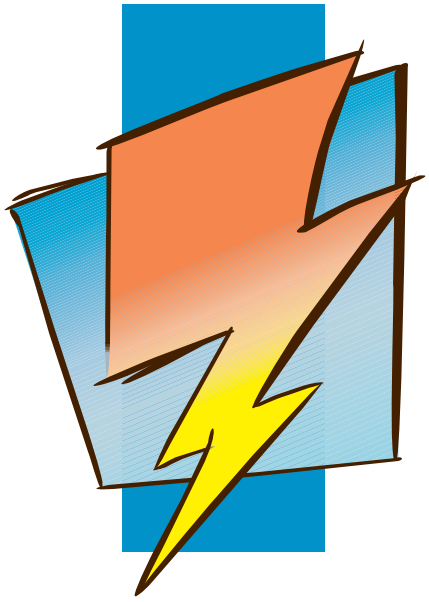
Open Watcom FORTRAN 77

User's Guide



Version 2.0

Open Watcom

Notice of Copyright

For more information please visit http://www.openwatcom.org/

Preface

The Open Watcom FORTRAN 77 Optimizing Compiler (Open Watcom F77) 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 this compiler includes the full language definition as well as significant extensions to the language. Open Watcom F77 evolved out of the demands of our users for a companion optimizing compiler to Open Watcom's WATFOR-77 "load-and-go" compiler.

The "load-and-go" approach to processing FORTRAN programs emphasizes fast compilation rates and quick placement into execution of FORTRAN applications. This type of compiler is used heavily during the debugging phase of the application. At this stage of application development, the "load-and-go" compiler optimizes the programmer's time ... not the program's time. However, once parts of the application have been thoroughly debugged, it may be advantageous to turn to a compiler which will optimize the execution time of the executable code.

Open Watcom F77 is a highly optimizing compiler based on the code generation technology that was developed for Open Watcom's highly-praised C and C++ optimizing compilers. Open Watcom F77 is a traditional compiler in the sense that it creates object files which must be linked into an executable program.

The *Open Watcom FORTRAN 77 User's Guide* describes how to use Open Watcom FORTRAN 77 with DOS, OS/2, Windows 3.x, Windows NT, and Windows 95.

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.

We would like to thank IMSL of Houston, Texas for providing us with copies of their Mathematics and Statistics libraries. The IMSL *Math Library* is a collection of subprograms for mathematical problem solving and the *Statistics Library* is a collection of subprograms for statistical analysis. The self test procedures provided with these libraries proved to be an immense help in testing Open Watcom F77 on the personal computer.

We also used the "FORTRAN Compiler Validation System, Version 2.0" to test the conformance of Open Watcom F77 with the full FORTRAN 77 language standard. This package is provided by the National Technical Information Service of the U.S. Department of Commerce in Springfield, Virginia. The validation system was developed by the Federal Software Testing Center.

If you find problems in the documentation or have some good suggestions, we would like to hear from you.

July, 1997.

Trademarks Used in this Manual

AutoCAD Development System is a trademark of Autodesk, Inc.

DOS/4G is a trademark of Tenberry Software, Inc.

IBM Developer's WorkFrame/2, Presentation Manager, and OS/2 are trademarks of International Business Machines Corp. IBM is a registered trademark of International Business Machines Corp.

Intel and Pentium are registered trademarks of Intel Corp.

Microsoft, Windows and Windows 95 are registered trademarks of Microsoft Corp. Windows NT is a trademark of Microsoft Corp.

NetWare, NetWare 386, and Novell are registered trademarks of Novell, Inc.

Phar Lap, 286|DOS-Extender, and 386|DOS-Extender are trademarks of Phar Lap Software, Inc.

QNX is a trademark of QNX Software Systems Ltd.

WATCOM is a trademark of Sybase, Inc. and its subsidiaries.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Open Watcom FORTRAN 77 User's Guide

1 About This Manual

This manual contains the following chapters:

Chapter 1 — "About This Manual".

This chapter provides an overview of the contents of this guide.

Chapter 2 — "Open Watcom FORTRAN 77 Compiler Options" on page 5.

This chapter also provides a summary and reference section for the valid compiler options.

Chapter 3 — "The Open Watcom FORTRAN 77 Compiler" on page 23.

This chapter describes how to compile an application from the command line, describes compiler environment variables, provides examples of command line use of the compiler, and and describes compiler diagnostics.

Chapter 4 — "The Open Watcom FORTRAN 77 Libraries" on page 29.

This chapter describes the Open Watcom FORTRAN 77 run-time libraries.

Chapter 5 — "Open Watcom FORTRAN 77 Compiler Directives" on page 33.

This chapter describes compiler directives including INCLUDE file processing.

Chapter 6 — "Open Watcom FORTRAN 77 File Handling" on page 39.

This chapter describes run-time file handling.

Chapter 7 — "The Open Watcom F77 Subprogram Library" on page 59.

This chapter describes subprograms available for special operations.

Chapter 8 — "16-bit Memory Models" on page 97.

This chapter describes the Open Watcom FORTRAN 77 memory models (including code and data models), the tiny memory model, the mixed memory model, linking applications for the various memory models, creating a tiny memory model application, and memory layout in an executable.

Chapter 9 — "16-bit Assembly Language Considerations" on page 101.

This chapter describes issues relating to 16-bit interfacing such as parameter passing conventions.

Chapter 10 — "16-bit Pragmas" on page 115.

This chapter describes the use of pragmas with the 16-bit compilers.

Chapter 11 — "32-bit Memory Models" on page 147.

This chapter describes the Open Watcom FORTRAN 77 memory models (including code and data models), the flat memory model, the mixed memory model, linking applications for the various memory models, and memory layout in an executable.

Chapter 12 — "32-bit Assembly Language Considerations" on page 151.

This chapter describes issues relating to 32-bit interfacing such as parameter passing conventions.

Chapter 13 — "32-bit Pragmas" on page 165.

This chapter describes the use of pragmas with the 32-bit compilers.

Appendix A. — "Use of Environment Variables" on page 199.

This appendix describes all the environment variables used by the compilers and related tools.

Appendix B. — "Open Watcom F77 Diagnostic Messages" on page 209.

This appendix lists all of the Open Watcom F77 diagnostic messages with an explanation for each.

2 Open Watcom FORTRAN 77 Compiler Options

Source files can be compiled using either the IDE, command-line compilers or IBM WorkFrame/2. This chapter describes all the compiler options that are available.

