its name is. File naming conventions may differ from one computing system to the next. As well, different FORTRAN 77 compilers may have different file naming conventions.

***Access*** "Access" simply refers to the way in which we can position to and read or write the data in a particular record in a file. There are two ways in which records can be accessed in a file; *sequentially* or *directly.*

Using the *sequential access* method, records may be read or written in order starting with the first record and proceeding to the last record. For example, it would be quite impossible to read or write the tenth record in a file and then read or write the third record. Similarly the eleventh record must be read or written before we can access the twelfth record. If we adopt the convention that each record in a file has a record number then the first record is record number 1, the second is 2, and so on. This numbering convention is important when we look at the other access method which is "direct".

Using the *direct access* method, records may be read or written in any order. It is possible to read or write the tenth record of a file and then the third and then the twelfth and so on. A caveat: a record cannot be read if it has never been written since the file was created. In direct access, the idea of a record number is very important and so by convention, we number them starting at 1 as the first record and proceeding on up. With direct access, if you create a new file and write record number 10 then the file has ten records regardless of the fact that only one has been written. You could, at some later time, write records 1 through 9 (in whatever order you please) and add additional records by writing records with record numbers greater than 10.

Some files have the property of being able to sustain both of these access methods. Some files may only have one of these properties. For example, most line printers cannot be

accessed directly. You have no choice but to write records sequentially. Sometimes a file that was created using the sequential access method may not be accessed using the direct method or vice versa. FORTRAN calls this property of a file the "set of allowed access methods".

***Record Form*** Some files have the property of being able to handle both formatted and unformatted record formats. Some files may only have one of these properties. For example, if you tried to write unformatted records to a line printer, the result might be gibberish. On the other hand a graphics printer may readily accept unformatted records for reproducing graphical images on paper. FORTRAN calls this property of a file the "set of allowed forms".

***Record Length*** Another property of a file is record length. Some files may have restrictions on the length of a record. Some files do not allow records of zero length. Other files, such as printers, may restrict the length of a record to some maximum. FORTRAN calls this property the "set of allowed record lengths".

## 10.4.2 Internal Files

Internal files are special files that reside only in memory. They do not exist before or after the execution of a FORTRAN program, only during the execution of a program. An internal file allows you to treat memory in the computer as if it were one or more records in a file. The file must be a character variable, character array element, character array, or character substring. A record in this file may be a character variable, character array element or character substring.

Another way of looking at this is that an internal file that is either a character variable, character array element or character substring can contain only one record but an internal file that is a character array can contain several records (as many as there are elements in the array).

***Properties of Internal Files***

***Records*** Unless the name of a character array is used, only one record is contained in an internal file. The length of this record is the same as the length of the variable, array element, or substring. If the file is a character array then each element in the array is a record. The order of the records in the file is the same as the order of the elements in the array. The length of a record in this case is the same as the length of the character array elements.

If the number of characters written to a record in an internal file is less than the length of the record then the record is padded with blanks.

***Definition*** A record may be read only if the variable, array element, or substring is defined (i.e., it has been assigned some value). Definition may not only result from an output statement such as **`WRITE`**. It may also be defined through other means; for example, a character assignment statement.

***Position*** For all input/output statements, the file is positioned at the beginning of the first record. Multiple records may be read or written using the "slash" format edit descriptor (see the chapter entitled "Format" on page 225).

***Restrictions*** Only sequential access formatted input and output statements ( **`READ`** and **`WRITE`**) may be used to transfer data to and from records in an internal file.

Although FORTRAN 77 states that list-directed formatted input/output to an internal file is not permitted, Open Watcom FORTRAN 77 allows you to use list-directed formatted input/output statements. This is an extension to the language standard.

*Example:*

```
          WRITE(INTFIL,*) X, Y, Z
```

No other input/output statements ( **OPEN**, **ENDFILE**, **REWIND**, etc.) may be used.

Internal files may be used to convert data from one format to another. The following example illustrates one use of internal files.

*Example:*

```
          CHARACTER*11 INPUT
          PRINT *, 'TYPE IN ''I'' FOLLOWED BY AN INTEGER'
          PRINT *, 'OR TYPE IN ''R'' FOLLOWED BY A REAL'
          READ 100, INPUT
    100   FORMAT( A11 )
          IF( INPUT(1:1) .EQ. 'I' )THEN
              READ( UNIT=INPUT(2:11), FMT='(I10)' ) IVAR
              PRINT *, 'AN INTEGER WAS ENTERED ', IVAR
          ELSE IF( INPUT(1:1) .EQ. 'R' )THEN
              READ( UNIT=INPUT(2:11), FMT='(F10.3)' ) RVAR
              PRINT *, 'A REAL NUMBER WAS ENTERED ', RVAR
          END IF
          END
```

After checking for an `"I"` or `"R"` as the first character of the character variable `INPUT`, the appropriate internal **READ** statement is executed.

# 10.5 Units

Many FORTRAN 77 input/output statements refer to external files using a mechanism called the *unit.* There are many units available to the FORTRAN 77 programmer. Open Watcom FORTRAN 77 numbers these units from 0 to 999; thus the unit number is a non-negative integer less than 1000.

A unit may be associated with a particular file. This association is called *connection.* Any unit may or may not be connected to a file. There are a number of ways in which this connection may be established.

A unit may be *preconnected* to a file before execution of a program begins. The User's Guide describes the mechanism for preconnecting a unit to a file.

Alternatively, a unit may become connected to a file by the execution of an **OPEN** statement.

All input/output statements except **OPEN**, **CLOSE**, and **INQUIRE** must refer to a unit that is connected to a file. Open Watcom FORTRAN 77 automatically establishes a connection of the unit to a file if no connection previously existed. Consider the following example in which unit number 1 is not previously connected to a file.

*Example:*

```
WRITE(1,*) 'Output on unit 1'
END
```

Open Watcom FORTRAN 77 constructs a file name using the specified unit number. The format of the file name is described in the User's Guide since it varies from one computer system to the next.

Connection of a unit to a file does not imply that the file must exist. For example, it could be a new file. When we speak of a unit being connected to a file, we can also say that a file is connected to a unit. Under the rules of FORTRAN, it is illegal to connect the same file to more than one unit *at the same time.* However, a file may be connected to different units at different times. We shall explain how this is possible.

A file may be *disconnected* from a unit by the use of the **CLOSE** statement.

*Example:*

```
CLOSE(UNIT=1)
```

Under certain circumstances, the file may be disconnected from a unit by the use of the **OPEN** statement.

*Example:*

```
OPEN(UNIT=1,FILE='FILE1')
.
.
.
OPEN(UNIT=1,FILE='FILE2')
```

In the above example, the second **OPEN** statement disconnects unit 1 from one file and connects it to a second file. You may think of the second **OPEN** statement as automatically closing the first file and then establishing a connection to the second file.

If a unit has been disconnected from a file through the execution of a **CLOSE** statement, the unit may subsequently be connected to the same file or to a different file. It also follows that a file which has been disconnected from one unit number may be connected to the same unit number or a different unit number. The following example may help to illustrate this last point.

*Example:*

```
OPEN(UNIT=1,FILE='FILE1')
.
.
.
CLOSE(UNIT=1)
OPEN(UNIT=2,FILE='FILE1')
```

Once a file has been disconnected, the only means for referring to the file is by its name in an **OPEN** statement or an **INQUIRE** statement.

# 10.6 Specifiers

All input/output statements contain one or more specifiers. They appear in a list separated by commas. Some of the more common specifiers are those listed below. Not all of them need be used in every input/output statement. You should consult the description of the input/output statement under consideration to discover which specifiers are allowed and what they mean.

| | |
|---|---|
| ***[UNIT =] u*** | the unit specifier |
| ***[FMT =] f*** | the format specifier |
| ***REC = rn*** | the record specifier |
| ***IOSTAT = ios*** | the input/output status specifier |
| ***ERR = s*** | the error specifier |
| ***END = s*** | the end-of-file specifier |

We shall look at these specifiers in more detail.

## 10.6.1 The Unit Specifier

The form of a unit specifier in an input/output statement is:

***[UNIT =] u*** `u` is an *external unit identifier* or an *internal file identifier.*

1. An external unit identifier is a non-negative integer expression or an asterisk `(*)` in which case unit 5 is assumed for an input statement and unit 6 is assumed for an output statement. The unit identifier must not be an asterisk for the **`BACKSPACE`**, **`ENDFILE`** and **`REWIND`** statements.
2. An internal file identifier is the name of a character variable, character array, character array element, or character substring.

If the optional **`UNIT=`** specifier is omitted then the unit specifier must be the first item in the list of specifiers.

## 10.6.2 Format Specifier

The form of a format specifier in an input/output statement is:

***[FMT =] f*** `f` is a *format identifier.* A format identifier is one of the following:

1. A statement label of a **`FORMAT`** statement that appears in the same program unit as the format identifier.
2. An integer variable name that has been assigned the statement label of a **`FORMAT`** statement that appears in the same program unit as the format identifier (see the **`ASSIGN`** statement).
3. An integer array name.
4. A character array name.

5. Any character expression except one involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant (see the **PARAMETER** statement).
6. An asterisk `(*),` indicating *list-directed* formatting.
7. A **NAMELIST** name, indicating *namelist-directed* formatting.

If the optional **FMT=** specifier is omitted then the format specifier must be the second item in the list of specifiers and **UNIT=** must not be specified for the first item in the list.

## 10.6.3 Record Specifier

The form of a record specifier in an input/output statement is:

***REC = rn*** `rn` is an integer expression whose value must be positive. It is the number of the record to be read when a file is connected for *direct access.*

## 10.6.4 Input/Output Status Specifier

The form of an input/output status specifier in an input/output statement is:

***IOSTAT = ios*** `ios` is an integer variable or integer array element. It is defined with zero if no error occurs, a positive integer value if an error occurs, or a negative integer value if an end-of-file occurs.

If an input/output error or end-of-file condition occurs during the execution of an input/output statement and the input/output status specifier is present then execution of the program is not terminated. Input/output errors may result from a violation of the rules of FORTRAN or from a file system error. For example, a negative unit number will result in an error since this is a violation of the rules of FORTRAN. An example of a file system error might be an attempt to create a file on a non-existent file storage device.

Consult the User's Guide for a list of Open Watcom FORTRAN 77 diagnostic messages. An input/output status of `nn` corresponds to the message `IO-nn.` For example, if the status returned is 3 then the error is:

```
IO-03  ENDFILE statement requires sequential access mode
```

## 10.6.5 Error Specifier

The form of an error specifier in an input/output statement is:

***ERR = s*** `s` is a statement label. When an error occurs, execution is transferred to the statement labelled by `s`.

If an input/output error occurs during the execution of an input/output statement and the **ERR=** specifier is present then execution of the program is not terminated.

### 10.6.6 End-of-File Specifier

The form of an end-of-file specifier in an input/output statement is:

***END = s*** `s` is a statement label. When an end-of-file condition occurs, execution is transferred to the statement labelled by `s`.

If an end-of-file condition occurs during the execution of an input/output statement and the **`END=`** specifier is present then execution of the program is not terminated.

## 10.7 Printing of Formatted Records

Printing occurs when *formatted* records are transferred to a device which interprets the first character of the record as a special spacing command. The remaining characters in the record are "printed". Printing can be accomplished by use of either the **`PRINT`** statement or the **`WRITE`** statement. What actually determines whether or not you are "printing" is the device (or file) to which records are transferred.

The first character of the record controls the vertical spacing. This feature is quite often called ASA (American Standards Association) carriage control.

| Character | Vertical Spacing Before Printing |
|---|---|
| Blank | One Line |
| 0 | Two Lines |
| - | Three Lines |
| 1 | To First Line of Next Page |
| + | No Advance |

The "−" control character is an extension to the FORTRAN 77 language that is supported by many "printing" devices.

# 11 Format

## 11.1 Introduction

A *format* specification used in conjunction with formatted I/O provides a means of specifying the way internal data is converted to a character string and vice versa. A format specification can be given in two ways.

1. In a **FORMAT** statement.
2. As values of character expressions or character arrays.

## 11.2 The FORMAT Statement

The form of a **FORMAT** statement is

```
label    FORMAT fs
```

***where:***

***label*** is the statement label used by an I/O statement to identify the **FORMAT** statement.

***fs*** is a format specification which will be described later.

*Example:*

```
          REAL X
          X = 234.43
          PRINT 100, X
100       FORMAT(F10.2)
          END
```

In the previous example, the **PRINT** statement uses the format specification in the **FORMAT** statement whose statement label is 100 to display the value of X.

## 11.3 FORMAT as a Character Expression

Instead of specifying the statement label of a **FORMAT** statement, a character expression can be used. The previous example could be modified as follows and achieve the identical result.

*Example:*

```
REAL X
X = 234.43
PRINT '(F10.2)', X
END
```

When using a character expression to represent a format specification, the format specification can be preceded by blank characters and followed by any character data without affecting the format specification. The following example produces the identical result to the previous example.

*Example:*

```
REAL X
X = 234.43
PRINT '    (F10.2) THIS IS FOR X', X
END
```

If a character array is used to describe the format specification, the format specification is considered to be the concatenation of all the character array elements in the order given by array element ordering described in the chapter entitled "Arrays" on page 159. Note that if a character array element is used, the format specification is considered to be only that array element.

*Example:*

```
REAL X
CHARACTER*5 FMTSPEC(3)
X = 234.43
FMTSPEC(1)='('
FMTSPEC(2)='F10.2'
FMTSPEC(3)=')'
PRINT FMTSPEC, X
END
```

# 11.4 Format Specification

A *format specification* has the following form.

```
( [flist] )
```

***where:***

***flist*** is a list whose items are separated by commas. The forms of the items in `flist` are:

```
[r] ed

ned

[r] fs
```

***ed*** is a repeatable edit descriptor.

***ned*** is a nonrepeatable edit descriptor.

***fs*** is a format specification with a nonempty list `flist`.

***r*** is a positive unsigned integer constant called a *repeat specification.*

The comma separating the items of `flist` can be omitted in the following cases.

1. Between a `P` edit descriptor and an `F`, `E`, `D` or `G` edit descriptor which immediately follows.
2. Before or after a slash edit descriptor.
3. Before or after a colon edit descriptor.

Open Watcom FORTRAN 77 allows the omission of a comma between the items of `flist`. Care should be taken when omitting commas between edit descriptors. For example, the format specification `(I5 2I3)` may appear to be an `I5` edit descriptor followed by two `I3` edit descriptors when in actuality it is interpreted as an `I52` edit descriptor followed by an `I3` edit descriptor.

# 11.5 Repeatable Edit Descriptors

The forms of *repeatable edit descriptors* are:

```
Iw
Iw.m
Fw.d
Ew.d
Ew.dEe
Dw.d
Gw.d
Gw.dEe
Lw
A
Aw
```

As an extension to the FORTRAN 77 language, the following repeatable edit descriptors are also supported.

```
Ew.dDe
Zw
```

***where:***

`I`, `F`, `E`, `D`, `G`, `L`, `A` and `Z` indicate the method of editing.

`w` and `e` are positive unsigned integer constants.

`d` and `m` are unsigned integer constants.

# 11.6 Nonrepeatable Edit Descriptors

The forms of *nonrepeatable edit descriptors* are:

```
'hh...h' (apostrophe)
nHhh...h
Tc
TLc
TRc
nX
/
:
S
SP
SS
kP
BN
BZ
X
```

As an extension to the FORTRAN 77 language, the following nonrepeatable edit descriptors are also supported.

```
$
\
```

***where:***

Apostrophe, `H`, `T`, `TL`, `TR`, `X`, `/`, `:`, `S`, `SP`, `SS`, `P`, `BN`, `BZ`, `\` and `$` indicate the method of editing.

`h` is a character.

`n` and `c` are positive unsigned integer constants.

`k` is an optionally signed integer constant.

Open Watcom FORTRAN 77 allows edit descriptors to be specified using lower case letters.

# 11.7 Editing

Edit descriptors are used to describe the way the editing between internal representation of data and the characters of a record in a file is to take place. When the edit descriptors `I`, `F`, `E`, `D`, `G`, `L`, `A`, `H`, `Z` or apostrophe are processed, they process a sequence of characters called a *field.* On input, the field is the character data read from a record; on output it is the character data written to a record. The number of characters in a field is called the *field width.*

## 11.7.1 Apostrophe Editing

The *apostrophe edit descriptor* has the same form as a character constant and can only be used on output. It causes the characters in the format specification enclosed in quotes to be written. The field width is the number of characters enclosed in quotes.