For information about compiling applications from the IDE, see the ***Open Watcom Graphical Tools User's Guide.***

For information about compiling applications from the command line, see the chapter entitled "The Open Watcom FORTRAN 77 Compiler" on page 23.

For information about creating applications using IBM WorkFrame/2, refer to IBM's ***OS/2 Programming Guide*** for more information.

2.1 Open Watcom F77 Options Summary

In this section, we present a terse summary of the Open Watcom F77 options. The next section describes these options in more detail. This summary is displayed on the screen by simply entering the "WFC" or "WFC386" command with no arguments.

Compiler options:	***Description:***
0	(16-bit only) assume 8088/8086 processor
1	(16-bit only) assume 188/186 processor
2	(16-bit only) assume 286 processor
3	assume 386 processor
4	assume 486 processor
5	assume Pentium processor
6	assume Pentium Pro processor
[NO]ALign	align COMMON segments
[NO]AUtomatic	all local variables on the stack
BD	(32-bit only) dynamic link library
BM	(32-bit only) multithread application
[NO]BOunds	generate subscript bounds checking code
BW	(32-bit only) default windowed application
[NO]CC	carriage control recognition requested for output devices such as the console
CHInese	Chinese character set
[NO]COde	constants in code segment
D1	include line # debugging information
D2	include full debugging information
[NO]DEBug	perform run-time checking
DEFine=<macro>	define macro
[NO]DEPendency	generate file dependencies
[NO]DEScriptor	pass character arguments using string descriptor
DIsk	write listing file to disk

DT=<size>	set data threshold
[NO]ERrorfile	generate an error file
[NO]EXPlicit	declare type of all symbols
[NO]EXtensions	issue extension messages
[NO]EZ	(32-bit only) Easy OMF-386 object files
FO=<obj_default>	set default object file name
[NO]FORmat	relax format type checking
FPC	generate calls to floating-point library
FPD	enable generation of Pentium FDIV bug check code
FPI	generate inline 80x87 instructions with emulation
FPI87	generate inline 80x87 instructions
FPR	floating-point backward compatibility
FP2	generate inline 80x87 instructions
FP3	generate inline 80387 instructions
FP5	optimize floating-point for Pentium
FP6	optimize floating-point for Pentium Pro
[NO]FSfloats	FS not fixed
[NO]GSfloats	GS not fixed
HC	Codeview debugging information
HD	DWARF debugging information
HW	Open Watcom debugging information
[NO]INCList	write content of INCLUDE files to listing
INCPath=[d:]path	[d:]path... path for INCLUDE files
[NO]IPromote	promote INTEGER*1 and INTEGER*2 arguments to INTEGER*4
Japanese	Japanese character set
KOrean	Korean character set
[NO]LFwithff	LF with FF
[NO]LIBinfo	include default library information in object file
[NO]LISt	generate a listing file
[NO]MAngle	mangle COMMON segment names
MC	(32-bit only) compact memory model
MF	(32-bit only) flat memory model
MH	(16-bit only) huge memory model
ML	large memory model
MM	medium memory model
MS	(32-bit only) small memory model
OB	(32-bit only) base pointer optimizations
OBP	branch prediction
OC	do not convert "call" followed by "ret" to "jmp"
OD	disable optimizations
ODO	DO-variables do not overflow
OF	always generate a stack frame
OH	enable repeated optimizations (longer compiles)
OI	generate statement functions in-line
OK	enable control flow prologues and epilogues
OL	perform loop optimizations
OL+	perform loop optimizations with loop unrolling
OM	generate floating-point 80x87 math instructions in-line
ON	numeric optimizations
OP	precision optimizations
OR	instruction scheduling
OS	optimize for space

OT	optimize for time
OX	equivalent to OBP, ODO, OI, OK, OL, OM, OR, and OT (16-bit) or OB, OBP, ODO, OI, OK, OL, OM, OR, and OT (32-bit)
PRint	write listing file to printer
[NO]Quiet	operate quietly
[NO]Reference	issue unreferenced warning
[NO]RESource	messages in resource file
[NO]SAve	SAVE local variables
[NO]SC	(32-bit only) stack calling convention
[NO]SEpcomma	allow comma separator in formatted input
[NO]SG	(32-bit only) automatic stack growing
[NO]SHort	set default INTEGER/LOGICAL size to 2/1 bytes
[NO]SR	save/restore segment registers
[NO]SSfloats	(16-bit only) SS is not default data segment
[NO]STack	generate stack checking code
[NO]SYntax	syntax check only
[NO]TErminal	messages to terminal
[NO]TRace	generate code for run-time traceback
TYpe	write listing file to terminal
[NO]WArnings	issue warning messages
[NO]WILd	relax wild branch checking
[NO]WIndows	(16-bit only) compile code for Windows
[NO]XFloat	extend floating-point precision
[NO]XLine	extend line length to 132

A summary of the option defaults follows:

0	16-bit only
5	32-bit only
ALign	
NOAUtomatic	
NOBOunds	
NOCC	
NOCOde	
NODEBug	
DEPendency	
DEScriptor	
DT=256	
ERrorfile	
NOEXPlicit	
NOEXtensions	
NOEZ	32-bit only
NOFORmat	
FPI	
FP2	16-bit only
FP3	32-bit only
NOFPD	
FSfloats	all but flat memory model
NOFSfloats	flat memory model only
GSfloats	
NOINCList	
NOIPromote	

NOLFwithff	
LIBinfo	
NOLISt	
NOMAngle	
ML	16-bit only
MF	32-bit only
NOQuiet	
Reference	
NORESource	
NOSAve	
NOSC	32-bit only
NOSEpcomma	
NOSG	32-bit only
NOSHort	
NOSR	
NOSSfloats	16-bit only
NOSTack	
NOSYntax	
TErminal	
NOTRace	
WArnings	
NOWILd	
NOWIndows	16-bit only
NOXFloat	
NOXLine	

2.2 Compiler Options

Compiler options may be entered in one of two places. They may be included in the options list of the command line or they may be included as comments of the form "C$option", "c$option", or "*$option" in the source input stream. The compiler recognizes these special comments as compiler directives.

Some options may only be specified in the options list of the command line. Unless otherwise stated, an option can appear on the command line only. We also indicate what the default is for an option or group of options.

When specifying options in the source file, it is possible to specify more than one option on a line. For example, the following source line tells Open Watcom F77 to not issue any warning or extension messages.

Example:

```
*$nowarn noext
```

Note that only the first option must contain the "*$", "C$", or "c$" prefix.

Short forms are indicated by upper case letters.

Option:	***Description:***
0	(16-bit only) Open Watcom F77 will make use of only 8088/8086 instructions in the generated object code. The resulting code will run on 8086 and all upward compatible processors. This is the default option for the 16-bit compiler.

1 (16-bit only) Open Watcom F77 will make use of 188/186 instructions in the generated object code whenever possible. The resulting code probably will not run on 8086 compatible processors but it will run on 186 and all upward compatible processors.

2 (16-bit only) Open Watcom F77 will make use of 286 instructions in the generated object code whenever possible. The resulting code probably will not run on 8086 or 186 compatible processors but it will run on 286 and all upward compatible processors.

3 Open Watcom F77 will assume a 386 processor and will generate instructions based on 386 instruction timings.

4 Open Watcom F77 will assume a 486 processor and will generate 386 instructions based on 486 instruction timings. The code is optimized for 486 processors rather than 386 processors.

5 Open Watcom F77 will assume a Pentium processor and will generate 386 instructions based on Pentium instruction timings. The code is optimized for Pentium processors rather than 386 processors. This is the default option for the 32-bit compiler.

6 Open Watcom F77 will assume a Pentium Pro processor and will generate 386 instructions based on Pentium Pro instruction timings. The code is optimized for Pentium Pro processors rather than 386 processors.

[NO]ALign The "align" option tells the compiler to allocate all COMMON blocks on paragraph boundaries (multiples of 16). If you do not want COMMON blocks to be aligned, specify "noalign". The default is "align".

[NO]AUtomatic
The "automatic" option tells the compiler to allocate all local variables, including arrays, on the stack. This is particularly useful for recursive functions or subroutines that require a new set of local variables to be allocated for each recursive invocation. Note that the "automatic" option may significantly increase the stack requirements of your application. You can increase your stack size by using the "STACK" option when you link your application.

BD (32-bit only, OS/2 and Windows NT only) This option causes the compiler to imbed the appropriate DLL library name in the object file and to include the appropriate DLL initialization code sequence when the application is linked.

BM (32-bit only, OS/2 and Windows NT only) This option causes the compiler to imbed the appropriate multi-thread library name in the object file.

[NO]BOunds The "bounds" option causes the generation of code that performs array subscript and character substring bounds checking. Note that this option may significantly reduce the performance of your application but is an excellent way to eliminate many programming errors. The default option is "nobounds".

BW (OS/2, Windows 3.x, and Windows NT only) This option causes the compiler to import a special symbol so that the default windowing library code is linked into your application.

[NO]CC The "cc" option specifies that the output to devices contains carriage control information that is to be interpreted appropriately for the output device (e.g., console device). ASA carriage control characters are converted to ASCII vertical spacing control characters.

Note that a blank carriage control character will automatically be generated for list-directed output and will be interpreted as a single-line spacing command.

CHInese This option is part of the national language support provided by Open Watcom F77. It instructs the compiler that the source code contains characters from the Traditional Chinese character set. This includes double-byte characters. This option enables the use of Chinese variable names. The compiler's run-time system will ensure that character strings are not split in the middle of a double-byte character when output spans record boundaries (as can happen in list-directed output).

[NO]COde The "code" option causes the code generator to place character and numeric constants in code segment. Data generated for FORMAT statements will also be placed in the code segment. The default option is "nocode".

D1 Line number information is included in the object file ("type 1 debugging information"). This option provides additional information to Open Watcom Debugger (at the expense of larger object files and executable files). Line numbers are handy when debugging your application with Open Watcom Debugger.

D2 In addition to line number information, local symbol and data type information is included in the object file ("type 2 debugging information"). Although global symbol information can be made available to Open Watcom Debugger through a Open Watcom Linker option, local symbol and typing information must be requested when the source file is compiled. This option provides additional information to Open Watcom Debugger (at the expense of larger object files and executable files). However, it will make the debugging chore somewhat easier.

[NO]DEBug The "debug" option causes the generation of run-time checking code. This includes subscript and substring bounds checking as well as code that allows a run-time traceback to be issued when an error occurs. The default option is "nodebug".

DEFine=<macro>

This option is equivalent to specifying the following "define" compiler directive.

```
*$define <macro>
```

The macro specified by the "define" option or compiler directive becomes defined. The definition status of the specified macro can be checked using the "ifdef", "ifndef", "elseifdef" or "elseifndef" compiler directives. This allows source code to be conditionally compiled depending on the definition status of the macro.

The macro `__i86__` is a special macro that is defined by the compiler and identifies the target as a 16-bit Intel 80x86 compatible environment.

The macro `__386__` is a special macro that is defined by the compiler and identifies the target as a 32-bit Intel 386 compatible environment.

The macro `__stack_conventions__` is a special macro that is defined by the 32-bit compiler when the "sc" compiler option is specified to indicate that stack calling conventions are to be used for code generation.

The macro `__fpi__` is a special macro that is defined by the compiler when one of the following floating-point options is specified: "fpi" or "fpi87".

[NO]DEPendency

The "dependency" option specifies that file dependencies are to be included in the object file. This is the default option. This option is used by the Open Watcom Integrated Development Environment to determine if an object file is up-to-date with respect to the source files used to build it. You can specify the "nodependency" option if you do not want file dependencies to be included in the object file.

[NO]DEScriptor

The "descriptor" option specifies that string descriptors are to be passed for character arguments. This is the default option. You can specify the "nodescriptor" option if you do not want string descriptors to be passed for character arguments. Instead, the pointer to the actual character data and the length will be passed as two arguments. The arguments for the length will be passed as additional arguments following the normal argument list. For character functions, the pointer to the data and the length of the character function will be passed as the first two arguments.

DIsk When this option is used in conjunction with the "list" option, the listing file is written to the current directory of the default disk. The listing file name will be the same as the source file name but the file extension will be `.lst`. By default, listing files are written to disk. The "disk" option will override any previously specified "type" or "print" option.