*Example:*

```
PRINT '(''HI THERE'')'
END
```

In the previous example, the string

```
HI THERE
```

would be the output produced by the **PRINT** statement.

## 11.7.2 H Editing

The `nH` edit descriptor causes the `n` characters following the `H`, including blanks, to be written. Like the apostrophe edit descriptor, it can only appear in a format specification used for output.

*Example:*

```
PRINT '(8HHI THERE)'
END
```

In the previous example, the string

```
HI THERE
```

would be the output produced by the **PRINT** statement.

## 11.7.3 Positional Editing: T, TL, TR and X Editing

The `T`, `TL`, `TR` and `X` edit descriptors specify at which position the next character will be read from or written to the record. In the case of input, this allows data to be read more than once with different edit descriptors. On output, it is possible to overwrite data previously written.

On output it is possible to use positional editing to create a record in which gaps appear. That is, there may be parts of the record where no data has been written. The parts of a record in which no data has been written are filled with blanks. The effect is as if the record was previously initialized to blanks. Note that positioning does not cause any data to be transmitted.

The `Tc` edit descriptor specifies that the next character to be transmitted is to be from the `c`th character position in the record. The `TLc` edit descriptor specifies that the next character to be transmitted is to be from the `c`th position backward from the current position. The `TRc` edit descriptor is identical to the `TLc` edit descriptor except that positioning is forward from the current position. The `nX` edit descriptor behaves identically to the `TRc` edit descriptor; the transmission of the next character is `n` character positions forward from the current position. If `n` is omitted then the transmission of the next character is 1 character position forward from the current position.

*Example:*

```
          PRINT '(''THE NUMBER       IS AN INTEGER'',TL19,
         $      ''12345'')'
          END
```

The output produced is

```
    THE NUMBER 12345 IS AN INTEGER
```

## 11.7.4 Slash Editing

The *slash edit descriptor* indicates the end of data transfer on the current record. On input from a record connected for sequential access, the remaining characters in the record are skipped and the file is positioned to the start of the next record. Note that entire records may be skipped. On output, a new record is created and becomes the last and current record of the file. Note that a record with no characters can be written. If the file is an internal file or a direct access file, the record is filled with blanks.

For a file connected for direct access, the current record number is increased by one and the file is positioned at the beginning of that record.

## 11.7.5 Colon Editing

The *colon edit descriptor* terminates processing of the format specification if there are no more items in the I/O list. If there are items remaining in the I/O list, the colon edit descriptor has no effect.

## 11.7.6 S, SP and SS Editing

The `S`, `SP` and `SS` edit descriptors control optional plus characters in numeric output fields. They only effect the `I`, `F`, `E`, `D` and `G` edit descriptors during output and have no effect on input. The FORTRAN 77 standard specifies that before processing a format specification, the appearance of a plus sign in numeric output fields is optional and is determined by the processor. Open Watcom FORTRAN 77 does not produce plus signs in numeric output fields. When an `SP` edit descriptor is encountered, a plus sign is produced in any subsequent position that optionally contains a plus sign. When as `SS` edit descriptor is encountered, a plus sign is not produced in any subsequent position that optionally contains a plus sign. If an `S` edit descriptor is encountered, the option is returned to the processor.

*Example:*

```
          PRINT '(1H<,I5,SP,I5,SS,I5,1H>)',1,2,3
          END
```

The output produced by the **`PRINT`** statement in the previous example is:

```
    <    1   +2    3>
```

## 11.7.7 P Editing

The form of a `P` edit descriptor is `kP` where `k` is an optionally signed integer constant called the *scale factor*. The value of the scale factor is zero at the beginning of each I/O statement. The scale factor applies to all subsequent `F`, `E`, `D` and `G` edit descriptors until another scale factor is encountered. The scale factor affects editing in the following way.

1. On input with F, E, D and G editing, provided that no exponent exists in the field, the effect is that the represented number equals the internally represented number multiplied by 10**k.
2. On input with F, E, D and G editing, the scale factor has no effect if there is an exponent in the field.
3. On F output editing, the effect is that the represented number equals the internally represented number multiplied by 10**k.
4. On output with E and D editing, the simple real constant (see the chapter entitled "Names, Data Types and Constants" on page 151) part of the data is multiplied by 10**k and the exponent is reduced by k.
5. On output with G editing, the scale factor has no effect unless the magnitude of the datum is outside the range that allows F editing (see the section entitled "G Editing" on page 235). If E editing is required, the scale factor has the same effect as with E output editing.

## 11.7.8 BN and BZ Editing

The BN and BZ edit descriptors are used to describe the interpretation of embedded blanks in numeric input fields. They only effect I, F, E, D and G editing and have no effect during output. When a BN edit descriptor is encountered in a format specification, embedded blanks in subsequent numeric input fields are ignored. However, a field of all blanks has the value of zero. If a BZ edit descriptor is encountered, then all embedded blanks in subsequent numeric input fields are treated as zeroes. At the beginning of each I/O statement, all blanks are treated as zeroes or ignored depending on the value of the **BLANK=** specifier (see the **OPEN** statement) currently in effect for the unit.

## 11.7.9 $ or \ Editing (Extension)

The $ and \ edit descriptors behave identically. The $ and \ edit descriptors are intended for output to an interactive device such as a terminal. They are a Open Watcom FORTRAN 77 extensions. The output record is displayed at the terminal leaving the cursor at the end of the record; the carriage return at the end of the line is suppressed. Its use is intended for prompting for input so that the response can be entered immediately following the prompt.

Depending on the type of terminal, the prompt may be returned as part of the input. An application must be aware of the way a particular terminal behaves. The following example demonstrates this. Note that the format specification in the **FORMAT** statement labelled 20 ignores the first eleven characters of the response since the prompt also appears in the response.

*Example:*

```
          INTEGER AGE
          WRITE(6,FMT=10)
10        FORMAT( 'Enter age: ',$ )
          READ(5,20) AGE
20        FORMAT( 11X, I2 )
          PRINT *,'Your age is ',AGE
          END
```

If the terminal you are using does not return the prompt as part of the response (that is, a read from the terminal only includes characters typed at the terminal), the format specification in the **FORMAT** statement labelled 20 must be changed, as in the following example, to achieve the same result.

*Example:*

```
        INTEGER AGE
        WRITE(6,FMT=10)
10      FORMAT( 'Enter age: ',\/ )
        READ(5,20) AGE
20      FORMAT( I2 )
        PRINT *,'Your age is ',AGE
        END
```

## 11.7.10 Numeric Editing: I, F, E, D and G Edit Descriptors

Numeric edit descriptors are used to specify I/O of integer, real, double precision, complex and double precision complex data. The following rules apply to all numeric edit descriptors.

1. On input, leading blanks are not significant. The interpretation of blanks other than leading blanks is determined by any `BN` or `BZ` edit descriptors in effect and the **`BLANK=`** specifier (see the **`OPEN`** statement). A field of all blanks is always zero. Plus signs are optional.
2. On input, with `F`, `E`, `D` and `G` editing, the decimal location specified in the edit descriptor is overridden by a decimal point appearing in the input field.
3. On output, the plus sign is optional and is determined by the `S`, `SP` and `SS` edit descriptors. A negative quantity is represented by a negative sign. A minus sign is never produced when outputting a value of zero.
4. On output, the representation is always right justified in the field with leading blanks inserted at the beginning of the field if the number of characters in the representation is less than the field width.
5. On output, if the number of characters in the external representation is greater than the field width or an exponent exceeds its specified length using `Ew.dEe`, `Gw.dEe`, `Ew.dDe` or `Gw.dDe` edit descriptors, the entire field is filled with asterisks.

### 11.7.10.1 Integer Editing: Iw and Iw.m Edit Descriptors

The `Iw` and `Iw.m` edit descriptors indicate that the field width of the field to be edited is `w`. The item in the I/O list must be of type integer; on input the I/O list item will be defined by integer data, on output the I/O list item must be defined with an integer datum.

On input, the `Iw.m` edit descriptor is treated identically to the `Iw` edit descriptor. The output field for the `Iw` edit descriptor consists of zero or more leading blanks followed by a minus sign if the value of the I/O list item is negative or an optional plus sign otherwise, followed by the magnitude of the integer datum with no leading zeroes. Note that the integer constant contains at least one digit. On output, the `Iw.m` edit descriptor specifies that at least `m` digits are to be displayed with leading zeroes if necessary. The value of `m` must be less than or equal to the value of `w`. If `m` is zero and the value of the datum is zero, then the output field is filled with blanks.

*Example:*

```
        PRINT '(1H<,I4.4,I5,1H>)',23,2345
```

The output produced by the **`PRINT`** statement in the previous example is the string:

```
<0023 2345>
```

### 11.7.10.2 Floating-point Editing: F, E, D and G Edit Descriptors

The F, E, D and G edit descriptors describe the editing of real, double precision, complex and double precision complex data. The I/O list item corresponding to one of these edit descriptors must be of type real, double precision, complex or double precision complex. On input, the I/O list item will become defined with a datum whose type is the same as that of the I/O list item. On output, the I/O list item must be defined with a datum whose type is that of the I/O list item.

### 11.7.10.3 F Editing

An F edit descriptor has the form Fw.d where w is the field width and d is the number of digits in the fractional part. The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost d digits with leading zeroes assumed if necessary, are interpreted as the fractional part of the value represented. An exponent of one of the following forms may follow.

1. A signed integer constant.
2. An E or D followed by an optionally signed integer constant.

Consider the following example, where the decimal point is omitted. The formula used in the evaluation is:

```
                          -d       (exponent subfield)
(integer subfield) x 10    x 10
```

If the specification is F10.8 and the input quantity is 31415E+5 then the following conversion takes place.

```
                      -8      5
       00031415 x 10    x 10

                      5
=    .00031415 x 10

=    31.415
```

In other words, the decimal point is assumed to lie to the left of the 8 digits (padded with zeroes on the left) forming the fractional part of the input value.

The output field produced by an F edit descriptor consists of blanks if necessary followed by a minus sign if the item in the I/O list is negative or an optional plus sign otherwise, followed by a string of digits containing a decimal point which represents the magnitude of the I/O list item. The string representing the magnitude of the I/O list item is modified according to the scale factor and is rounded to d fractional digits. An optional leading zero is produced only if the magnitude of the I/O list item is less than one. Note that a leading zero is required if there would otherwise be no digits in the output field.

*Example:*

```
PRINT '(1H<,F8.4,1H>)', 234.43
```

The output produced by the **PRINT** statement in the previous example is the string:

```
<234.4300>
```

## 11.7.10.4 E and D Editing

The Ew.d, Dw.d and Ew.dEe edit descriptors indicate that the field width is w, the fractional part contains d digits unless a scale factor greater than one is in effect, and the exponent consists of e digits. The e has no effect on input.

The form of the input field and processing of it is the same as that for F editing. The form of the output field is:

```
[+] [0] . x  x  ... x  exp
[-]      1  2      d
```

***where:***

***p*** indicates a plus or minus sign.

***x's*** are the d most significant digits of the value after rounding.

***exp*** is a decimal exponent.

The form of the exponent is as follows.

1. When using the Ew.d edit descriptor, the form of the exponent is

```
E+nn
 or       if |exp| <= 99
E-nn
```

and

```
+nnn
 or       if 99 < |exp| <= 999
-nnn
```

2. When using the Ew.dEe edit descriptor, the form of the exponent is

```
E+n ... n   where |exp| <= (10**e)-1
 - 1     e
```

3. When using the Dw.d edit descriptor, the form of the exponent is

```
D+nn
 or       if |exp| <= 99
D-nn
```

and

```
+nnn
 or       if 99 < |exp| <= 999
-nnn
```

Note that a sign in the exponent is always present. If the exponent is 0, a plus sign is used. The forms Ew.d and Dw.d are not to be used if |exp| > 999.

*Example:*

```
      PRINT '(1H<,E10.4,1H>,1H<,E9.4,1H>,1H<,E12.4E3,1H>)',
     $      .5, .5, .5
      END
```

The output from the **PRINT** statement in the previous example is the string:

```
<0.5000E+00><.5000E+00>< 0.5000E+000>
```

The scale factor `k` in a `P` edit descriptor controls decimal normalization as follows:

1. If `-d < k <= 0`, then the output field contains `|k|` leading zeroes and `d-|k|` significant digits after the decimal point.
2. If `0 < k < d+2`, the output field contains exactly `k` significant digits to the left of the decimal point and `d-k+1` significant digits to the right of the decimal point.
3. Other values of `k` are not permitted.

The `Ew.dDe` edit descriptor behaves in the same way as the `Ew.dEe` edit descriptor on input; on output the only difference is that the letter `D` is used to mark the exponent instead of the letter `E`.

## 11.7.10.5 G Editing

The `Gw.d` and `Gw.dEe` edit descriptors indicate that the field width is `w`, the fractional part contains `d` digits unless a scale factor greater than one is in effect, and the exponent consists of `e` digits.

`G` input editing is the same as `F` input editing.

The representation on `G` output editing depends on the magnitude of the datum being edited. Let `M` be the magnitude of the datum being edited. Then `G` output editing behaves as follows.

1. If `M < 0.1` or `M >= 10**d`, `Gw.d` output editing is equivalent to `kPEw.d` output editing and `Gw.dEe` output editing is equivalent to `kPEw.dEe` output editing where `k` is the scaling factor currently in effect.
2. If `0.1 <= M < 10**d`, the scale factor has no effect and the value of `M` determines the editing as shown in the following table.

| Magnitude of Datum | Equivalent Edit Descriptor |
|---|---|
| 0.1<=M<1 | F<w-n>.d followed by n blanks |
| 1<=M<10 | F<w-n>.<d-1> followed by n blanks |
| . | . |
| . | . |
| . | . |
| 10**(d-2)<=M<10**(d-1) | F<w-n>.1 followed by n blanks |
| 10**(d-1)<=M<10**d | F<w-n>.0 followed by n blanks |

***where:***

***<w-n>*** stands for the integer represented by evaluating `w-n`.

***<d-1>*** stands for the integer represented by evaluating `d-1`.

***n*** is 4 for `Gw.d` editing and `e+2` for `Gw.dEe` editing.

*Example:*

```
PRINT '(1H<,G12.6,1H>,1H<,G12.4E4,1H>)', .5, .5
END
```

The output from the **PRINT** statement in the previous example is the string:

```
<0.500000    ><0.5000      >
```

### 11.7.10.6 Complex Editing

Since a complex datum consists of a pair of real or double precision data, the editing of a complex datum is specified by two successive pairs of `F`, `E`, `D` or `G` edit descriptors. The two descriptors may be different and may be separated by any number of non-repeatable edit descriptors. Double precision complex editing is identical to complex editing.

## 11.7.11 L Edit Descriptor

The `Lw` edit descriptor is used for I/O list items of type logical. The field width is `w`.

On input the I/O list item will become defined with a datum of type logical. The input field consists of optional blanks, followed by an optional decimal point followed by a `T` or `F` for true or false respectively. The `T` and `F` may be followed by additional characters in the field. Open Watcom FORTRAN 77 allows `t` and `f` in addition to `T` and `F` on input.

On output, the I/O list item must be defined with a datum of type logical. The output field consists of `w-1` blanks followed by a `T` for true or `F` for false.

*Example:*

```
PRINT '(1H<,L3,L5,1H>)',.TRUE.,.FALSE.
```

The output produced by the **PRINT** statement in the previous example is the string:

```
<  T    F>
```

## 11.7.12 A Edit Descriptor

The `A[w]` edit descriptor is used for I/O list items of type character. On input, the I/O list item becomes defined with character data. On output, the I/O list item must be defined with character data. If `w` is specified in the edit descriptor, the field width is `w` otherwise the field width is the number of characters in the I/O list item.

Open Watcom FORTRAN 77 also permits I/O list items of non-character data types. On input, the I/O list item becomes defined with the binary representation of the character data. On output, the I/O list item is assumed to be defined with character data.

If `len` is the length of the I/O list item and `w` is specified in `A` input editing so that `w` is greater than `len`, the rightmost `len` characters of the input field will be taken. If `w` is less than `len`, then the `w` characters in the input field will be taken and padded with `len-w` blanks.

If w is specified in A output editing so that w is greater than len, then the output field will consist of w-len blanks followed by the len characters of the I/O list item. If w is less than or equal to len, the output field will consist of the first w characters of the I/O list item.

*Example:*

```
PRINT '(1H<,A5,A8,1H>)','ABCDEFG','123'
```

The output produced by the **PRINT** statement in the previous example is the string:

```
<ABCDE     123>
```

## 11.7.13 Z Editing (Extension)

The Zw edit descriptor is used to display the hexadecimal representation of data or read hexadecimal data. It is a Open Watcom FORTRAN 77 extension. The Zw edit descriptor can be used for I/O list items of any type. The field width is w.