DT=<size> The "data threshold" option is used to set the minimum size for data objects to be included in the default data segment. Normally, all data objects smaller than 256 bytes in size are placed in the default data segment. When there is a large amount of static data, it is often useful to set the data threshold size so that all objects of the specified size or larger are placed into another segment. For example, the option:

```
/DT=100
```

causes all data objects of 100 bytes or more to be placed in a far data segment. The "data threshold" only applies to the large and huge memory models where there can be more than one data segment. The default data threshold value is 256.

[NO]ERrorfile This option is used to control whether error messages are output to a separate error file. The error file is a disk file of type `.err` and is produced if any diagnostic messages are issued by the compiler. Specifying "noerrorfile" prevents the creation of an error file. By default, an error file is created.

If an error file exists before compilation begins, it will be erased. If no diagnostic messages are produced then an error file will not be created even though the "errorfile" option is selected. This option has no effect on the inclusion of diagnostic messages in the source listing file or the production of diagnostic messages on the screen.

[NO]EXPlicit The "explicit" option requires the type of all symbols to be explicitly declared. An error message will be issued by the compiler if a symbol that does not appear in a type declaration statement is encountered. Specifying this option is equivalent to using the ***IMPLICIT NONE*** statement. By default, symbols do not have to be explicitly typed.

[NO]EXtensions

This option is used to control the printing of extension messages. This option may be specified on the command line or it may be placed anywhere in the source input stream. In a source file, the option appears as a comment line and takes the following form.

```
*$[NO]EXtensions
```

The "extensions" option enables the printing of extension messages, while "noextensions" disables the printing of these messages. By default, extension messages are not printed.

[NO]EZ (32-bit only) Open Watcom F77 will generate an object file in Phar Lap Easy OMF-386 (object module format) instead of the default Microsoft OMF. The default option is "noez".

FO=<obj_default>

By default, the object file name is constructed from the source file name. Using the "fo" option, the default object file drive, path, file name and extension can be specified.

Example:

```
C>wfc386 report /fo=d:\programs\obj\
```

A trailing "\" must be specified for directory names. If, for example, the option was specified as "/fo=d:\programs\obj" then the object file would be called `D:\PROGRAMS\OBJ.OBJ`.

A default extension must be preceded by a period (".").

Example:

```
C>wfc386 report /fo=d:\programs\obj\.dbo
```

[NO]FORmat The "format" option suppresses the run-time checking that ensures that the type of an input/output list item matches the format edit descriptor in a format specification. This allows an input/output list item of type INTEGER to be formatted using an F, E or D edit descriptor. It also allows an input/output list item of a floating-point type to be formatted using an I edit descriptor. Normally, this generates an error. The "format" option is particularly useful for applications that use integer arrays to store integer and floating-point data. The default option is "noformat".

FPC All floating-point arithmetic is done with calls to a floating-point emulation library. This option should be used when speed of floating-point emulation is favoured over code size.

FPI (16-bit only) Open Watcom F77 will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. Depending on which library the code is linked against, these instructions will be left as is or they will be replaced by special interrupt instructions. In the latter case, floating-point will be emulated if an 80x87 is not present. This is the default floating-point option if none is specified.

(32-bit only) Open Watcom F77 will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. When any module containing floating-point operations is compiled with the "fpi" option, coprocessor emulation software will be included in the application when it is linked. Thus, an 80x87 coprocessor need not be present at run-time. This is the default floating-point option if none is specified.

FPI87 (16-bit only) Open Watcom F77 will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. An 80x87 coprocessor must be present at run-time. If the "2" option is used in conjunction with this option, Open Watcom F77 will generate 287/387 compatible instructions; otherwise Open Watcom F77 will generate 8087 compatible instructions.

(32-bit only) Open Watcom F77 will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. When the "fpi87" option is used exclusively, coprocessor emulation software is not included in the application when it is linked. A 80x87 coprocessor must be present at run-time.

16-bit Notes:

1. When any module in an application is compiled with a particular "floating-point" option, then all modules must be compiled with the same option.

2. Different math libraries are provided for applications which have been compiled with a particular floating-point option. See the chapter entitled "The Open Watcom FORTRAN 77 Libraries" on page 29.

32-bit Notes:

1. When any module in an application is compiled with the "fpc" option, then all modules must be compiled with the "fpc" option.

2. When any module in an application is compiled with the "fpi" or "fpi87" option, then all modules must be compiled with one of these two options.

3. If you wish to have floating-point emulation software included in the application, you should select the "fpi" option. A 387 coprocessor need not be present at run-time.

4. Different math libraries are provided for applications which have been compiled with a particular floating-point option. See the chapter entitled "The Open Watcom FORTRAN 77 Libraries" on page 29.

FP2 Open Watcom F77 will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. For Open Watcom compilers generating 16-bit, this is the default. For 32-bit applications, use this option if you wish to support those few 386 systems that are equipped with an 80287 numeric data processor ("fp3" is the default for Open Watcom compilers generating 32-bit code). However, for 32-bit applications, the use of this option will reduce execution performance.

FP3 Open Watcom F77 will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. For 16-bit applications, the use of this option will limit the range of systems on which the application will run but there are execution performance improvements.

FP5 Open Watcom F77 will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. The sequence of floating-point instructions will be optimized for greatest possible performance on the Intel Pentium processor. For 16-bit applications, the use of this option will limit the range of systems on which the application will run but there are execution performance improvements.

FP6 Open Watcom F77 will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. The sequence of floating-point instructions will be optimized for greatest possible performance on the Intel Pentium Pro processor. For 16-bit applications, the use of this option will limit the range of

systems on which the application will run but there are execution performance improvements.

[NO]FPD A subtle problem was detected in the FDIV instruction of Intel's original Pentium CPU. In certain rare cases, the result of a floating-point divide could have less precision than it should. Contact Intel directly for more information on the issue.

As a result, the run-time system startup code has been modified to test for a faulty Pentium. If the FDIV instruction is found to be flawed, the low order bit of the run-time system variable `__chipbug` will be set.

If the FDIV instruction does not show the problem, the low order bit will be clear. If the Pentium FDIV flaw is a concern for your application, there are two approaches that you could take:

1. You may test the `__chipbug` variable in your code in all floating-point and memory models and take appropriate action (such as display a warning message or discontinue the application).

2. Alternately, you can use the "fpd" option when compiling your code. This option directs the compiler to generate additional code whenever an FDIV instruction is generated which tests the low order bit of `__chipbug` and, if on, calls the software workaround code in the math libraries. If the bit is off, an in-line FDIV instruction will be performed as before.