On output, w must be greater than or equal to twice the size (in bytes) of the I/O list item since each byte is represented by two hexadecimal digits. For example, real data requires four bytes. Hence, w must be at least eight.

*Example:*

```
PRINT '(1H<,Z8,1H>)', 256
```

The output produced by the **PRINT** statement in the previous example is the string:

```
<00000100>
```

If w is greater then the number of hexadecimal digits required to represent the data, the leftmost print positions of the output field are filled with blanks.

*Example:*

```
PRINT '(1H<,Z10,1H>)','ABCD'
```

The output produced by the **PRINT** statement in the previous example is the string

```
<  C1C2C3C4>
```

if the EBCDIC character set is being used or

```
<  41424344>
```

if the ASCII character set is being used.

On input, if w is greater than twice the size (in bytes) of the I/O list item, the leftmost characters are truncated from the input field. For example, if the input field contains the string

```
91A2C3D4
```

and is read into a character variable whose length is two, the character would contain the hexadecimal data C3D4. If w is less than twice the size (in bytes) of the I/O item, the I/O item is padded to the left with hexadecimal zeroes. For example, if the input field contains the string

```
81C1
```

and is read into a character variable whose length is four, the character would contain the hexadecimal data `000081C1.`

# 11.8 Format-Directed Input/Output

*Format-directed input/output* (I/O) is formatted input or output controlled by a format specification. The action taken during formatted input or output depends on the next edit descriptor in the format specification and the next item in the input/output list if one exists.

A format specification is processed from left to right. An edit descriptor or a format specification with a repeat specification of `r` is processed as a list of `r` edit descriptors or format specifications. A repeat specification of one is equivalent to no repeat specification.

For each repeatable edit descriptor in the format specification, there corresponds one item in the I/O list except an I/O list item of type complex where two repeatable floating-point edit descriptors are required. Non-repeatable edit descriptors do not correspond to any I/O list item; they communicate information directly with the record. Whenever a repeatable edit descriptor is encountered in a format specification, there must be a corresponding item in the I/O list. The edited information is transmitted appropriately between the item and the record.

Format processing is terminated when any of the following conditions occur.

1. When an edit descriptor has no corresponding item in the I/O list.
2. When a colon edit descriptor is encountered and there are no more items in the I/O list.
3. When the right parenthesis is encountered and there are no more items in the I/O list.

If the right parenthesis of the complete format specification is encountered and the I/O list has not been exhausted, the file is positioned at the next record and format processing resumes at the start of the format specification terminated by the last preceding right parenthesis. If there is no such right parenthesis, format processing resumes at the start of the complete format specification. The part of the format specification that is reused must contain at least one repeatable edit descriptor. If format processing resumes at a left parenthesis preceded by a repeat specification, the repeat specification is also reused. The scale factor, sign control edit descriptors and blank control edit descriptors are not affected when part of a format specification is reused.

# 11.9 List-Directed Formatting

*List-directed formatting* is input/output without a format specification.

*Example:*

```
READ( un, * ) X, Y, Z
READ( UNIT=un, FMT=* ) X, Y, Z
READ *, X, Y, Z
WRITE( un, * ) X, Y, Z
WRITE( UNIT=un, FMT=* ) X, Y, Z
PRINT *, X, Y, Z
```

In the previous example, an asterisk instead of a format specification indicates list-directed formatting.

Omitting the asterisk and format specification also indicates list-directed formatting.

*Example:*

```
READ, X, Y, Z
PRINT, X, Y, Z
```

Records used during list-directed formatting are called list-directed records. A list-directed record is a sequence of values and value separators. Any sequence of blanks is treated as a single blank except when it appears in a character constant. The end of a record has the same effect as a blank character.

A *value* is one of the following:

1. A constant.
2. A null value.
3. `r*c`
4. `r*`

***where:***

***r*** is an unsigned, nonzero integer constant.

***c*** is a constant.

The `r*c` form is equivalent to `r` successive occurrences of `c`. The `r*` form is equivalent to `r` successive occurrences of the null value. In these two forms, blanks are permitted only where they are allowed in the constant `c`.

A *value separator* is one of the following:

1. A comma preceded and followed by any number of blanks.
2. A slash preceded and followed by any number of blanks. A slash as a value separator terminates the execution of the input statement. The definition status of the remaining input items in the input list remains the same as it was prior to the input statement.
3. Any number of blanks between two values.

A null value is specified by having no character between successive value separators, no characters preceding the first value separator in a record or the `r*` form. It has no effect on the current value of the input item. Note that the end of record following a value separator does not specify a null value.

## 11.9.1 List-Directed Input

The input forms acceptable to format specifications for a given type are also acceptable for list-directed formatting with certain exceptions.

1. Blanks are never used as zeroes and blanks embedded in constants are not allowed except in character constants.

2. An input item of type real or double precision must have an input field suitable for F editing except that no fractional digits are assumed unless a decimal point is present in the field. Such a field will be called a *numeric input field.*

3. An input item of type complex or double precision complex must consist of a left parenthesis followed by two numeric input fields separated by a comma and followed by a right parenthesis. The numeric input fields may be preceded or followed by blanks. The end of record can only appear between the first numeric field and the comma or between the comma and the second numeric field. Note that a null value must not be used as the real or imaginary part but may represent the entire complex constant.

4. An input item of type logical must not include either a slash or a comma among the optional characters allowed in L editing.

5. An input item of type character consists of a non-empty string of characters enclosed in apostrophes. Apostrophes in character constants are represented by two consecutive apostrophes without a blank or end of record separating them. Character constants may span records. If this is the case, the end of record does cause a blanks to be inserted into the character constant. Note that a comma or slash in a character constant is not a value separator. A character input field is assigned to a character input item as though it were a character assignment.

### 11.9.2 List-Directed Output

The form of the output field produced by list-directed output is similar to the form required by list-directed input. The output of a character constant does not include the enclosing quotes and an apostrophe in a character constant is output as a single apostrophe. The values are separated by one or more blanks. When printed, each record will start with a blank if the file is a carriage-control oriented file. For example, the source listing file produced by Open Watcom FORTRAN 77 is such a file.

## 11.10 Namelist-Directed Formatting (Extension)

The **READ**, **WRITE**, and **PRINT** statements may be used to transmit data between a file and the variables specified in a **NAMELIST** statement.

*Example:*

```
      CHARACTER*20 NAME
      CHARACTER*20 STREET
      CHARACTER*15 CITY
      CHARACTER*20 STATE
      CHARACTER*20 COUNTRY
      CHARACTER*10 ZIP_CODE
      INTEGER AGE
      INTEGER MARKS(10)
      NAMELIST /nl/ NAME, STREET, CITY, STATE,
     +              COUNTRY, ZIP_CODE, AGE, MARKS
           .
           .
           .
      READ( un, nl )
      READ( UNIT=un, FMT=nl )
      READ nl
      WRITE( un, nl )
      WRITE( UNIT=un, FMT=nl )
      PRINT nl
```

## 11.10.1 Namelist-Directed Input (Extension)

The input data must be in a special format. The first character in each record must be blank. The second character in the first record of a group of data records must be an ampersand (&) or dollar sign ($) immediately followed by the **NAMELIST** name. The **NAMELIST** name must be followed by a blank and must not contain any imbedded blanks. This name is followed by data items, optionally separated by commas. The end of a data group is signaled by the character "&" or "$", optionally followed by the string "END". If the "&" character was used to start the group, then it must be used to end the group. If the "$" character was used to start the group, then it must be used to end the group.

```
12345678901234567890...
 &NL
 item1, item2, item3,
 item4, item5, ...
```

The form of the data items in an input record is:

***Name = Constant***

The name may be a variable name, an array element name, or a character substring name. The constant may be integer, real, complex, logical or character. Logical constants may be in the form "T" or ".TRUE" and "F" or ".FALSE". Character constants must be contained within apostrophes. Subscripts and substring indices must be of integer type.

***ArrayName = Set of Constants***

The set of constants consists of constants of the type integer, real, complex, logical or character. The constants are separated by commas. The number of constants must be less than or equal to the number of elements in the array. Successive occurrences of the same constant may be represented in the form `r*constant`, where `r` is a non-zero integer constant specifying the number of times the constant is to occur. Consecutive commas within a list indicate that the values of the array elements remain unchanged.

The variable and array names specified in the input file must appear in the **NAMELIST** list, but the order is not important. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the **NAMELIST** list. The list can contain names of items in **COMMON** but must not contain dummy argument names.

Each data record must begin with a blank followed by a complete variable or array name or constant. Embedded blanks are not permitted in names or constants. Trailing blanks after integers and exponents are treated as zeros.

*Example:*

```
&PERSON
   NAME = 'John Doe'
   STREET = '22 Main St.' CITY = 'Smallville'
   STATE = 'Texas'        COUNTRY = 'U.S.A.'
   ZIP_CODE = '78910-1203'
   MARKS = 73, 82, 3*78, 89, 2*93, 91, 88
   AGE = 23
&END
```

The input forms acceptable to format specifications for a given type are also acceptable for namelist-directed formatting with certain exceptions.

1. Blanks are never used as zeroes and blanks embedded in constants are not allowed except in character constants.

2. An input item of type real or double precision must have an input field suitable for F editing except that no fractional digits are assumed unless a decimal point is present in the field. Such a field will be called a *numeric input field.*

3. An input item of type complex or double precision complex must consist of a left parenthesis followed by two numeric input fields separated by a comma and followed by a right parenthesis. The numeric input fields may be preceded or followed by blanks. The end of record can only appear between the first numeric field and the comma or between the comma and the second numeric field. Note that a null value must not be used as the real or imaginary part but may represent the entire complex constant.

4. An input item of type logical must not include either a slash or a comma among the optional characters allowed in L editing.

5. An input item of type character consists of a non-empty string of characters enclosed in apostrophes. Apostrophes in character constants are represented by two consecutive apostrophes without a blank or end of record separating them. Character constants may span records. If this is the case, the end of record does cause a blank to be inserted into the character constant. Note that a comma or slash in a character constant is not a value separator. A character input field is assigned to a character input item as though it were a character assignment.

## 11.10.2 Namelist-Directed Output

The form of the data items in an output record is suitable for input using a namelist-directed **`READ`** statement.

1. Output records are written using the ampersand character (&), not the dollar sign ($), although the dollar sign is accepted as an alternative during input. That is, the output data is preceded by "&name" and is followed by "&END".

2. All variable and array names specified in the **`NAMELIST`** list and their values are written out, each according to its type.

3. Character data is enclosed in apostrophes.

4. The fields for the data are made large enough to contain all the significant digits.

5. The values of a complete array are written out in columns.

# 12 Functions and Subroutines

## 12.1 Introduction

Functions and subroutines are procedures that fall into one of the following categories.

1. Statement functions
2. Intrinsic functions
3. External functions
4. Subroutines

First let us introduce some terminology.

A *program unit* is a collection of Open Watcom FORTRAN 77 statements and comments that can be either a main program or a subprogram.

A *main program* identifies the program unit where execution is to begin. A main program is a program unit which has as its first statement a **PROGRAM** statement or one which does not have a **PROGRAM**, **FUNCTION**, **SUBROUTINE** or **BLOCK DATA** statement as its first statement. Complete execution of the main program implies the complete execution of the program. Each executable program can contain only one main program.

A *subprogram* is a program unit that either has a **FUNCTION**, **SUBROUTINE** or **BLOCK DATA** statement as its first statement. This chapter will only deal with subprograms that have a **FUNCTION** or **SUBROUTINE** statement as its first statement.

## 12.2 Statement Functions

A statement function is a procedure defined by a single statement. Its definition must follow all specification statements and precede the first executable statement. The statement defining a statement function is not an executable statement.

A *statement function* has the following form.

```
sf ( [d [,d] ...] ) = e
```

***where:***

***sf*** is the name of the statement function.

***d*** is a statement function dummy argument.

***e*** is an expression.

The expression `e` and the statement function name `sf` must conform according to the rules of assignment as described in the chapter entitled "Assignment Statements" on page 187.

The statement function dummy arguments are variable names and are used to indicate the order, number and type of the arguments of the statement function. A dummy argument name of a statement function must only appear once in the dummy argument list of the statement function. Its scope is the statement defining the statement function. That is, it becomes defined when the statement function is referenced and undefined when execution of the statement function is completed. A name that is a statement function dummy argument can also be the name of a variable, a common block, the dummy argument of another statement function or appear in the dummy argument list of a **`FUNCTION`**, **`SUBROUTINE`** or **`ENTRY`** statement. It cannot be used in any other context.

The expression `e` can contain any of the following as operands.

1. A constant.
2. A symbolic constant.
3. A variable reference. This can be a reference to a statement function dummy argument or to a variable that appears within the same program unit which defines the statement function. If the statement function dummy argument has the same name as a variable in the same program unit, the statement function dummy argument is used. The variable reference can also be a dummy argument that appears in the dummy argument list of a **`FUNCTION`** or **`SUBROUTINE`** statement. If it is a dummy argument that has appeared in the dummy argument list of an **`ENTRY`** statement, then the **`ENTRY`** statement must have previously appeared.
4. An array element reference.
5. An intrinsic function reference.
6. A reference to a statement function whose defining statement has previously appeared.
7. An external function reference.
8. A dummy procedure reference.
9. An expression enclosed in parentheses which adheres to the rules specified for the expression `e.`

## 12.2.1 Referencing a Statement Function

A statement function is referenced by its use in an expression. The process of executing a statement function involves the following steps.

1. The expressions that form the actual arguments to the statement function are evaluated.
2. The dummy arguments of the statement function are associated with the actual arguments.
3. The expression `e` is evaluated.
4. The value of the result is converted to the type of the statement function according to the rules of assignment and is available to the expression that contained the reference to the statement function.

The actual arguments must agree in order, number and type with the corresponding dummy arguments.

*Example:*

```
      SUBROUTINE CALC( U, V )
      REAL POLY, X, Y, U, V, Z, CONST
*
* Define a Statement Function.
*
      POLY(X,Y) = X**2 + Y**2 + 2.0*X*Y + CONST
*
* Invoke the Statement Function.
*
      CONST = 23.5
      Z = POLY( U, V )
      PRINT *, Z
      END
```

In the previous example, note that after the execution of the statement function, the values of `X` and `Y` are not equal to the value of `U` and `V` respectively; they are undefined.

## 12.2.2 Statement Function Restrictions

1. A statement function is local to the program unit in which it is defined. Thus, a statement function name is not allowed to appear in an **`EXTERNAL`** statement and cannot be passed to another procedure as an actual argument. The following example illegally attempts to pass the statement function `F` to the subroutine `SAM.`

   *Example:*

   ```
       * Illegally passing a statement function
       * to a subroutine.
             PROGRAM MAIN
             F(X) = X
                  .
                  .
                  .
             CALL SAM( F )
                  .
                  .
                  .
             END
   ```

2. If a statement function `F` contains a reference to another statement function `G,` then the statement defining `G` must have previously appeared. In the following example, the expression defining the statement function `F` illegally references a statement function `G` whose defining statement follows the statement defining `F.`

*Example:*

```
    * Illegal order of statement functions.
                .
                .
                .
          F(X) = X + G(X)
          G(X) = X + 2
                .
                .
                .
```

3. The statement function name must not be the same name of any other entity in the program unit except possibly the name of a common block.

4. If a dummy argument of a statement function is of type CHARACTER, then its length specification must be an integer constant expression. The following is illegal.

   *Example:*

```
          SUBROUTINE SAM( X )
          CHARACTER*(*) X
    * Illegal - CHARACTER*(*) dummy argument not
    *           allowed in statement function.
          F(X) = X
          PRINT *, F('ABC')
          END
```

5. An actual argument to a statement function can be any expression, except character expressions involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant.

# 12.3 Intrinsic Functions

An *intrinsic function* is a function that is provided by Open Watcom FORTRAN 77.

## 12.3.1 Specific Names and Generic Names of Intrinsic Functions

All intrinsic functions can be referenced by using the *generic name* or the *specific name* of the intrinsic function. The specific name uniquely identifies the function to be performed. The type of the result is predefined thus its name need not appear in a type statement. For example, CLOG is a specific name of the generic LOG function and computes the natural logarithm of a complex number. The type of the result is also COMPLEX.

When the generic name is used, a specific name is selected based on the data type of the actual argument. For example, the generic name of the natural logarithm intrinsic function is LOG. To compute the natural logarithm of REAL, DOUBLE PRECISION, COMPLEX or DOUBLE PRECISION COMPLEX data, the generic name LOG can be used. Generic names simplify the use of intrinsic functions because the same name can be used with more than one type of argument.

*Notes:*

1. It is also possible to pass intrinsic functions to subprograms. When doing so, only the specific name of the intrinsic function can be used as an actual argument. The specific name must have appeared in an **INTRINSIC** statement.

2. If an intrinsic function has more than one argument, each argument must be of the same type.

3. The generic and specific name of an intrinsic function is the same for some intrinsic functions. For example, the specific name of the intrinsic function which computes the sine of an argument whose type is REAL is called SIN which is also the generic name of the sine function.

The following sections present all generic and specific names of intrinsic functions and describe how they are used. The following is a guide to interpreting the information presented.

Data types are represented by letter codes.

1. CHARACTER is represented by CH.
2. LOGICAL is represented by L.
3. INTEGER is represented by I.
4. INTEGER*1 is represented by I1.
5. INTEGER*2 is represented by I2.
6. REAL (REAL*4) is represented by R.
7. DOUBLE PRECISION (REAL*8) is represented by D.
8. Single precision COMPLEX (COMPLEX*8) is represented by C.
9. Double precision COMPLEX (COMPLEX*16) is represented by Z.

The "Definition" description gives the mathematical definition of the function performed by the intrinsic function. There are two fields for each intrinsic function. The "Name" field lists the specific and generic names of the intrinsic functions. When the name of an intrinsic function is a generic name, it is indicated by the word "generic" in parentheses; all other names are specific names. The "Usage" field describes how the intrinsic functions are used. "R ←ATAN2(R,R)" is a typical entry in this field. The name of the intrinsic function always follows the " ←". In this example the name of the intrinsic function is ATAN2. The data type of the arguments to the intrinsic function are enclosed in parentheses, are separated by commas, and always follow the name of the intrinsic function. In this case, ATAN2 requires two arguments both of type REAL. The type of the result of the intrinsic function is indicated by the type preceding the " ←". In this case, the result of ATAN2 is of type REAL.

Open Watcom FORTRAN 77 extensions to the FORTRAN 77 language are flagged by a dagger (†).

## 12.3.2 Type Conversion: Conversion to integer

***Definition:*** `int(a)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***INT (generic)*** | I ←INT(I), I ←INT(R), I ←INT(D), I ←INT(C), I ←INT(Z) † |
| ***INT*** | I ←INT(R) |
| ***HFIX*** | I2 ←HFIX(R) † |
| ***IFIX*** | I ←IFIX(R) |
| ***IDINT*** | I ←IDINT(D) |

***Notes:*** The value of int(X) is X if X is of type INTEGER. If X is of type REAL or DOUBLE PRECISION, then int(X) is 0 if |X|<1 and the integer whose magnitude is the largest integer that does not exceed the magnitude of X and has the same sign of X if |X| > 1. If X is of type COMPLEX or COMPLEX*16, int(X) is int(real part of X).

† is an extension to FORTRAN 77.

## 12.3.3 Type Conversion: Conversion to real

| ***Name:*** | ***Usage:*** |
|---|---|
| ***REAL (generic)*** | R ←REAL(I), R ←REAL(R), R ←REAL(D), R ←REAL(C), R ←REAL(Z) † |
| ***REAL*** | R ←REAL(I) |
| ***FLOAT*** | R ←FLOAT(I) |
| ***SNGL*** | R ←SNGL(D) |

***Notes:*** For X of type COMPLEX, REAL(X) is the real part of X. For X of type COMPLEX*16, REAL(X) is the single precision representation of the real part of X.

† is an extension to FORTRAN 77.

## 12.3.4 Type Conversion: Conversion to double precision

| *Name:* | *Usage:* |
|---|---|
| ***DBLE (generic)*** | D ←DBLE(I), D ←DBLE(R), D ←DBLE(D), D ←DBLE(C), D ←DBLE(Z) † |
| ***DREAL*** | D ←DREAL(Z) † |
| ***DFLOAT*** | D ←DFLOAT(I) † |
| ***Notes:*** | For X of type COMPLEX, DBLE(X) is the double precision representation of the real part of X. For X of type COMPLEX*16, DBLE(X) is the real part of X. |
| | † is an extension to FORTRAN 77. |

## 12.3.5 Type Conversion: Conversion to complex

| *Name:* | *Usage:* |
|---|---|
| ***CMPLX (generic)*** | C ←CMPLX(I), C ←CMPLX(I,I), C ←CMPLX(R), C ←CMPLX(R,R), C ←CMPLX(D), C ←CMPLX(D,D), C ←CMPLX(C), C ←CMPLX(Z) † |
| ***Notes:*** | If X is of type COMPLEX, then CMPLX(X) is X. If X is of type COMPLEX*16, then CMPLX(X) is a complex number whose real part is REAL(real part of X) and imaginary part is REAL(imaginary part of X). |
| | If X is not of type COMPLEX, then CMPLX(X) is the complex number whose real part is REAL(X) and imaginary part is REAL(0). CMPLX(X,Y) is the complex number whose real part is REAL(X) and whose imaginary part is REAL(Y) for X,Y not of type COMPLEX. |
| | † is an extension to FORTRAN 77. |

## 12.3.6 Type Conversion: Conversion to double complex

| *Name:* | *Usage:* |
|---|---|
| ***DCMPLX (generic)*** † | Z ←DCMPLX(I), Z ←DCMPLX(I,I), Z ←DCMPLX(R), Z ←DCMPLX(R,R), Z ←DCMPLX(D), Z ←DCMPLX(D,D), Z ←DCMPLX(C), Z ←DCMPLX(Z) |
| ***Notes:*** | If X is of type COMPLEX*16, then DCMPLX(X) is X. If X is of type COMPLEX, then DCMPLX(X) is a COMPLEX*16 number whose real part is DBLE(real part of X) and imaginary part is DBLE(imaginary part of X). |
| | If X is not of type COMPLEX*16, then DCMPLX(X) is the COMPLEX*16 number whose real part is DBLE(X) and imaginary part is DBLE(0). DCMPLX(X,Y) is the COMPLEX*16 number whose real part is DBLE(X) and whose imaginary part is DBLE(Y) for X,Y not of type COMPLEX. |
| | † is an extension to FORTRAN 77. |

## 12.3.7 Type Conversion: Character conversion to integer

| *Name:* | *Usage:* |
|---|---|
| *ICHAR* | I ←ICHAR(CH) |

*Notes:* ICHAR returns an integer which describes the position of the character in the processor collating sequence. The first character in the collating sequence is in position 0 and the last character of the collating sequence is in position n−1 where n is the number of characters in the collating sequence. The value of ICHAR(X) for X a character of length one is such that 0 <= ICHAR(X) <= n−1. For any characters X and Y, the following holds true.

1. X .LT. Y if and only if ICHAR(X) .LT. ICHAR(Y)
2. X .EQ. Y if and only if ICHAR(X) .EQ. ICHAR(Y)

CHAR is the inverse of ICHAR.

## 12.3.8 Type Conversion: Conversion to character

| *Name:* | *Usage:* |
|---|---|
| *CHAR* | CH ←CHAR(I) |

*Notes:* CHAR returns the character in the i'th position of the processor collating sequence. The first character in the collating sequence is in position 0 and the last character of the collating sequence is in position n−1 where n is the number of characters in the collating sequence. The value of CHAR(I) is of type CHARACTER of length one. The argument I must be in the range 0 <= I <= n−1.

ICHAR is the inverse of CHAR.

## 12.3.9 Truncation

*Definition:* `int(a)`

| *Name:* | *Usage:* |
|---|---|
| *AINT (generic)* | R ←AINT(R), D ←AINT(D) |
| *AINT* | R ←AINT(R) |
| *DINT* | D ←DINT(D) |

*Notes:* The value of int(X) is X if X is of type INTEGER. If X is of type REAL or DOUBLE PRECISION, then int(X) is 0 if |X|<1 and the integer whose magnitude is the largest integer that does not exceed the magnitude of X and has the same sign of X if |X| > 1. If X is of type COMPLEX or COMPLEX*16, int(X) is int(real part of X).

## 12.3.10 Nearest Whole Number

| | |
|---|---|
| ***Definition:*** | `int(a+.5)` if a>=0; `int(a-.5)` if a<0 |
| ***Name:*** | ***Usage:*** |
| ***ANINT (generic)*** | R ←ANINT(R), D ←ANINT(D) |
| ***ANINT*** | R ←ANINT(R) |
| ***DNINT*** | D ←DNINT(D) |

## 12.3.11 Nearest Integer

| | |
|---|---|
| ***Definition:*** | `int(a+.5)` if a>=0; `int(a-.5)` if a<0 |
| ***Name:*** | ***Usage:*** |
| ***NINT (generic)*** | I ←NINT(R), I ←NINT(D) |
| ***NINT*** | I ←NINT(R) |
| ***IDNINT*** | I ←IDNINT(D) |

## 12.3.12 Absolute Value

| | |
|---|---|
| ***Definition:*** | `(ar**2+ai**2)**1/2` if a is complex; `\|a\|` otherwise |
| ***Name:*** | ***Usage:*** |
| ***ABS (generic)*** | I ←ABS(I), I1 ←ABS(I1) †, I2 ←ABS(I2) †, R ←ABS(R), D ←ABS(D), R ←ABS(C), D ←ABS(Z) † |
| ***IABS*** | I ←IABS(I) |
| ***I1ABS*** | I1 ←I1ABS(I1) † |
| ***I2ABS*** | I2 ←I2ABS(I2) † |
| ***ABS*** | R ←ABS(R) |
| ***DABS*** | D ←DABS(D) |
| ***CABS*** | R ←CABS(C) |
| ***CDABS †*** | D ←CDABS(Z) |
| ***Notes:*** | A complex number is an ordered pair of real numbers, `(ar,ai)` where `ar` is the real part and `ai` is the imaginary part of the complex number.<br><br>† is an extension to FORTRAN 77. |

## 12.3.13 Remainder

***Definition:*** `mod(a1,a2) = a1-int(a1/a2)*a2`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***MOD (generic)*** | I ←MOD(I,I), I1 ←MOD(I1,I1) †, I2 ←MOD(I2,I2) †, R ←MOD(R,R), D ←MOD(D,D), |
| ***MOD*** | I ←MOD(I,I) |
| ***I1MOD*** | I1 ←I1MOD(I1,I1) † |
| ***I2MOD*** | I2 ←I2MOD(I2,I2) † |
| ***AMOD*** | R ←AMOD(R,R) |
| ***DMOD*** | D ←DMOD(D,D) |

***Notes:*** The value of int(X) is X if X is of type INTEGER. If X is of type REAL or DOUBLE PRECISION, then int(X) is 0 if |X|<1 and the integer whose magnitude is the largest integer that does not exceed the magnitude of X and has the same sign of X if |X| > 1. If X is of type COMPLEX or COMPLEX*16, int(X) is int(real part of X).

The value of MOD, I1MOD, I2MOD, AMOD or DMOD is undefined if the value of a2 is 0.

## 12.3.14 Transfer of Sign

***Definition:*** `sign(a1,a2) =` |a1| if a2>=0; −|a1| if a2<0

| ***Name:*** | ***Usage:*** |
|---|---|
| ***SIGN (generic)*** | I ←SIGN(I,I), I1 ←SIGN(I1,I1) †, I2 ←SIGN(I2,I2) †, R ←SIGN(R,R), D ←SIGN(D,D) |
| ***ISIGN*** | I ←ISIGN(I,I) |
| ***I1SIGN*** | I1 ←I1SIGN(I1,I1) † |
| ***I2SIGN*** | I2 ←I2SIGN(I2,I2) † |
| ***SIGN*** | R ←SIGN(R,R) |
| ***DSIGN*** | D ←DSIGN(D,D) |

***Notes:*** If the value of a1 is 0, the result is 0 which has no sign.

## 12.3.15 Positive Difference

***Definition:*** `a1-a2` if a1>a2; `0` if a1<=a2

| ***Name:*** | ***Usage:*** |
|---|---|
| ***DIM (generic)*** | I ←DIM(I,I), I1 ←DIM(I1,I1) †, I2 ←DIM(I2,I2) †, R ←DIM(R,R), D ←DIM(D,D) |
| ***IDIM*** | I ←IDIM(I,I) |
| ***I1IDIM*** | I1 ←I1DIM(I1,I1) † |
| ***I2IDIM*** | I2 ←I2DIM(I2,I2) † |
| ***DIM*** | R ←DIM(R,R) |
| ***DDIM*** | D ←DDIM(D,D) |

## 12.3.16 Double Precision Product

***Definition:*** `a1*a2`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***DPROD*** | D ←DPROD(R,R) |

## 12.3.17 Choosing Largest Value

***Definition:*** `max(a1,a2,...)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***MAX (generic)*** | I ←MAX(I,...), I1 ←MAX(I1,...) †, I2 ←MAX(I2,...) †, R ←MAX(R,...), D ←MAX(D,...) |
| ***MAX0*** | I ←MAX0(I,...) |
| ***I1MAX0*** | I1 ←I1MAX0(I1,...) † |
| ***I2MAX0*** | I2 ←I2MAX0(I2,...) † |
| ***AMAX1*** | R ←AMAX1(R,...) |
| ***DMAX1*** | D ←DMAX1(D,...) |
| ***AMAX0*** | R ←AMAX0(I,...) |
| ***MAX1*** | I ←MAX1(R,...) |

## 12.3.18 Choosing Smallest Value

| | |
|---|---|
| ***Definition:*** | `min(a1,a2,...)` |
| ***Name:*** | ***Usage:*** |
| ***MIN (generic)*** | I ←MIN(I,...), I1 ←MIN(I1,...) †, I2 ←MIN(I2,...) †, R ←MIN(R,...), D ←MIN(D,...) |
| ***MIN0*** | I ←MIN0(I,...) |
| ***I1MIN0*** | I1 ←I1MIN0(I1,...) † |
| ***I2MIN0*** | I2 ←I2MIN0(I2,...) † |
| ***AMIN1*** | R ←AMIN1(R,...) |
| ***DMIN1*** | D ←DMIN1(D,...) |
| ***AMIN0*** | R ←AMIN0(I,...) |
| ***MIN1*** | I ←MIN1(R,...) |

## 12.3.19 Length

| | |
|---|---|
| ***Definition:*** | Length of character entity |
| ***Name:*** | ***Usage:*** |
| ***LEN*** | I ←LEN(CH) |
| ***Notes:*** | The argument to the LEN function need not be defined. |

## 12.3.20 Length Without Trailing Blanks

| | |
|---|---|
| ***Definition:*** | Length of character entity excluding trailing blanks |
| ***Name:*** | ***Usage:*** |
| ***LENTRIM*** | I ←LENTRIM(CH) |

## 12.3.21 Index of a Substring

***Definition:*** `index(a1,a2)` is location of substring `a2` in string `a1`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***INDEX*** | I ←INDEX(CH,CH) |

***Notes:*** INDEX(x,y) returns the starting position of a substring in x which is identical to y. The position of the first such substring is returned. If y is not contained in x, zero is returned.

## 12.3.22 Imaginary Part of Complex Number

***Definition:*** `ai`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***IMAG (generic) †*** | R ←IMAG(C), D ←IMAG(Z) |
| ***AIMAG*** | R ←AIMAG(C) |
| ***DIMAG*** | D ←DIMAG(Z) † |

***Notes:*** A complex number is an ordered pair of real numbers, `(ar,ai)` where `ar` is the real part and `ai` is the imaginary part of the complex number.

† is an extension to FORTRAN 77.

## 12.3.23 Conjugate of a Complex Number

***Definition:*** `(ar,-ai)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***CONJG (generic) †*** | C ←CONJG(C), Z ←CONJG(Z) |
| ***CONJG*** | C ←CONJG(C) |
| ***DCONJG*** | Z ←DCONJG(Z) † |

***Notes:*** A complex number is an ordered pair of real numbers, `(ar,ai)` where `ar` is the real part and `ai` is the imaginary part of the complex number.

† is an extension to FORTRAN 77.

## 12.3.24 Square Root

***Definition:*** `a**1/2`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***SQRT (generic)*** | R ←SQRT(R), D ←SQRT(D), C ←SQRT(C), Z ←SQRT(Z) † |
| ***SQRT*** | R ←SQRT(R) |
| ***DSQRT*** | D ←DSQRT(D) |
| ***CSQRT*** | C ←CSQRT(C) |
| ***CDSQRT*** | Z ←CDSQRT(Z) † |

***Notes:*** The argument to SQRT must be >= 0. The result of CSQRT and CDSQRT is the principal value with the real part >= 0. When the real part of the result is 0, the imaginary part is >= 0.