If you know that your application will never run on a defective Pentium CPU, or your analysis shows that the FDIV problem will not affect your results, you need not use the "fpd" option.

FPR Use this option if you want to generate floating-point instructions that will be compatible with version 9.0 or earlier of the compilers. For more information on floating-point conventions see the sections entitled "Using the 80x87 to Pass Arguments" on page 140 and "Using the 80x87 to Pass Arguments" on page 192.

[NO]FSfloats The "fsfloats" option enables the use of the FS segment register in the generated code. This is the default for all but the flat memory model. In the flat memory model, the default is "nofsfloats" (the FS segment register is not used in the generated code).

[NO]GSfloats The "gsfloats" option enables the use of the GS segment register in the generated code. This is the default. If you would like to prevent the use of the GS segment register in the the generated code, specify the "nogsfloats" option.

HC The type of debugging information that is to be included in the object file is "Codeview". The default type of debugging information is "Dwarf" (HD). If you wish to use the Microsoft Codeview debugger, then choose the "HC" option. When linking the application, you must also choose the appropriate Open Watcom Linker DEBUG directive. See the ***Open Watcom Linker User's Guide*** for more information.

HD The type of debugging information that is to be included in the object file is "Dwarf". This is the default type of debugging information. If you wish to use the Microsoft Codeview debugger, then choose the "HC" option. When linking the application, you must also choose the appropriate Open Watcom Linker DEBUG directive. See the ***Open Watcom Linker User's Guide*** for more information.

HW The type of debugging information that is to be included in the object file is "Open Watcom". The default type of debugging information is "Dwarf" (HD). If you wish to use the Microsoft Codeview debugger, then choose the "HC" option. When linking the application, you must also choose the appropriate Open Watcom Linker DEBUG directive. See the ***Open Watcom Linker User's Guide*** for more information.

[NO]INCList This option is used to control the listing of the contents of INCLUDE files to the listing file. The "inclist" option enables the listing of INCLUDE files, while "noinclist" disables the listing of these files. The default option is "noinclist".

INCPath=[d:]path

[d:]path... This option is used to specify directories that are to be searched for include files. Each path is separated from the previous by a semicolon (";"). These directories are searched in the order listed before those in the **FINCLUDE** environment variable.

[NO]IPromote The "ipromote" option causes the compiler to promote the INTEGER*1 and INTEGER*2 arguments of some INTEGER*4 intrinsics without issuing an error diagnostic. This allows code such as the following to be compiled without error:

Example:

```
INTEGER I*1, J*2
I = 1
J = 2
PRINT *, IDIM( I, J )
END
```

This works for the following intrinsic functions: ABS(), IABS(), DIM(), IDIM(), SIGN(), ISIGN(), MAX(), AMAX0(), MAX0(), MIN(), AMIN0(), and MIN0(). When the "ipromote" option is specified, all integer arguments that are passed to these functions are promoted to INTEGER*4.

Japanese This option is part of the national language support provided by Open Watcom F77. It instructs the compiler that the source code contains characters from the Japanese character set. This includes double-byte characters. This option enables the use of Japanese variable names. The compiler's run-time system will ensure that character strings are not split in the middle of a double-byte character when output spans record boundaries (as can happen in list-directed output).

KORean This option is part of the national language support provided by Open Watcom F77. It instructs the compiler that the source code contains characters from the Korean character set. This includes double-byte characters. This option enables the use of Korean variable names. The compiler's run-time system will ensure that character strings are not split in the middle of a double-byte character when output spans record boundaries (as can happen in list-directed output).

[NO]LFwithff This option is used to control whether a line-feed character (LF=CHAR(10)) is to be emitted before a form-feed character (FF=CHAR(12)) is emitted. This option applies to carriage control handling. Normally, the run-time system will emit only a form-feed character to cause a page eject when the ASA control character "1" is found in the first column of a record. The "lfwithff" option will cause the run-time system to emit a line-feed character and then a form-feed character.

The "lfwithff" option will have little effect on printers, but it will change the appearance of output to the screen by eliminating overwritten text when form-feed characters are not handled by the output device. The default option is "nolfwithff".

[NO]LIBinfo This option is used to control the inclusion of default library information in the object file. The "libinfo" option enables the inclusion of default library information, while "nolibinfo" disables the inclusion of this information. The default option is "libinfo".

[NO]LISt This option may be specified on the command line or it may be placed anywhere in the source input stream. On the command line, this option is used to control the creation of a listing file. The "list" option causes a listing file to be created while "nolist" requests that no listing file be created. The default option is "nolist".

In a source file, the option appears as a comment line and takes the following form.

```
*$[NO]LISt
```

Specifying *$LIST causes the source lines that follow this option to be listed in the source listing file while *$NOLIST disables the listing of the source lines that follow. This option cannot appear on the same source line with other options.

[NO]MAngle This option is used to alter COMMON block segment and class names.

Example:

```
REAL R, S
COMMON /BLK/ R, S
END
```

For a named COMMON block called "BLK", the default convention is to name the segment "BLK" and the class "BLK".

```
BLK             SEGMENT PARA COMMON USE32 'BLK'
```

When you use the "mangle" option, the segment is named "_COMMON_BLK" and the class is named "_COMMON_BLK_DATA".

```
_COMMON_BLK     SEGMENT PARA COMMON USE32 '_COMMON_BLK_DATA'
```

MC (32-bit only) The "compact" memory model (small code, big data) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147.

MF (32-bit only) The "flat" memory model (code and data up to 4 gigabytes) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147. This is the default memory model option.

MH (16-bit only) The "huge" memory model (big code, huge data) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147.

ML The "large" memory model (big code, big data) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147. This is the default 16-bit memory model option.

MM The "medium" memory model (big code, small data) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147.

MS (32-bit only) The "small" memory model (small code, small data) is selected. The various models supported by Open Watcom F77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147.

OB (32-bit only) This option allows the use of the ESP register as a base register to reference local variables and subprogram arguments in the generated code. This can reduce the size of the prologue/epilogue sequences generated by the compiler thus improving overall performance. Note that when this option is specified, the compiler will abort when there is not enough memory to optimize the subprogram. By default, the code generator uses more memory-efficient algorithms when a low-on-memory condition is detected.

OBP This option causes the code generator to try to order the blocks of code emitted such that the "expected" execution path (as determined by a set of simple heuristics) will be straight through, with other cases being handled by jumps to separate blocks of code "out of line". This will result in better cache utilization on the Pentium. If the heuristics do not apply to your code, it could result in a performance decrease.

OC This option may be used to disable the optimization where a "CALL" followed by a "RET" (return) is changed into a "JMP" (jump) instruction. This option is required if you wish to link an overlayed program using the Microsoft DOS Overlay Linker. The Microsoft DOS Overlay Linker will create overlay calls for a "CALL" instruction only. This option is not required when using the Open Watcom Linker. This option is not assumed by default.

OD Non-optimized code sequences are generated. The resulting code will be much easier to debug when using Open Watcom Debugger. By default, Open Watcom F77 will select "od" if "d2" is specified.

ODO Optimized DO-loop iteration code is generated. Caution should be exercised with the use of this option since the case of an iterating value overflowing is assumed to never occur. The following example should not be compiled with this option since the terminal value of `IX` wraps from a positive integer to a negative integer.

Example:

```
INTEGER*2 IX
DO IX=32766,32767
    .
    .
    .
ENDDO
```

The values of `IX` are 32766, 32767, -32768, -32767, ... since `IX` is INTEGER*2 (a 16-bit signed value) and it never exceeds the terminal value.

OF This option selects the generation of traceable stack frames for those functions that contain calls or require stack frame setup. To use Open Watcom's "Dynamic Overlay Manager" (DOS only), you must compile all modules using the "of" option. For near functions, the following function prologue sequence is generated.

16-bit:

```
push BP
mov  BP,SP
```

32-bit:

```
push EBP
mov  EBP,ESP
```

For far functions, the following function prologue sequence is generated.

16-bit:

```
inc  BP
push BP
mov  BP,SP
```

32-bit:

```
inc  EBP
push EBP
mov  EBP,ESP
```

The BP/EBP value on the stack will be even or odd depending on the code model. For 16-bit DOS systems, the Dynamic Overlay Manager uses this information to determine if the return address on the stack is a short address (16-bit offset) or long address (32-bit segment:offset). This option is not assumed by default.

OH This option enables repeated optimizations (which can result in longer compiles).

OI This option causes code for statement functions to be generated in-line.

OK This option enables flowing of register save (from prologue) down into the subprogram's flow graph.

OL Loop optimizations are performed. This includes moving loop-invariant expressions outside the loops. This option is not assumed by default.

OL+ Loop optimizations are performed including loop unrolling. This includes moving loop-invariant expressions outside the loops and can cause loops to be turned into straight-line code. This option is not assumed by default.

OM Generate inline 80x87 code for math functions like sin, cos, tan, etc. If this option is selected, it is the programmer's responsibility to make sure that arguments to these functions are within the range accepted by the `fsin, fcos,` etc. instructions since no run-time check is made.

If the "ot" option is also specified, the `exp` function is generated inline as well. This option is not assumed by default.

ON This option allows the compiler to perform certain numerical calculations in a more efficient manner. Consider the following example.

```
Z1 = X1 / Y
Z2 = X2 / Y
```

If the "on" option is specified, the code generator will generate code that is equivalent to the following.

```
T = 1 / Y
Z1 = X1 * T
Z2 = X2 * T
```

Since floating-point multiplication is more efficient that division, the code generator decided to first compute the reciprocal of Y and then multiply X1 and X2 by the reciprocal of Y.

Note that this optimization may produce less slightly different results since some, for certain values, precision is lost when computing the reciprocal. By using this option, you are indicating that you are willing to accept the loss in precision for the gain in performance.

OP By default, floating-point variables may be cached in 80x87 floating-point registers across statements when compiling with the "fpi" or "fpi87" options. Floating-point register temporaries use 64 bits of precision in the mantissa whereas single and double-precision variables use fewer bits of precision in the mantissa. The use of this option will force the result to be stored in memory after each FORTRAN statement is executed. This will produce less accurate but more predictable floating-point results. The code produced will also be less efficient when the "op" option is used.

Example:

```
XMAX = X + Y / Z
YMAX = XMAX + Q
```

When the "op" option is used in conjunction with the "fpi" or "fpi87" option, the compiler's code generator will update `XMAX` before proceeding with the second statement. In the second statement, the compiler will reload `XMAX` from memory rather than using the result of the previous statement. The effect of the "op" option on the resulting code can be seen by the increased code size statistic as well as through the use of the Open Watcom Disassembler. This option is not assumed by default.

OR This option enables reordering of instructions (instruction scheduling) to achieve better performance on pipelined architectures such as the 486. Selecting this option will make it slightly more difficult to debug because the assembly language instructions generated for a source statement may be intermixed with instructions generated for surrounding statements. This option is not assumed by default.

OS Space is favoured over time when generating code (smaller code but possibly slower execution). By default, Open Watcom F77 selects a balance between "space" and "time".

OT Time is favoured over space when generating code (faster execution but possibly larger code). By default, Open Watcom F77 selects a balance between "space" and "time".

OX Specifying the "ox" option is equivalent to specifying the "ob" (32-bit only), "obp", "odo", "oi", "ok", "ol", "om", "or", and "ot" options.

PRint This option is used to direct the listing file to the printer (device name "PRN") instead of the disk. The "print" option will override any previously specified "type" or "disk" option. The default is to create a listing file on the disk.

[NO]Quiet The "quiet" option suppresses the banner and summary information produced by the compiler. Only diagnostic messages will be displayed. The default option is "noquiet".

[NO]Reference

When the "reference" option is specified, warning messages will be issued for all unreferenced symbols. In a source file, the option appears as a comment line and takes the following form.

```
*$[NO]Reference
```

This option is most useful when used in an include file that is included by several subprograms. Consider an include file that defines many parameter constants and only a few are referenced by any one subprogram. If the first line of the include file is

```
*$noreference
```

and the last line is

```
*$reference
```

warning messages for all unused parameter constants in the include file would be suppressed. The default option is "reference".

[NO]RESource

The "resource" option specifies that the run-time error messages are contained as resource information in the executable file. All messages will be extracted from the resource area of the executable file when they are required; no messages will be linked with the application. The default option is "noresource".

[NO]SAve The "save" option is used to instruct Open Watcom F77 to "save" all local variables of subprograms. All local variables are treated as if they had appeared in FORTRAN 77 ***SAVE*** statements. By default, local variables are not saved unless named in a ***SAVE*** statement (i.e., "nosave" is the default option).