† is an extension to FORTRAN 77.

## 12.3.25 Exponential

***Definition:*** `e**a`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***EXP (generic)*** | R ←EXP(R), D ←EXP(D), C ←EXP(C), Z ←EXP(Z) † |
| ***EXP*** | R ←EXP(R) |
| ***DEXP*** | D ←DEXP(D) |
| ***CEXP*** | C ←CEXP(C) |
| ***CDEXP*** | Z ←CDEXP(Z) † |

***Notes:*** The result of a complex function is the principal value.

† is an extension to FORTRAN 77.

## 12.3.26 Natural Logarithm

***Definition:***

$$\log_e(a)$$

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LOG (generic)*** | R ←LOG(R), D ←LOG(D), C ←LOG(C), Z ←LOG(Z) † |
| ***ALOG*** | R ←ALOG(R) |
| ***DLOG*** | D ←DLOG(D) |
| ***CLOG*** | C ←CLOG(C) |
| ***CDLOG*** | Z ←CDLOG(Z) † |

***Notes:***

The value of `a` must be > 0. The argument of CLOG and CDLOG must not be (0,0). The result of CLOG and CDLOG is such that $-\pi <$ imaginary part of the result $\leq \pi$. The imaginary part of the result is $\pi$ only when the real part of the argument is < 0 and the imaginary part of the argument = 0.

The result of a complex function is the principal value.

† is an extension to FORTRAN 77.

## 12.3.27 Common Logarithm

***Definition:***

$$\log_{10}(a)$$

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LOG10 (generic)*** | R ←LOG10(R), D ←LOG10(D) |
| ***ALOG10*** | R ←ALOG10(R) |
| ***DLOG10*** | D ←DLOG10(D) |

## 12.3.28 Sine

| | |
|---|---|
| ***Definition:*** | `sin(a)` |
| ***Name:*** | ***Usage:*** |
| ***SIN (generic)*** | R ←SIN(R), D ←SIN(D), C ←SIN(C), Z ←SIN(Z) † |
| ***SIN*** | R ←SIN(R) |
| ***DSIN*** | D ←DSIN(D) |
| ***CSIN*** | C ←CSIN(C) |
| ***CDSIN*** | Z ←CDSIN(Z) † |
| ***Notes:*** | All angles are assumed to be in radians. |
| | The result of a complex function is the principal value. |
| | † is an extension to FORTRAN 77. |

## 12.3.29 Cosine

| | |
|---|---|
| ***Definition:*** | `cos(a)` |
| ***Name:*** | ***Usage:*** |
| ***COS (generic)*** | R ←COS(R), D ←COS(D), C ←COS(C), Z ←COS(Z) † |
| ***COS*** | R ←COS(R) |
| ***DCOS*** | D ←DCOS(D) |
| ***CCOS*** | C ←CCOS(C) |
| ***CDCOS*** | Z ←CDCOS(Z) † |
| ***Notes:*** | All angles are assumed to be in radians. |
| | The result of a complex function is the principal value. |
| | † is an extension to FORTRAN 77. |

## 12.3.30 Tangent

| | |
|---|---|
| ***Definition:*** | `tan(a)` |
| ***Name:*** | ***Usage:*** |
| ***TAN (generic)*** | R ←TAN(R), D ←TAN(D) |
| ***TAN*** | R ←TAN(R) |
| ***DTAN*** | D ←DTAN(D) |
| ***Notes:*** | All angles are assumed to be in radians. |

## 12.3.31 Cotangent

| | |
|---|---|
| ***Definition:*** | `cotan(a)` |
| ***Name:*** | ***Usage:*** |
| ***COTAN (generic) †*** | R ←COTAN(R), D ←COTAN(D) |
| ***COTAN*** | R ←COTAN(R) † |
| ***DCOTAN*** | D ←DCOTAN(D) † |
| ***Notes:*** | All angles are assumed to be in radians. |
| | † is an extension to FORTRAN 77. |

## 12.3.32 Arcsine

| | |
|---|---|
| ***Definition:*** | `arcsin(a)` |
| ***Name:*** | ***Usage:*** |
| ***ASIN (generic)*** | R ←ASIN(R), D ←ASIN(D) |
| ***ASIN*** | R ←ASIN(R) |
| ***DASIN*** | D ←DASIN(D) |
| ***Notes:*** | The absolute value of the argument of ASIN and DASIN must be <= 1. The result is such that $-\pi/2$ <= result <= $\pi/2$. |

## 12.3.33 Arccosine

| | |
|---|---|
| ***Definition:*** | `arccos(a)` |
| ***Name:*** | ***Usage:*** |
| ***ACOS (generic)*** | R ←ACOS(R), D ←ACOS(D) |
| ***ACOS*** | R ←ACOS(R) |
| ***DACOS*** | D ←DACOS(D) |
| ***Notes:*** | The absolute value of the argument of ACOS and DACOS must be <= 1. The result is such that 0 <= result <= π. |

## 12.3.34 Arctangent

| | |
|---|---|
| ***Definition:*** | `arctan(a)` |
| ***Name:*** | ***Usage:*** |
| ***ATAN (generic)*** | R ←ATAN(R), D ←ATAN(D) |
| ***ATAN*** | R ←ATAN(R) |
| ***DATAN*** | D ←DATAN(D) |
| ***Definition:*** | `arctan(a1/a2)` |
| ***Name:*** | ***Usage:*** |
| ***ATAN2 (generic)*** | R ←ATAN2(R,R), D ←ATAN2(D,D) |
| ***ATAN2*** | R ←ATAN2(R,R) |
| ***DATAN2*** | D ←DATAN2(D,D) |
| ***Notes:*** | The result of ATAN and DATAN is such that -π/2 <= result <= π/2. If the value of the first argument of ATAN2 and DATAN2 is positive then the result is positive. If the value of the first argument is 0, the result is 0 if the second argument is positive and π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is 0, the absolute value of the result is π/2. The arguments must not both be 0. The result of ATAN2 and DATAN2 is such that -π < result <= π. |

### 12.3.35 Hyperbolic Sine

| | |
|---|---|
| ***Definition:*** | `sinh(a)` |
| ***Name:*** | ***Usage:*** |
| ***SINH (generic)*** | R ←SINH(R) D ←SINH(D) |
| ***SINH*** | R ←SINH(R) |
| ***DSINH*** | D ←DSINH(D) |

### 12.3.36 Hyperbolic Cosine

| | |
|---|---|
| ***Definition:*** | `cosh(a)` |
| ***Name:*** | ***Usage:*** |
| ***COSH (generic)*** | R ←COSH(R), D ←COSH(D) |
| ***COSH*** | R ←COSH(R) |
| ***DCOSH*** | D ←DCOSH(D) |

### 12.3.37 Hyperbolic Tangent

| | |
|---|---|
| ***Definition:*** | `tanh(a)` |
| ***Name:*** | ***Usage:*** |
| ***TANH (generic)*** | R ←TANH(R), D ←TANH(D) |
| ***TANH*** | R ←TANH(R) |
| ***DTANH*** | D ←DTANH(D) |

## 12.3.38 Gamma Function

***Definition:*** `gamma(a)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***GAMMA (generic)*** | R ←GAMMA(R), D ←GAMMA(D) |
| ***GAMMA*** | R ←GAMMA(R) |
| ***DGAMMA*** | D ←DGAMMA(D) |

## 12.3.39 Natural Log of Gamma Function

***Definition:***

$$\log_e(\text{gamma}(a))$$

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LGAMMA (generic)*** | R ←LGAMMA(R), D ←LGAMMA(D) |
| ***ALGAMA*** | R ←ALGAMA(R) |
| ***DLGAMA*** | D ←DLGAMA(D) |

## 12.3.40 Error Function

***Definition:*** `erf(a)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ERF (generic)*** | R ←ERF(R), D ←ERF(D) |
| ***ERF*** | R ←ERF(R) |
| ***DERF*** | D ←DERF(D) |

## 12.3.41 Complement of Error Function

***Definition:*** `1-erf(a)`

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ERFC (generic)*** | R ←ERFC(R), D ←ERFC(D) |
| ***ERFC*** | R ←ERFC(R) |
| ***DERFC*** | D ←DERFC(D) |

## 12.3.42 Lexically Greater Than or Equal

***Definition:*** a1>=a2

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LGE*** | L ←LGE(CH,CH) |

***Notes:*** The ASCII collating sequence is used to evaluate the relation.

## 12.3.43 Lexically Greater Than

***Definition:*** a1>a2

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LGT*** | L ←LGT(CH,CH) |

***Notes:*** The ASCII collating sequence is used to evaluate the relation.

## 12.3.44 Lexically Less Than or Equal

***Definition:*** a1<=a2

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LLE*** | L ←LLE(CH,CH) |

***Notes:*** The ASCII collating sequence is used to evaluate the relation.

## 12.3.45 Lexically Less Than

| | |
|---|---|
| ***Definition:*** | a1<a2 |
| ***Name:*** | ***Usage:*** |
| ***LLT*** | L ←LLT(CH,CH) |
| ***Notes:*** | The ASCII collating sequence is used to evaluate the relation. |

## 12.3.46 Binary Pattern Processing Functions: Boolean AND

| | |
|---|---|
| ***Definition:*** | `iand(i,j)` Boolean AND |
| ***Name:*** | ***Usage:*** |
| ***IAND (generic)*** | I ←IAND(I,I), I1 ←IAND(I1,I1), I2 ←IAND(I2,I2) |
| ***IAND*** | I ←IAND(I,I) |
| ***I1AND*** | I1 ←I1AND(I1,I1) |
| ***I2AND*** | I2 ←I2AND(I2,I2) |

## 12.3.47 Binary Pattern Processing Functions: Boolean Inclusive OR

| | |
|---|---|
| ***Definition:*** | `ior(i,j)` Boolean inclusive OR |
| ***Name:*** | ***Usage:*** |
| ***IOR (generic)*** | I ←IOR(I,I), I1 ←IOR(I1,I1), I2 ←IOR(I2,I2) |
| ***IOR*** | I ←IOR(I,I) |
| ***I1OR*** | I1 ←I1OR(I1,I1) |
| ***I2OR*** | I2 ←I2OR(I2,I2) |

## 12.3.48 Binary Pattern Processing Functions: Boolean Exclusive OR

***Definition:*** `ieor(i,j)` Boolean exclusive OR

| ***Name:*** | ***Usage:*** |
|---|---|
| ***IEOR (generic)*** | I ←IEOR(I,I), I1 ←IEOR(I1,I1), I2 ←IEOR(I2,I2) |
| ***IEOR*** | I ←IEOR(I,I) |
| ***I1EOR*** | I1 ←I1EOR(I1,I1) |
| ***I2EOR*** | I2 ←I2EOR(I2,I2) |

## 12.3.49 Binary Pattern Processing Functions: Boolean Complement

***Definition:*** `not(i)` Boolean complement

| ***Name:*** | ***Usage:*** |
|---|---|
| ***NOT (generic)*** | I ←NOT(I), I1 ←NOT(I1), I2 ←NOT(I2) |
| ***NOT*** | I ←NOT(I) |
| ***I1NOT*** | I1 ←I1NOT(I1) |
| ***I2NOT*** | I2 ←I2NOT(I2) |

## 12.3.50 Binary Pattern Processing Functions: Logical Shift

***Definition:*** `ishl(j,n)` Logical shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ISHL (generic)*** | I ←ISHL(I,I), I1 ←ISHL(I1,I1), I2 ←ISHL(I2,I2) |
| ***ISHL*** | I ←ISHL(I,I) |
| ***I1ISHL*** | I1 ←I1SHL(I1,I1) |
| ***I2ISHL*** | I2 ←I2SHL(I2,I2) |

***Definition:*** `ishft(j,n)` Logical shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ISHFT (generic)*** | I ←ISHFT(I,I), I1 ←ISHFT(I1,I1), I2 ←ISHFT(I2,I2) |
| ***ISHFT*** | I ←ISHFT(I,I) |
| ***I1ISHFT*** | I1 ←I1SHFT(I1,I1) |
| ***I2ISHFT*** | I2 ←I2SHFT(I2,I2) |

***Notes:*** There are three shift operations: logical, arithmetic and circular. These shift operations are implemented as integer functions having two arguments. The first argument, `j,` is the value to be shifted and the second argument, `n,` is the number of bits to shift. If `n` is less than 0, a right shift is performed. If `n` is greater than 0, a left shift is performed. If `n` is equal to 0, no shift is performed. Note that the arguments are not modified.

In a logical shift, bits shifted out from the left or right are lost. Zeros are shifted in from the opposite end.

In an arithmetic shift, `j` is considered a signed integer. In the case of a right shift, zeros are shifted into the left if `j` is positive and ones if `j` is negative. Bits shifted out of the right are lost. In the case of a left shift, zeros are shifted into the right and bits shifted out of the left are lost.

In a circular shift, bits shifted out one end are shifted into the opposite end. No bits are lost.

## 12.3.51 Binary Pattern Processing Functions: Arithmetic Shift

***Definition:*** `isha(j,n)` Arithmetic shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ISHA (generic)*** | I ←ISHA(I,I), I1 ←ISHA(I1,I1), I2 ←ISHA(I2,I2) |
| ***ISHA*** | I ←ISHA(I,I) |
| ***I1ISHA*** | I1 ←I1SHA(I1,I1) |
| ***I2ISHA*** | I2 ←I2SHA(I2,I2) |

***Notes:*** There are three shift operations: logical, arithmetic and circular. These shift operations are implemented as integer functions having two arguments. The first argument, `j,` is the value to be shifted and the second argument, `n,` is the number of bits to shift. If `n` is less than 0, a right shift is performed. If `n` is greater than 0, a left shift is performed. If `n` is equal to 0, no shift is performed. Note that the arguments are not modified.

In a logical shift, bits shifted out from the left or right are lost. Zeros are shifted in from the opposite end.

In an arithmetic shift, `j` is considered a signed integer. In the case of a right shift, zeros are shifted into the left if `j` is positive and ones if `j` is negative. Bits shifted out of the right are lost. In the case of a left shift, zeros are shifted into the right and bits shifted out of the left are lost.

In a circular shift, bits shifted out one end are shifted into the opposite end. No bits are lost.

## 12.3.52 Binary Pattern Processing Functions: Circular Shift

***Definition:*** `ishc(j,n)` Circular shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ISHC (generic)*** | I ←ISHC(I,I), I1 ←ISHC(I1,I1), I2 ←ISHC(I2,I2) |
| ***ISHC*** | I ←ISHC(I,I) |
| ***I1ISHC*** | I1 ←I1SHC(I1,I1) |
| ***I2ISHC*** | I2 ←I2SHC(I2,I2) |

***Notes:*** There are three shift operations: logical, arithmetic and circular. These shift operations are implemented as integer functions having two arguments. The first argument, `j,` is the value to be shifted and the second argument, `n,` is the number of bits to shift. If `n` is less than 0, a right shift is performed. If `n` is greater than 0, a left shift is performed. If `n` is equal to 0, no shift is performed. Note that the arguments are not modified.

In a logical shift, bits shifted out from the left or right are lost. Zeros are shifted in from the opposite end.

In an arithmetic shift, `j` is considered a signed integer. In the case of a right shift, zeros are shifted into the left if `j` is positive and ones if `j` is negative. Bits shifted out of the right are lost. In the case of a left shift, zeros are shifted into the right and bits shifted out of the left are lost.

In a circular shift, bits shifted out one end are shifted into the opposite end. No bits are lost.

## 12.3.53 Binary Pattern Processing Functions: Bit Testing

***Definition:*** Test bit - a2'th bit of a1 is tested. If it is 1, .TRUE. is returned. If it is 0, .FALSE. is returned.

| ***Name:*** | ***Usage:*** |
|---|---|
| ***BTEST (generic)*** | L ←BTEST(I,I), L ←BTEST(I1,I1), L ←BTEST(I2,I2) |
| ***BTEST*** | L ←BTEST(I,I) |
| ***I1BTEST*** | L ←I1BTEST(I1,I1) |
| ***I2BTEST*** | L ←I2BTEST(I2,I2) |

## 12.3.54 Binary Pattern Processing Functions: Set Bit

***Definition:*** Set bit - Return a1 with a2'th bit set.

| ***Name:*** | ***Usage:*** |
|---|---|
| ***IBSET (generic)*** | I ←IBSET(I,I), I1 ←IBSET(I1,I1), I2 ←IBSET(I2,I2) |
| ***IBSET*** | I ←IBSET(I,I) |
| ***I1IBSET*** | I1 ←I1BSET(I1,I1) |
| ***I2IBSET*** | I2 ←I2BSET(I2,I2) |

## 12.3.55 Binary Pattern Processing Functions: Clear Bit

***Definition:*** Clear bit - Return a1 with a2'th bit cleared.

| ***Name:*** | ***Usage:*** |
|---|---|
| ***IBCLR (generic)*** | I ←IBCLR(I,I), I1 ←IBCLR(I1,I1), I2 ←IBCLR(I2,I2) |
| ***IBCLR*** | I ←IBCLR(I,I) |
| ***I1IBCLR*** | I1 ←I1BCLR(I1,I1) |
| ***I2IBCLR*** | I2 ←I2BCLR(I2,I2) |

## 12.3.56 Binary Pattern Processing Functions: Change Bit

***Definition:*** Change bit - Return a1 with a2'th bit complemented.

| ***Name:*** | ***Usage:*** |
|---|---|
| ***IBCHNG (generic)*** | I ←IBCHNG(I,I), I1 ←IBCHNG(I1,I1), I2 ←IBCHNG(I2,I2) |
| ***IBCHNG*** | I ←IBCHNG(I,I) |
| ***I1IBCHNG*** | I1 ←I1BCHNG(I1,I1) |
| ***I2IBCHNG*** | I2 ←I2BCHNG(I2,I2) |

## 12.3.57 Binary Pattern Processing Functions: Arithmetic Shifts

***Definition:*** `lshift(j,n)` Arithmetic left shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LSHIFT (generic)*** | I ←LSHIFT(I,I), I1 ←LSHIFT(I1,I1), I2 ←LSHIFT(I2,I2) |
| ***LSHIFT*** | I ←LSHIFT(I,I) |
| ***I1LSHIFT*** | I1 ←I1LSHIFT(I1,I1) |
| ***I2LSHIFT*** | I2 ←I2LSHIFT(I2,I2) |

***Definition:*** `rshift(j,n)` Arithmetic right shift

| ***Name:*** | ***Usage:*** |
|---|---|
| ***RSHIFT (generic)*** | I ←RSHIFT(I,I), I1 ←RSHIFT(I1,I1), I2 ←RSHIFT(I2,I2) |
| ***RSHIFT*** | I ←RSHIFT(I,I) |
| ***I1RSHIFT*** | I1 ←I1RSHIFT(I1,I1) |
| ***I2RSHIFT*** | I2 ←I2RSHIFT(I2,I2) |

***Notes:***

With these shift functions, `n` must be a non-negative integer. In an arithmetic shift, `j` is considered a signed integer. In the case of a left shift, zeros are shifted into the right and bits shifted out of the left are lost. In the case of a right shift, zeros are shifted into the left if `j` is positive and ones if `j` is negative. Bits shifted out of the right are lost.

If `n` is equal to 0, no shift is performed. Note that the arguments are not modified.

These functions are compiled as in-line code unless they are passed as arguments.

## 12.3.58 Allocated Array

***Definition:*** Is array A allocated?

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ALLOCATED*** | L ←ALLOCATED(A) |

## 12.3.59 Memory Location

***Definition:*** Location of A where A is any variable, array or array element

| ***Name:*** | ***Usage:*** |
|---|---|
| ***LOC*** | I ←LOC(A) |

## 12.3.60 Size of Variable or Structure

***Definition:*** Size of A in bytes where A is any constant, variable, array, or structure

| ***Name:*** | ***Usage:*** |
|---|---|
| ***ISIZEOF*** | I ←ISIZEOF(A) |

***Notes:*** The size reported for a constant or simple variable is based on its type. The size of a CHARACTER constant is the number of characters in the constant. The size reported for an array is the size of the storage area required for the array. The size reported for a structure is the size of the storage area required for the structure. An assumed-size CHARACTER variable, assumed-size array, or allocatable array has size 0.

## 12.3.61 Volatile Reference

***Definition:*** A is a volatile reference

| ***Name:*** | ***Usage:*** |
|---|---|
| ***VOLATILE*** | A ←VOLATILE(A) |

***Notes:*** A volatile reference to a symbol indicates that the value of the symbol may be modified in ways that are unknown to the subprogram. For example, a symbol in common being referenced in a subprogram may be modified by another subprogram that is processing an asynchronous interrupt. Therefore, any subprogram that is referencing the symbol to determine its value should reference this symbol using the `VOLATILE` intrinsic function so that the value currently being evaluated agrees with the value last stored.

# 12.4 External Functions

An *external function* is a program unit that has a **FUNCTION** statement as its first statement. It is defined externally to the program units that reference it. The form of a **FUNCTION** statement is defined in the chapter entitled "FORTRAN Statements" on page 9.

The name of an external function is treated as if it was a variable. It is through the function name that the result of an external function becomes defined. This variable must become defined before the execution of the external function is completed. Once defined, it can be referenced or redefined. The value of this variable when a **RETURN** or **END** statement is executed is the result returned by the external function.

*Example:*

```
          INTEGER FUNCTION VECSUM( A, N )
          INTEGER A(N), I
          VECSUM = 0
          DO 10 I = 1, N
              VECSUM = VECSUM + A(I)
   10     CONTINUE
          END
```

If the variable representing the return value of the external function is of type CHARACTER with a length specification of `(*)`, it must not be the operand of a concatenation operator unless it appears in a character assignment statement.

It is also possible for an external function to return results through its dummy arguments by assigning to them. The following example demonstrates this.

*Example:*

```
      INTEGER MARKS(40), N
      REAL AVG, STDDEV, MEAN
      PRINT *, 'Enter number of marks'
      READ( 5, * ) N
      PRINT *, 'Enter marks'
      READ( 5, * ) (MARKS(I), I = 1, N)
      AVG = MEAN( MARKS, N, STDDEV )
      PRINT *, 'Mean = ', AVG,
     $          ' Standard Deviation = ', STDDEV
      END

*
* Define function MEAN to return the average by
* defining the function name and return the standard
* deviation by defining a dummy argument.
*
      REAL FUNCTION MEAN( A, N, STDDEV )
      INTEGER A, N, I
      REAL STDDEV
      DIMENSION A(N)
      MEAN = 0
      DO 10 I = 1, N
          MEAN = MEAN + A(I)
10    CONTINUE
      MEAN = MEAN / N
      STDDEV = 0
      DO 20 I = 1, N
          STDDEV = STDDEV + ( A(I) - MEAN )**2
20    CONTINUE
      STDDEV = SQRT( STDDEV / (N - 1) )
      END
```

## 12.4.1 Referencing an External Function

When an external function is referenced in an expression or a **CALL** statement, the following steps are performed.

1. The actual arguments are evaluated.
2. The actual arguments are associated with the corresponding dummy arguments.
3. The external function is executed.

The type of the external function reference must be the same as the type of the function name in the external function subprogram. If the external function is of type CHARACTER, the length must also match.

## 12.4.2 Actual Arguments for an External Function

An actual argument must be one of the following.

1. Any expression except a character expression involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant.
2. An array name.
3. An intrinsic function name (must be the specific name) that has appeared in an **INTRINSIC** statement.

4. An external procedure name.
5. A dummy procedure name.

The actual arguments of an external function reference must match the order, number and type of the corresponding dummy arguments. If a subroutine is an actual argument, then type agreement is not required since a subroutine has no type.

### 12.4.3 External Function Subprogram Restrictions

1. The name of an external function is a global name and must not be the same as any other global name or name local to the subprogram whose name is that of the external function. Note that the external function name is treated as a variable within the external function subprogram.

2. The name of a dummy argument is a name local to the subprogram and must not appear in an **EQUIVALENCE**, **PARAMETER**, **SAVE**, **INTRINSIC** or **DATA** statement within the same subprogram. It may appear in a **COMMON** statement only as the name of a common block.

3. The name of the external function can in no way, directly or indirectly, be referenced as a subprogram from within the subprogram it defines. It can appear in a type statement to establish its type only if the type has not been established in the **FUNCTION** statement.

## 12.5 Subroutines

A *subroutine* is a program unit that has a **SUBROUTINE** statement as it first statement. It is defined externally to the program units that reference it. The form of a **SUBROUTINE** statement can be found in the chapter entitled "FORTRAN Statements" on page 9.

A subroutine differs from a function in that it does not return a result and hence has no type associated with it. However, it is possible to return values from a subroutine by defining or redefining the dummy arguments of the subroutine.

### 12.5.1 Referencing a Subroutine: The CALL Statement

Unlike a function, a subroutine cannot appear in an expression. Subroutines are referenced by using a **CALL** statement. See the chapter entitled "FORTRAN Statements" on page 9 for details on the **CALL** statement. When a **CALL** statement is executed, the following steps are performed.

1. The actual arguments are evaluated.
2. The actual arguments are associated with the corresponding dummy arguments.
3. The subroutine is executed.

A subroutine can be called from any subprogram but must not be called by itself, indirectly or directly.

### 12.5.2 Actual Arguments for a Subroutine

Each actual argument in a subroutine call must be one of the following.

1. Any expression except a character expression involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant.

2. An array name.
3. An intrinsic function name (must be the specific name) that has appeared in an **INTRINSIC** statement.
4. An external procedure name.
5. A dummy procedure name.
6. An *alternate return specifier* of the form `*s` where `s` is a statement number of an executable statement in the subprogram which contained the **CALL** statement. This will be covered in more detail when the **RETURN** statement is discussed.

The actual arguments must agree in order, number and type with the corresponding dummy arguments. The type agreement does not apply to an actual argument which is an alternate return specifier or a subroutine name since neither has a type.

### 12.5.3 Subroutine Subprogram Restrictions

1. A subroutine subprogram can contain any statement except a **FUNCTION**, **BLOCK DATA** or **PROGRAM** statement.

2. The name of a subroutine is a global name and must not be used as another global name. Furthermore, no local name in the subroutine subprogram can have the same name as the subroutine.

3. The name of a dummy argument is local to the subroutine subprogram and must not appear in an **EQUIVALENCE**, **PARAMETER**, **SAVE**, **INTRINSIC** or **DATA** statement. It may appear in a **COMMON** statement only as the name of a common block.

## 12.6 The ENTRY Statement

An **ENTRY** statement allows execution of a subprogram to begin at a particular executable statement within the subprogram in which it appears. An **ENTRY** statement defines an alternate *entry point* into a subprogram and can appear anywhere after the **FUNCTION** statement in a function subprogram or the **SUBROUTINE** statement in a subroutine subprogram. Also, it must not appear as a statement between the beginning and end of a control structure. For example, an **ENTRY** statement cannot appear between a block **IF** statement and its corresponding **END IF** statement or between a **DO** statement and the corresponding terminal statement. It is possible to have more than one **ENTRY** statement in a subprogram. An **ENTRY** statement is a non-executable statement. The form of an **ENTRY** statement can be found in the chapter entitled "FORTRAN Statements" on page 9.

Each entry name defines an external function if it appears in a function, or an external subroutine if it appears in a subroutine and is referenced in the same way as the actual function or subroutine name would be referenced. Execution begins at the first executable statement that follows the **ENTRY** statement. The order, number, type and names of the dummy argument lists of an **ENTRY** statement may be different from that of a **FUNCTION**, **SUBROUTINE** or other **ENTRY** statement. However, there must still be agreement between the actual argument list used to reference an entry name and the dummy argument list in the corresponding **ENTRY** statement.

## 12.6.1 ENTRY Statements in External Functions

Entry names may also appear in type statements. Their type may or may not be the same type as other entry names or the actual name of the external function unless the function is of type CHARACTER. If the function is of type CHARACTER then the type of all the entry names must be of type CHARACTER. Conversely, if an entry name is of type CHARACTER, then all other entry names and the function name must be of type CHARACTER. An entry name, like external function names, is treated as a variable within the subprogram it appears. Within a function subprogram, there is an association between variables whose name is an entry name and the variable whose name corresponds to the external function. When such a variable becomes defined, all other such variables of the *same* type also become defined and other such variables not of the same type become undefined. This can be best illustrated by an example.

*Example:*

```
          PRINT *, EVAL(2), EVAL3(4.0)
          END

          INTEGER FUNCTION EVAL( X )
          INTEGER EVAL2, X
          REAL EVAL3, Y
          C = 1
          GOTO 10
          ENTRY EVAL2( X )
          C = 2
          GOTO 10
          ENTRY EVAL3( Y )
          C = 3
10        EVAL2 = C * X
          END
```

In the previous example, invoking `EVAL` would cause the result of 2 to be returned even though `EVAL` was never assigned to in the function `EVAL` but since `EVAL2` and `EVAL` are of the same type they are associated and hence defining `EVAL2` causes `EVAL` to be defined. However, invoking `EVAL3` would cause an undefined result to be returned since `EVAL3` is of type REAL and `EVAL2` is of type INTEGER and hence are not associated. `EVAL3` does not become defined.

## 12.6.2 ENTRY Statement Restrictions

1. An entry name may not appear in any statement previous to the **`ENTRY`** statement containing the entry name except in a type statement.

2. If an entry name in a function is of type CHARACTER, each entry name and the name of the function must also be of type CHARACTER. If the name of the function or the name of any entry point has a length specification of `(*)`, then all such entities must have a length specification of `(*)` otherwise they must all have a length specification of the same integer value.

3. If a dummy argument appears in an executable statement, then that statement can be executed provided that the dummy argument is in the dummy argument list of the procedure name referenced.

4. A name that appears as a dummy argument in an **`ENTRY`** statement must not appear in the expression of a statement function unless it is a dummy argument of the statement function, it has appeared in the dummy argument list of a **`FUNCTION`** or **`SUBROUTINE`** statement, or the **`ENTRY`** statement appears before the statement function statement.

5. A name that appears as a dummy argument in an **ENTRY** statement must not appear in an executable statement preceding the **ENTRY** statement unless it has also appeared in a **FUNCTION**, **SUBROUTINE**, or **ENTRY** statement that precedes the executable statement.

# 12.7 The RETURN Statement

A **RETURN** statement is a way to terminate the execution of a function or subroutine subprogram and return control to the program unit that referenced it. As an extension to FORTRAN 77, Open Watcom FORTRAN 77 permits the use of the **RETURN** statement in the main program. A subprogram (or main program) may contain more than one **RETURN** statement or it may contain no **RETURN** statement. In the latter case, the **END** statement has the same effect as a **RETURN** statement.

Execution of a **RETURN** or **END** statement causes all local entities to become undefined except for the following.

1. Entities specified in a **SAVE** statement.
2. Entities in blank common.
3. Initially defined entities that have neither been redefined nor become undefined.
4. Entities in a named common block that appears in the subprogram and in a program unit that references the subprogram directly or indirectly.

## 12.7.1 RETURN Statement in the Main Program (Extension)

The form of a **RETURN** statement in a main program is:

```
      RETURN
```

When a **RETURN** statement is executed in the main program, program execution terminates in the same manner as the **STOP** or **END** statement. This is an extension to FORTRAN 77.

## 12.7.2 RETURN Statement in Function Subprograms

The form of a **RETURN** statement in a function subprogram is:

```
      RETURN
```

When a **RETURN** statement is executed in a function subprogram, the function value must be defined. Control is then passed back to the program unit that referenced it.

### 12.7.3 RETURN Statement in Subroutine Subprograms

The form of a **RETURN** statement in a subroutine subprogram is:

```
RETURN [ e ]
```

***where:***

***e*** is an integer expression.

If the expression e is omitted or has a value less than one or greater than the number of asterisks appearing in the dummy argument list of the subroutine or entry name referenced, then control is returned to the next executable statement that follows the **CALL** statement in the referencing program unit. If 1 <= e <= n where n is the number of asterisks appearing in the **SUBROUTINE** or **ENTRY** statement which contains the referenced name, then the expression e identifies the eth asterisk in the dummy argument list. Control is returned to the statement identified by the alternate return specified in the **CALL** statement that corresponds to the eth asterisk in the dummy argument list of the referenced subroutine. The following example demonstrates the use of alternate return specifiers in conjunction with the **RETURN** statement.

*Example:*

```
      REAL X, Y
      READ *, X, Y
      CALL CMP( X, Y, *10, *20 )
      PRINT *, 'X equals Y'
      GOTO 30
10    PRINT *, 'X less than Y'
      GOTO 30
20    PRINT *, 'X greater than Y'
30    END

      SUBROUTINE CMP( X, Y, *, * )
      IF( X .LT. Y )RETURN 1
      IF( X .GT. Y )RETURN 2
      RETURN
      END
```

## 12.8 Subprogram Arguments

Arguments provide a means of communication between program units. Arguments are passed to subprograms through argument lists and are received by subprograms through argument lists. The argument list used to pass arguments to a subprogram is called the *actual argument list* and the arguments are called *actual arguments.* The argument list of the receiving subprogram is called the *dummy argument list* and the arguments are called *dummy arguments.* The actual argument list must agree with the dummy argument list in number, order and type.