[NO]SC (32-bit only) If the "sc" option is used, Open Watcom F77 will pass all arguments on the stack. The resulting code will be larger than that which is generated for the register method of passing arguments. The default option is "nosc".

[NO]SEpcomma

The "sepcomma" option allows the comma (",") to be used as field separator in formatted input. Thus the following code would work with the input described.

Example:

```
      REAL R, S

      READ(5,21) R, S
      PRINT *, R, S
21    FORMAT(2F11.3)
      END
```

Normally the following input would result in a run-time error message.

```
0.79,0.21
```

[NO]SG (32-bit only) The "sg" option is useful for 32-bit OS/2 multi-threaded applications. It requests the code generator to emit a run-time call at the start of any function that has more than 4K bytes of automatic variables (variables located on the stack). Under 32-bit OS/2, the stack is grown automatically in 4K pages using the stack "guard page" mechanism. The stack consists of in-use committed pages topped off with a special guard page. A memory reference into the 4K guard page causes OS/2 to grow the stack by one 4K page and to add a new 4K guard page. This works fine when there is less than 4K of automatic variables in a function. When there is more than 4K of automatic data, the stack must be grown in an orderly fashion, 4K bytes at a time, until the stack has grown sufficiently to accommodate all the automatic variable storage requirements.

The "stack growth" run-time routine is called `__GRO`.

The default option is "nosg".

[NO]SHort The "short" option is used to instruct Open Watcom F77 to set the default INTEGER size to 2 bytes and the default LOGICAL size to 1 bytes. As required by the FORTRAN 77 language standard, the default INTEGER size is 4 bytes and the default LOGICAL size is 4 bytes. The default option is "noshort".

[NO]SR The "sr" option instructs Open Watcom F77 to generate subprogram prologue and epilogue sequences that save and restore any segment registers that are modified by the subprogram. Caution should be exercised when using this option. If the value of the segment register being restored matches the value of a segment that was freed within the subprogram, a general protection fault will occur in protected-mode environments. The default, "nosr", does not save and restore segment registers.

[NO]SSfloats (16-bit only) The "ssfloats" option specifies that the segment register SS does not necessarily point to the default data segment. The "ssfloats" option must be specified when compiling a module that is part of an OS/2 multi-threaded application or dynamic link library. By default, it is assumed that the SS segment register contains the segment address of the default data segment (i.e., "nossfloats").

[NO]STack If "stack" is specified, Open Watcom F77 will emit code at the beginning of every subprogram that will check for the "stack overflow" condition. By default, stack overflow checking is omitted from the generated code ("nostack").

[NO]SYntax If "syntax" is specified, Open Watcom F77 will check the source code only and omit the generation of object code. Syntax checking, type checking, and so on are performed as usual. By default, code is generated if there are no source code errors (i.e., "nosyntax" is the default).

[NO]TErminal The "noterminal" option may be used to suppress the display of diagnostic messages to the screen. By default, diagnostic messages are displayed.

[NO]TRace The "trace" option causes the generation of code that allows a traceback to be issued when an error occurs during the execution of your program. The default option is "notrace".

TYpe This option is used to direct the listing file to the terminal (device name "CON") instead of the disk. The "type" option will override any previously specified "print" or "disk" option. The default is to create a listing file on the disk.

[NO]WArnings
This option is used to control the printing of warning messages. By default, warning

messages are printed. This option may be specified on the command line or it may be placed anywhere in the source input stream. In a source file, the option appears as a comment line and takes the following form.

```
*$[NO]WArnings
```

The "warnings" option enables the printing of warning messages, while "nowarnings" disables the printing of these messages.

[NO]WILd The "wild" option suppresses the compile-time checking that normally causes an error to be issued when an attempt is made to transfer control into a block structure from outside the block structure and vice versa. For example, this option will allow a transfer of control into an IF-block from outside the IF-block (which is normally prohibited). The default option is "nowild".

Extreme caution should be exercised when using this option. For example, transfer of control into a DO-loop from outside the DO-loop can cause unpredictable results. This programming style is not encouraged by this option. The option has been made available so that existing programs that do not adhere to the branching restrictions imposed by the FORTRAN 77 standard (i.e. mainframe applications that are being ported to the PC environment), can be compiled by Open Watcom FORTRAN 77.

[NO]WIndows (16-bit only) The "windows" option causes the compiler to generate the prologue/epilogue code sequences necessary for use in Microsoft Windows applications. The default option is "nowindows".

[NO]XFloat The "xfloat" option specifies that all REAL variables are treated as if they had been declared as "DOUBLE PRECISION". This effectively increases the precision of REAL variables. Note that the "xfloat" option has implications on the alignment of variables in common blocks. The default option is "noxfloat".

[NO]Xline The "xline" option informs the Open Watcom F77 compiler to extend the last column of the statement portion of a line to column 132. The default is 72.

3 The Open Watcom FORTRAN 77 Compiler

This chapter describes the following topics:

- Command line syntax (see "Open Watcom FORTRAN 77 Command Line Format")
- Environment variables used by the compilers (see "WFC/WFC386 Environment Variables" on page 24)
- Examples of command line syntax (see "Open Watcom FORTRAN 77 Command Line Examples" on page 24)
- Interpreting diagnostic messages (see "Compiler Diagnostics" on page 25)
- Include file handling (see "Open Watcom FORTRAN 77 INCLUDE File Processing" on page 27)

3.1 Open Watcom FORTRAN 77 Command Line Format

The formal Open Watcom FORTRAN 77 command line syntax is shown below.

WFC [options] [d:][path]filename[.ext] [options]
WFC386 [options] [d:][path]filename[.ext] [options]

The square brackets [ ] denote items which are optional.

WFC is the name of the 16-bit Open Watcom F77 compiler.

WFC386 is the name of the 32-bit Open Watcom F77 compiler.

d: is an optional drive specification such as "A:", "B:", etc. If not specified, the default drive is assumed.

path is an optional path specification such as `\PROGRAMS\SRC\`. If not specified, the current directory is assumed.

filename is the file name of the file to be compiled.

ext is the file extension of the file to be compiled. If omitted, a file extension of "FOR" is assumed. If the period "." is specified but not the extension, the file is assumed to have no file extension.

options is a list of valid Open Watcom F77 options, each preceded by a slash ("/") or a dash ("–"). Certain options can include a "no" prefix to disable an option. Options may be specified in any order, with the rightmost option taking precedence over any conflicting options specified to its left.