## 12.8.1 Dummy Arguments

Statement function, external functions and subroutines use dummy arguments to define the type of actual arguments they expect. A dummy argument is one of the following.

1. Variable.
2. Array.
3. Dummy procedure.
4. Asterisk `(*)` indicating a statement label.

*Notes:*

1. A statement function dummy argument may only be a variable.

2. An asterisk can only be a dummy argument for a subroutine subprogram.

Dummy arguments that are variables of type INTEGER can be used in dummy array declarators. No dummy argument may appear in an **`EQUIVALENCE`**, **`DATA`**, **`PARAMETER`**, **`SAVE`**, **`INTRINSIC`** or **`COMMON`** statement except as a common block name. A dummy argument must not be the same name as the subprogram name specified in the **`FUNCTION`**, **`SUBROUTINE`** or **`ENTRY`** statement. Other than these restrictions, dummy arguments can be used in the same way an actual name of the same class would be used.

## 12.8.2 Actual Arguments

Actual arguments specify the entities that are to be associated with the dummy arguments when referencing a subroutine or function. Actual arguments can be any of the following.

1. Any expression, except character expression involving the concatenation of an operand whose length specification is `(*)` unless the operand is a symbolic constant.
2. An array name.
3. An intrinsic function name.
4. An external function or subroutine name.
5. A dummy procedure name.
6. An *alternate return specifier* of the form `*s` where `s` is a statement number of an executable statement in the subprogram which contained the **`CALL`** statement.

*Notes:*

1. A statement function actual argument can only be a variable or an expression.

2. An alternate return specifier can only be an actual argument in the actual argument list of a subroutine reference.

## 12.8.3 Association of Actual and Dummy Arguments

When a function or subroutine reference is executed, an association is established between the actual arguments and the corresponding dummy arguments. The first dummy argument is associated with the first actual argument, the second dummy argument is associated with the second actual argument, etc. Association requires that the types of the actual and dummy arguments agree. A subroutine has no type and when used as an actual argument must be associated with a dummy procedure. An alternate return specifier

has no type and must be associated with an asterisk. Arguments can be passed through more than one level of procedure reference. In this case, valid association must exist at all intermediate levels as well as the last level. Argument association is terminated upon the execution of a **RETURN** or **END** statement.

### 12.8.3.1 Length of Character Actual and Dummy Arguments

If a dummy argument is of type CHARACTER, the corresponding actual argument must also be of type CHARACTER and the length of the dummy argument must be less than or equal to the length of the actual argument. If the length of the dummy argument is `len` then the `len` leftmost characters of the actual argument are associated with the dummy argument.

If a dummy argument of type CHARACTER is an array name, then the restriction on the length is on the whole array and not for each array element. The length of an array element of the dummy argument may be different from the length of the array element of the corresponding actual array, array element, or array element substring, but the dummy array argument must not extend beyond the end of the associated actual array.

### 12.8.3.2 Variables as Dummy Arguments

A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, substring or expression. Only if the actual argument is a variable, array element or substring can the corresponding actual argument be redefined.

### 12.8.3.3 Arrays as Dummy Arguments

A dummy argument that is an array may be associated with an actual argument that is an array, array element or array element substring. The number and size of the dimensions in the actual argument array declarator may be different from the number and size of the dimensions in the dummy argument array declarator.

If the actual argument is a non-character array name, then the size of the dummy argument array must not exceed the size of the actual argument array. An element of the actual array becomes associated with the element in the dummy array with the same subscript value. Association by array element of character arrays exists only if the lengths of the array elements are the same. If their lengths are not the same, the dummy and actual array elements will not consist of the same characters.

If the actual argument is a non-character array element name whose subscript value is *asv* the size of the dummy argument array must not exceed the size of the actual argument array less *asv - 1*. Furthermore, the dummy argument array element whose subscript value is *dsv* becomes associated with the actual argument array element whose subscript value is *asv + dsv - 1*. Consider the following example.

*Example:*

```
DIMENSION A(10)
CALL SAM( A(3) )
END

SUBROUTINE SAM( B )
DIMENSION B(5)
     .
     .
     .
END
```

In the previous example, array `A` is an actual argument and the array `B` is the dummy argument. Suppose we wanted to know which element of `A` is associated with the 4th element of `B`. Then *asv* would have value 3 since the array element `A(3)` is the actual argument, and *dsv* is 4. Then the 4th element in `B` is 3 + 4 - 1 = 6th element of A.

If the actual argument is a character array name, character array element name or character array element substring which begins at character storage unit *ach,* then the character storage unit *dch* of the dummy argument array is associated with the character storage unit *ach + dch - 1* of the actual array. The size of the dummy character array must not exceed the size of the actual argument array.

### 12.8.3.4 Procedures as Dummy Arguments

A dummy argument that is a dummy procedure can only be associated with an actual argument that is one of the following.

1. Intrinsic function.
2. External function.
3. External Subroutine.
4. Another dummy procedure.

If the dummy argument is used as a subroutine (that is it is invoked using a **`CALL`** statement) then the corresponding actual argument must either be a subroutine or a dummy procedure. If the dummy argument is used as an external function, then the corresponding actual argument must be an intrinsic function, external function or dummy procedure. Note that it may not be possible to determine in a given program unit whether a dummy procedure is associated with a function or subroutine. In the following example it is not possible to tell by looking at this program unit whether `PROC` is an external subroutine or function.

*Example:*

```
SUBROUTINE SAM( PROC )
EXTERNAL PROC
     .
     .
CALL SAM1( PROC )
     .
     .
     .
END
```

### 12.8.3.5 Asterisks as Dummy Arguments

A dummy argument that is an asterisk may only appear in the dummy argument list of a **SUBROUTINE** statement or an **ENTRY** statement in a subroutine subprogram and may be associated only with an actual argument that is an alternate return specifier in a **CALL** statement which references the subroutine.

*Example:*

```
          CHARACTER*10 RECORD(5)
          I = 2
          CALL SAM( I, *999, 3HSAM )
          PRINT *, 'I should be skipped'
    999   PRINT *, 'I should be printed'
          END
          SUBROUTINE SAM( I, *, K)
          CHARACTER*3 K
          PRINT *, K
          RETURN 1
          END
```

# *Appendices*

# A. Open Watcom FORTRAN 77 Extensions to Standard FORTRAN 77

This appendix summarizes the extensions supported by Open Watcom FORTRAN 77.

1. The **INCLUDE** statement for embedding source from another file is supported.

```
INCLUDE 'SRC'
```

2. Symbolic names are unique up to 32 characters. Also, '$', '_', and lowercase letters are allowed in symbolic names.

3. Lowercase and uppercase letters are treated in the same way except in:

    1. character and hollerith constants
    2. apostrophe and H edit descriptors

4. End-of-line comments are permitted.

```
PRINT *, 'Hello world'  ! print 'Hello World'
```

5. The **IMPLICIT NONE** statement is supported.

6. An asterisk enclosed in parentheses is allowed with the type **CHARACTER** when specified in an **IMPLICIT** statement.

```
IMPLICIT CHARACTER*(*) (Z)
```

7. Length specifiers are allowed with types specified in **IMPLICIT** statements.

```
IMPLICIT INTEGER*2 (I-N)
```

8. Length specifiers are allowed with type specification statements.

```
LOGICAL*1, LOGICAL*4
INTEGER*1, INTEGER*2, INTEGER*4
REAL*4, REAL*8
COMPLEX*8, COMPLEX*16
```

   Length specifiers are also allowed with the type specified in **FUNCTION** statements.

```
COMPLEX*16 FUNCTION ZADD( X, Y )
```

9. Length specifiers are allowed with symbol names.

```
INTEGER I*2, A*2(10), B(20)*2

COMPLEX FUNCTION ZADD*16( X, Y )
```

10. The **DOUBLE COMPLEX** statement is supported (equivalent to **COMPLEX*16**).

11. Double precision complex constants are allowed.

```
Z = (1D0, 2D0)
```

12. Mixing operands of type **DOUBLE PRECISION** and **COMPLEX** to yield a **COMPLEX*16** result is allowed.

```
DOUBLE PRECISION X
COMPLEX Y, Z*16
Z = X + Y
```

13. User-defined structures are supported.

```
STRUCTURE/END STRUCTURE
UNION/END UNION
MAP/END MAP
RECORD
```

14. Both character and non-character data are allowed in the same common block.

```
INTEGER X
CHARACTER C
COMMON /BLK/ X, C
```

15. Data initialization of variables in common without a block data subprogram is allowed.

16. Equivalencing character to non-character data is permitted.

```
INTEGER X
CHARACTER C
EQUIVALENCE (X, C)
```

17. Single subscripts for multi-dimensional arrays is permitted in **EQUIVALENCE** statements.

18. Data initialization in a type specification statement is allowed.

```
DOUBLE PRECISION X/4.3D1/
```

19. Data initialization with hexadecimal constants is allowed.

```
INTEGER I/Z00000007/
```

20. Initializing character items with numeric data is permitted.

21. Hexadecimal and octal constants of the form `'abc'x` and `'567'o` are supported.

22. A character constant of the form `'abcdef'c` places a NUL character (CHAR(0)) at the end of the character string.

23. Hollerith constants can be used interchangeably with character constants.

```
CHARACTER*10 A, B
A = '1234567890'
B = 10H123456790
```

24. Several additional intrinsic functions are supported:

```
ALGAMA      ALLOCATED   BTEST       CDABS
CDCOS       CDSIN       CDEXP       CDSQRT
CDLOG       COTAN       DCMPLX      DCONJG
DCOTAN      DERF        DERFC       DFLOAT
DGAMMA      DIMAG       DLGAMA      DREAL
ERF         ERFC        GAMMA       HFIX
IAND        IBCHNG      IBCLR       IBSET
IEOR        IOR         ISHA        ISHC
ISHFT       ISHL        ISIZEOF     LENTRIM
LGAMMA      LOC         NOT         VOLATILE
```

25. The **LOC** intrinsic function returns the address of an expression.

26. The **ISIZEOF** intrinsic function returns the size of a structure name, the size of an array with a constant array declarator, or the size of a variable.

27. The **CHAR** intrinsic function is allowed in constant expressions.

28. The **ALLOCATE** and **DEALLOCATE** statements may be used to dynamically allocate and deallocate arrays.

29. The **ALLOCATED** intrinsic function may be used to determine if an allocatable array is allocated.

30. The following additional I/O specifiers for the **OPEN** statement are supported.

```
ACTION=
CARRIAGECONTROL=
RECORDTYPE=
RECL= is also allowed for files opened for
      sequential access
ACCESS='APPEND'
BLOCKSIZE=
SHARE=
```

31. The following additional I/O specifiers for the **INQUIRE** statement are supported.

```
ACTION=
CARRIAGECONTROL=
RECORDTYPE=
BLOCKSIZE=
SHARE=
```

32. In the **INQUIRE** statement, character data may also be returned in variables or array elements with a substring operation.

```
CHARACTER FN*20
INQUIRE( UNIT=1, FILE=FN(10:20) )
```

33. List-directed I/O is allowed with internal files.

34. No asterisk is required for list-directed I/O.

```
PRINT, X, Y
```

35. The **NAMELIST** statement is supported.

36. Non-character arrays are allowed as format specifiers.

37. The following format edit descriptors are allowed:

    ***Z*** for displaying data in hexadecimal format

    ***Ew.dDe*** same as Ew.dEe except D is used as exponentiation character

    ***$ or *** leave cursor at end of line

38. A repeat count is not required for the X edit descriptor (a repeat count of one is assumed).

39. Commas are optional between format edit descriptors.

```
100   FORMAT( 1X I5 )
```

40. It is possible to substring the return values of functions and statement functions.

```
CHARACTER*7 F, G
F() = '1234567'
PRINT *, F()(1:3), G()(4:7)
```

41. Functions may be invoked via the **CALL** statement. This allows the return value of functions to be ignored.

42. A **RETURN** statement is allowed in the main program.

43. Integer constants with more than 5 digits are allowed in the **STOP** and **PAUSE** statements.

```
PAUSE 123456

STOP 123456
```

44. Multiple assignment is allowed.

```
X = Y = Z = 0.0
```

45. The **.XOR.** operator is supported (equivalent to .NEQV.).

46. The **.AND.**, **.OR.**, **.NEQV.**, **.EQV.** and **.XOR.** operators may take integer arguments. They can be used to perform bit operations on integers.

47. Several additional program structure control statements are supported:

```
LOOP-ENDLOOP
UNTIL (can be used with WHILE and LOOP)
WHILE-ENDWILE
GUESS-ADMIT-ENDGUESS
ATENDDO-ENDATEND
ATEND
SELECT-ENDSELECT
DOWHILE-ENDDO
DO-ENDDO (no statement number)
REMOTEBLOCK-ENDBLOCK
EXECUTE
QUIT
EXIT
CYCLE
```

48. Block labels can be used to identify blocks of code.

```
LOOP : OUTER_LOOP
    <statements>
    LOOP : INNER_LOOP
        <statements>
        IF( X .GT.100 ) QUIT : OUTER_LOOP
        <statements>
    ENDLOOP
    <statements>
ENDLOOP
```

49. An integer expression in an **IF**, **ELSE IF**, **DO WHILE**, **WHILE** or **UNTIL** statement is allowed. The result of the integer expression is compared for inequality to the integer value 0.

## $

$ edit descriptor 228, 231

## .

.AND 288
.EQV 288
.NEQV 288
.OR 288
.XOR 288

## \

\ edit descriptor 228, 231

## A

A edit descriptor 236
ABS 251
access 217
  direct 217, 222
  sequential 217
ACCESS= 112
ACOS 260
actual argument 23, 44, 277
actual argument list 277
actual array declarator 164
adjustable array declarator 162
ADMIT 12, 83, 122, 206-208
AIMAG 255
AINT 250
ALGAMA 262
allocatable array declarator 163
ALLOCATE 13-14, 43-44, 163, 287
ALLOCATED 270, 287
ALOG 257
ALOG10 257
alternate return specifier 23, 274, 278
AMAX0 253
AMAX1 253
AMIN0 254
AMIN1 254
AMOD 252
ANINT 251
apostrophe edit descriptor 229
argument
  actual 44, 277
  dummy 44, 70, 77-78, 141, 277
arithmetic assignment statement 187
arithmetic constant expression 176
arithmetic expression 175
  factor 175
  primary 174
  term 175
  type of 177
arithmetic operators
  precedence 174
arithmetic relational expression 180
array
  assumed-size 44
  maximum size 160
array declarator 159
  actual 164
  adjustable 162
  allocatable 163
  assumed-size 162
  constant 162
  dummy 164
  maximum number of elements 160
array element 160
array elements
  maximum 160
ASA 223
ASIN 259
ASSIGN 16-17, 81, 118, 124, 131, 147, 188, 221
assignment statement
  arithmetic 187
  character 189
  extended 190
  logical 188
  statement label 188
assumed-size array 44
assumed-size array declarator 162
AT END 18, 57, 208-212
ATAN 260
ATAN2 260

## B

BACKSPACE 19-20, 61, 221
binary operator 173
blank common block 31
blank line 4
BLANK= 114, 231-232
BLOCK DATA 6, 21, 121, 243, 274
block data subprogram 39
BLOCKSIZE= 113
BN edit descriptor 231
BTEST 268
BZ edit descriptor 231

## C

CABS 251
CALL 23, 272-274, 277-278, 280-281, 288
carriage control 223
CASE 25, 66, 115, 122, 137-138, 201, 203-204
CASE DEFAULT 25, 115, 138, 201-204
case list 137
CCOS 258
CDABS 251
CDCOS 258
CDEXP 256
CDLOG 257
CDSIN 258
CDSQRT 256
CEXP 256
CHAR 179, 250, 287
CHARACTER 26, 28, 91, 285
character assignment statement 189
character constant 155
character constant expression 179
character expression 178
  primaries 178
character relational expression 180
character set
  FORTRAN 3
  processor 3
CLOG 257
CLOSE 29-30, 217, 219-220
CMPLX 249
collating sequence 180
colon edit descriptor 230
column major 72
comment line 4
comments
  end-of-line 4
COMMON 21, 31-32, 44, 73, 109, 160, 164, 241, 273-274, 278
common block
  blank 31
  named 21, 31
COMPLEX 33-35, 286
complex constant 154
complex constant expression 176
complex edit descriptor 236
COMPLEX*16 152, 286
CONJG 255
connection
  file 219
  unit 219
constant 153
  character 155
  complex 154
  double precision 154
  double precision complex 154
  hexadecimal 156
  hollerith 155
  integer 153
  logical 155
  octal 156
  real 153
  string 155
constant array declarator 162
constant expression 185
continuation line 4
CONTINUE 36, 48, 196
COS 258
COSH 261
COTAN 259
CSIN 258
CSQRT 256
CYCLE 37, 193, 195-199, 201, 210-211

## D

D edit descriptor 234-235
DABS 251
DACOS 260
DASIN 259
data 6-7, 21, 28, 32, 35, 39, 50, 52, 101, 105, 129, 156, 164, 215, 273-274, 278
data type
  summary 152

DATAN 260
DATAN2 260
DBLE 249
DCMPLX 249
DCONJD 255
DCOS 258
DCOSH 261
DCOTAN 259
DDIM 253
DEALLOCATE 14, 42, 44, 287
debug line 4
DEFAULT 25
DERF 262
DERFC 263
DEXP 256
DFLOAT 249
DGAMMA 262
DIM 253
DIMAG 255
DIMENSION 21, 44, 160, 164
dimension declarator 159
DINT 250
DIRECT 95
direct access 217, 222
disconnection 220
DLGAMA 262
DLOG 257
DLOG10 257
DMAX1 253
DMIN1 254
DMOD 252
DNINT 251
DO 6, 36-37, 45-48, 53, 59, 196-197, 210, 274
DO WHILE 37, 53, 197, 210, 289
dollar sign ($)
  in symbolic names 151
DOUBLE COMPLEX 49-50, 152, 286
DOUBLE PRECISION 51-52, 286
double precision complex constant 154
double precision complex constant expression 177
double precision constant 154
double precision constant expression 176
DPROD 253
DREAL 249
DSIGN 252
DSIN 258
DSINH 261
DSQRT 256
DTAN 259
DTANH 261
dummy argument 23, 44, 70, 77-78, 141, 277
  array 279
  asterisk 281
  dummy procedure 280
  of type CHARACTER 279
  variable 279
dummy argument list 277
dummy array declarator 164

## E

E edit descriptor 234
edit descriptor
  $ 231
  \ 231
  A 236
  apostrophe 229
  BN 231
  BZ 231
  colon 230
  H 229
  L 236
  numeric 232-236
    complex 236
    D 234-235
    E 234
    F 233
    I 232
  P 230
  positional 229
  positional T 229
  positional TL 229
  positional TR 229
  positional X 229
  repeatable 227-228
  S 230
  slash 230
  SP 230
  SS 230
  Z 237
ELSE 54-55, 63, 87, 193, 195
ELSE IF 54-55, 63, 87, 194-195, 289
END 4, 21, 32, 56, 132, 135, 212, 271, 276, 279
END AT END 57, 212
END BLOCK 58, 73, 131, 204-205
END DO 45, 53, 59, 74, 122, 196-198, 208-209
END GUESS 62, 83, 206-209
END IF 6, 54-56, 63, 87-88, 193-195, 208, 274
END LOOP 64, 74, 106, 122, 198-200, 207-211
END MAP 65, 107, 169
END SELECT 25, 66, 115, 137-138, 201-204, 208
END STRUCTURE 67, 140, 167
END UNION 68, 142, 169
END WHILE 69, 74, 122, 145, 199-200, 208-209

end-of-file 216
end-of-file specifier 223
end-of-line
  comments 4
END= 18, 211-212, 221, 223
ENDFILE 60-61, 217, 219, 221
endfile record 216
ENDGUESS 74, 209
ENTRY 6, 27, 70, 121, 244, 274-278, 281
entry point 274
EQUIVALENCE 21, 31-32, 71-72, 164, 169, 273-274, 278, 286
ERF 262
ERFC 263
ERR= 221-222
error specifier 222
EXECUTE 58, 73-74, 131, 204-205, 208-209
EXIT 74, 193, 195-199, 201, 209-210
EXP 256
exponent 153-154
expression
  arithmetic 175
  arithmetic constant 176
  complex constant 176
  double precision complex constant 177
  double precision constant 176
  evaluation of 184
  factor 175
  integer constant 176
  logical 184
  logical constant 184
  primary 174
  real constant 176
  relational 179
  term 175
extended assignment statement 190
extension
  $ edit descriptor 228
  \ edit descriptor 228
  E edit descriptor 227
  X edit descriptor 228
  Z edit descriptor 227
extensions
  language 285
  summary 285
EXTERNAL 75, 102, 245
external file 217
  access 217
  name 217
  properties 217
  record form 218
  record length 218
external function 271
external function name 70

## F

F edit descriptor 233
file 215, 217
  external 217
  internal 218
  name 217
file existence 217
FILE= 93, 95, 97-98, 112, 114
FLOAT 248
FMT= 124, 147, 221-222
FORM= 112
format 6, 16, 76, 118, 124, 147, 221, 225, 231
  field 228
  field width 228
  list-directed 118-119, 124-125, 147, 222
  namelist-directed 119, 124, 147, 222
  repeat specification 227
  see also 226
    edit descriptor 226
format specification 226
format specifier 221
format-directed I/O 238
FORMATTED 96
formatted input 125
formatted input/output 216
formatted record 216
FORTRAN 77
  language extensions 285
FROM 137, 201
function 6, 27, 77-78, 121, 153, 243-244, 271, 273-276, 278, 285
  external 271
  external name 70
  generic 153
  intrinsic 246
  statement 243

## G

GAMMA 262
generic function 153
  ABS 251
  ACOS 260
  AINT 250
  ANINT 251
  ASIN 259

ATAN 260
ATAN2 260
BTEST 268
CMPLX 249
CONJG 255
COS 258
COSH 261
COTAN 259
DBLE 249
DCMPLX 249
DIM 253
ERF 262
ERFC 263
EXP 256
GAMMA 262
IAND 264
IBCHNG 269
IBCLR 268
IBSET 268
IEOR 265
IMAG 255
INT 248
IOR 264
ISHA 266
ISHC 267
ISHFT 266
ISHL 265
LOG 257
LOG10 257
LSHIFT 269
MAX 253
MIN 254
MOD 252
NINT 251
NOT 265
REAL 248
RSHIFT 269
SIGN 252
SIN 258
SINH 261
SQRT 256
TAN 259
TANH 261
generic name 246
GO TO 16-17, 79-81, 131, 188-189, 196, 198, 201, 213
GUESS 12, 62, 83, 206

## H

H edit descriptor 229
hexadecimal constant 156
HFIX 248
hollerith constant 155

## I

I edit descriptor 232
I1ABS 251
I1AND 264
I1BCHNG 269
I1BCLR 268
I1BSET 268
I1BTEST 268
I1DIM 253
I1EOR 265
I1LSHIFT 269
I1MAX0 253
I1MIN0 254
I1MOD 252
I1NOT 265
I1OR 264
I1RSHIFT 269
I1SHA 266
I1SHC 267
I1SHFT 266
I1SHL 265
I1SIGN 252
I2ABS 251
I2AND 264
I2BCHNG 269
I2BCLR 268
I2BSET 268
I2BTEST 268
I2DIM 253
I2EOR 265
I2LSHIFT 269
I2MAX0 253
I2MIN0 254
I2MOD 252
I2NOT 265
I2OR 264
I2RSHIFT 269
I2SHA 266
I2SHC 267

I2SHFT 266
I2SHL 265
I2SIGN 252
IABS 251
IAND 264
IBCHNG 269
IBCLR 268
IBSET 268
ICHAR 250
IDIM 253
IDINT 248
IDNINT 251
IEOR 265
IF 6, 9, 54-55, 63, 83-84, 86-87, 193-195, 199, 208-210, 212-213, 274, 289
IFIX 248
IMPLICIT 6, 21, 26, 33, 49, 51, 89-91, 99, 103, 116, 127, 152-153, 285
IMPLICIT NONE 91, 285
implied-DO list 39
INCLUDE 6, 285
INDEX 255
initial line 4
input
  formatted 125
  list-directed 125
  unformatted 126
input/output
  formatted 216
  unformatted 216
INQUIRE 93, 97-98, 217, 219-220, 287
INT 248
INTEGER 91, 99-101
integer constant 153
integer constant expression 176
integer quotient 177
internal file 218
  definition 218
  position 218
  properties 218
  records 218
  restrictions 218
INTRINSIC 102, 247, 272-274, 278
intrinsic function 246
  ABS 251
  ACOS 260
  AIMAG 255
  AINT 250
  ALGAMA 262
  ALLOCATED 270
  ALOG 257
  ALOG10 257
  AMAX0 253
  AMAX1 253
  AMIN0 254
  AMIN1 254
  AMOD 252
  ANINT 251
  ASIN 259
  ATAN 260
  ATAN2 260
  BTEST 268
  CABS 251
  CCOS 258
  CDABS 251
  CDCOS 258
  CDEXP 256
  CDLOG 257
  CDSIN 258
  CDSQRT 256
  CEXP 256
  CHAR 179, 250
  CLOG 257
  CMPLX 249
  CONJG 255
  COS 258
  COSH 261
  COTAN 259
  CSIN 258
  CSQRT 256
  DABS 251
  DACOS 260
  DASIN 259
  DATAN 260
  DATAN2 260
  DBLE 249
  DCMPLX 249
  DCONJG 255
  DCOS 258
  DCOSH 261
  DCOTAN 259
  DDIM 253
  DERF 262
  DERFC 263
  DEXP 256
  DFLOAT 249
  DGAMMA 262
  DIM 253
  DIMAG 255
  DINT 250
  DLGAMA 262
  DLOG 257
  DLOG10 257
  DMAX1 253
  DMIN1 254
  DMOD 252
  DNINT 251
  DPROD 253

DREAL 249
DSIGN 252
DSIN 258
DSINH 261
DSQRT 256
DTAN 259
DTANH 261
ERF 262
ERFC 263
EXP 256
FLOAT 248
GAMMA 262
HFIX 248
I1ABS 251
I1AND 264
I1BCHNG 269
I1BCLR 268
I1BSET 268
I1BTEST 268
I1DIM 253
I1EOR 265
I1LSHIFT 269
I1MAX0 253
I1MIN0 254
I1MOD 252
I1NOT 265
I1OR 264
I1RSHIFT 269
I1SHA 266
I1SHC 267
I1SHFT 266
I1SHL 265
I1SIGN 252
I2ABS 251
I2AND 264
I2BCHNG 269
I2BCLR 268
I2BSET 268
I2BTEST 268
I2DIM 253
I2EOR 265
I2LSHIFT 269
I2MAX0 253
I2MIN0 254
I2MOD 252
I2NOT 265
I2OR 264
I2RSHIFT 269
I2SHA 266
I2SHC 267
I2SHFT 266
I2SHL 265
I2SIGN 252
IABS 251
IAND 264
IBCHNG 269
IBCLR 268
IBSET 268
ICHAR 250
IDIM 253
IDINT 248
IDNINT 251
IEOR 265
IFIX 248
INDEX 255
INT 248
IOR 264
ISHA 266
ISHC 267
ISHFT 266
ISHL 265
ISIGN 252
ISIZEOF 176, 270
LEN 254
LENTRIM 254
LGE 263
LGT 263
LLE 263
LLT 264
LOC 270
LSHIFT 269
MAX0 253
MAX1 253
MIN0 254
MIN1 254
MOD 252
NINT 251
NOT 265
REAL 248
RSHIFT 269
SIGN 252
SIN 258
SINH 261
SNGL 248
SQRT 256
TAN 259
TANH 261
VOLATILE 270
IOR 264
IOSTAT= 97-98, 221-222
ISHA 266
ISHC 267
ISHFT 266
ISHL 265
ISIGN 252
ISIZEOF 176, 270, 287

## K

keywords 151

## L

L edit descriptor 236
LEN 254
length specification 26, 34, 100, 104, 128
LENTRIM 254
LGE 263
LGT 263
line
  blank 4
  comment 4
  continuation 4
  debug 4
  initial 4
list-directed 219
list-directed format 118-119, 124-125, 147, 222
list-directed formatting 238
list-directed input 125
list-directed output 148
LLE 263
LLT 264
LOC 270, 287
LOCATION= 13-15, 43
LOGICAL 103-105
logical assignment statement 188
logical constant 155
logical constant expression 184
logical expression 184
  logical disjunct 183
  logical factor 183
  logical term 183
logical operator 181
LOOP 6, 37, 64, 106, 143, 198, 200, 210-211
lower case 5
lower case letters
  in symbolic names 151
LSHIFT 269

## M

main program 6, 243
MAP 65, 68, 107, 142, 169
MAP, END MAP 21
MAX0 253
MAX1 253
maximum
  number of array elements 160
  size of an array 160
MIN0 254
MIN1 254
MOD 252

## N

named common block 21, 31
NAMELIST 108-110, 119, 124, 147, 222, 240-242, 287
namelist-directed format 119, 124, 147, 222
NINT 251
nonrepeatable edit descriptors 228
NOT 265

## O

octal constant 156
OPEN 29, 95, 111-112, 114, 217, 219-220, 231-232, 287
operator
  binary 173
  precedence 174
  relational 179
  unary 173
order
  statement 7
OTHERWISE 25, 115, 138, 201
output
  list-directed 148

## P

P edit descriptor 230
PARAMETER 6-7, 21, 28, 91, 116, 118, 124, 147, 157, 184, 222, 273-274, 278
PAUSE 117, 288
positional edit descriptor 229
preconnection 219
PRINT 76, 108, 118-120, 131, 189, 205, 209-210, 215, 217, 223, 225, 229-230, 232-233, 235-237, 240
printing 223
PROGRAM 6, 121, 243, 274
program unit 4, 243

## Q

QUIT 12, 64, 83, 88, 106, 122, 193, 195-199, 201, 206-209

## R

READ 18, 57, 108, 110, 123-126, 211-212, 215, 218-219, 240, 242
REAL 127-129, 248
real constant 153
real constant expression 176
REC= 126, 149, 221-222
RECL= 96, 112, 114
record 21, 130, 140, 167, 216
  endfile 216
  fixed length 215
  form 218
  formatted 216
  length 218
  unformatted 216
  variable length 215
record specifier 222
RECORDTYPE= 113
relational expression 179
relational operator 179
REMOTE BLOCK 58, 73, 131, 204-205
repeatable edit descriptor 227
RETURN 32, 46, 56, 121, 132, 135, 213, 271, 274, 276-277, 279, 288
REWIND 61, 133-134, 219, 221
RSHIFT 269

## S

S edit descriptor 230
SAVE 21, 32, 121, 135, 164, 273-274, 276, 278
scale factor 230
SELECT 6, 25, 66, 115, 137, 201-203
sequence field 4
SEQUENTIAL 95
sequential access 217
SHARE= 113
SIGN 252
simple real constant 153
SIN 258
SINH 261
slash edit descriptor 230
SNGL 248
SP edit descriptor 230
specific name 246
specifier
  end-of-file 223
  error 222
  format 221
  record 222
  status 222
  unit 221
SQRT 256
SS edit descriptor 230
STAT= 13-14, 42
statement 4
statement function 6, 243
statement label 4
statement label assignment 188
statement order 7
status specifier 222
STATUS= 30, 112
STOP 46, 132, 139, 213, 276, 288
string constant 155
structure 67, 140, 152, 167
STRUCTURE, END STRUCTURE 21
subprogram 6, 243
  block data 39
subroutine 6, 23, 121, 141, 243-244, 273-278, 281
  name 70
subscript 160
subscript expression 161

subscript value 161
substring 165
substring expression 165
substring name 165
symbolic names 151
  dollar sign ($) in 151
  lower case letters in 151
  underscore (_) in 151

## T

T edit descriptor 229
TAN 259
TANH 261
TL edit descriptor 229
TR edit descriptor 229

## U

unary operator 173
underscore (_)
  in symbolic names 151
UNFORMATTED 96
unformatted input 126
unformatted input/output 216
unformatted record 216
UNION 68, 107, 142, 169
UNION, END UNION 21
unit 219
unit specifier 221
UNIT= 19, 29, 60, 93, 98, 111, 124, 147, 221-222
UNTIL 74, 106, 122, 143, 145, 200, 208-209, 289

## V

VOLATILE 144, 270

## W

WHILE 6, 9, 37, 69, 143, 145-146, 199-200, 210, 289
WRITE 108, 147-149, 215-218, 223, 240

## X

X edit descriptor 229

## Z

Z edit descriptor 237