3.2 WFC/WFC386 Environment Variables

The **WFC** environment variable can be used to specify commonly used WFC options. The **WFC386** environment variable can be used to specify commonly used WFC386 options. These options are processed before options specified on the command line.

Example:

```
C>set wfc=-d1 -ot
C>set wfc386=-d1 -ot
```

The above example defines the default options to be "d1" (include line number debugging information in the object file), and "ot" (favour time optimizations over size optimizations).

Whenever you wish to specify an option that requires the use of an "=" character, you can use the "#" character in its place. This is required by the syntax of the "SET" command.

Once a particular environment variable has been defined, those options listed become the default each time the associated compiler is used. The compiler command line can be used to override any options specified in the environment string.

These environment variables are not examined by the Open Watcom Compile and Link utilities. Since the Open Watcom Compile and Link utilities pass the relevant options found in their associated environment variables to the compiler command line, their environment variable options take precedence over the options specified in the environment variables associated with the compilers.

> ***Hint:*** If you are running DOS and you use the same compiler options all the time, you may find it handy to define the environment variable in your DOS system initialization file, `AUTOEXEC.BAT`.
>
> If you are running Windows NT, use the "System" icon in the **Control Panel** to define environment variables.
>
> If you are running OS/2 and you use the same compiler options all the time, you may find it handy to define the environment variable in your OS/2 system initialization file, `CONFIG.SYS`.

3.3 Open Watcom FORTRAN 77 Command Line Examples

The following are some examples of using Open Watcom FORTRAN 77 to compile FORTRAN 77 source programs.

Example 1:

```
C>wfc386 report /d1 /stack
```

The 32-bit Open Watcom F77 compiler processes `REPORT.FOR` producing an object file which contains source line number information. Stack overflow checking code is included in the object code.

Example 2:

```
C>wfc kwikdraw /2 /fpi87
```

The 16-bit Open Watcom F77 compiler processes `KWIKDRAW.FOR` producing object code for an Intel 286 system equipped with an Intel 287 numeric data processor (or any upward compatible 386/387, 486 or Intel Pentium system). While the choice of these options narrows the number of microcomputer systems where this code will execute, the resulting code will be highly optimized for this type of system.

Example 3:

```
C>wfc ..\source\modabs /d2
```

The 16-bit Open Watcom F77 compiler processes `..\SOURCE\MODABS.FOR` (a file in a directory which is adjacent to the current one). The object file is placed in the current directory. Included with the object code and data is information on local symbols and data types. The code generated is straight-forward, unoptimized code which can be readily debugged with Open Watcom Debugger.

Example 4:

```
C>wfc386 /mf calc
```

The 32-bit Open Watcom F77 compiler compiles `CALC.FOR` for the "flat" memory model. 32-bit memory models are described in the chapter entitled "32-bit Memory Models" on page 147. 32-bit argument passing conventions are described in the chapter entitled "32-bit Assembly Language Considerations" on page 151.

Example 5:

```
C>wfc386 kwikdraw /fpi87
```

The 32-bit Open Watcom F77 compiler processes `KWIKDRAW.FOR` producing object code for an Intel 386 system equipped with an Intel 80x87 numeric data processor.

Example 6:

```
C>set wfc=/short /d2 /fo#*.dbj
C>wfc ..\source\modabs
```

The options `/short`, `/d2` and `/fo=*.dbj` are established as defaults using the **WFC** environment variable. The 16-bit compiler processes `..\SOURCE\MODABS.FOR` (a file in a directory which is adjacent to the current one). The object file is placed in the current directory and it will have a default file extension of "DBJ". All INTEGER and LOGICAL variables will have a default type of INTEGER*2 and LOGICAL*1 unless explicitly typed as INTEGER*4 or LOGICAL*4. Source line number and local symbol information are included with the object file.

3.4 Compiler Diagnostics

If the Open Watcom F77 compiler prints diagnostic messages to the screen, it will also place a copy of these messages in a file in your current directory (unless the "noerrorfile" option is specified). The file will have the same file name as the source file and an extension of "err". The compiler issues three types of diagnostic messages, namely extensions, warnings and errors. An extension message indicates that you have used a feature which is supported by Open Watcom F77 but that is not part of the FORTRAN 77 language standard. A warning message indicates that the compiler has found a questionable problem in the source code (e.g., an unreachable statement, an unreferenced variable or statement number, etc.). A warning message does not prevent the production of an object file. An error message indicates that a problem is severe enough that it must be corrected before the compiler will produce an object file. The error file is a handy reference when you wish to correct the errors in the source file.

Just to illustrate the diagnostic features of Open Watcom F77, we will compile the following program called "DEMO1".

```
* This program demonstrates the following features of
* Open Watcom's FORTRAN 77 compiler:
*
*    1. Extensions to the FORTRAN 77 standard are flagged.
*
*    2. Compile time error diagnostics are extensive.  As many
*       errors as possible are diagnosed.
*
*    3. Warning messages are displayed where potential problems
*       can arise.
*
        PROGRAM MAIN
        DIMENSION A(10)
        DO I=1,10
            A(I) = I
            I = I + 1
        ENDLOOP
        GO TO 30
        J = J + 1
30      END
```

If we compile this program with the "extensions" option, the following output appears on the screen.

```
C>wfc demo1 /exten
WATCOM FORTRAN 77/16 Optimizing Compiler Version 2.0 1997/07/16 09:22:47
Copyright (c) 2002-2022 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
demo1.for(14): *EXT* DO-05 this DO loop form is not FORTRAN 77 standard
demo1.for(16): *ERR* DO-07 column 13, DO variable cannot be redefined
while DO loop is active
demo1.for(17): *ERR* SP-19 ENDLOOP statement does not match with DO
statement
demo1.for(19): *WRN* ST-08 this statement will never be executed due to
the preceding branch
demo1.for: 9 statements, 0 bytes, 1 extensions, 1 warnings, 2 errors
```

Here we see an example of the three types of messages, extension (*EXT*), error (*ERR*) and warning (*WRN*).

Diagnostic messages are also included in the listing file if the "list" option is specified. If we recompile our program and include the "list" option, a listing file will be created.

```
C>wfc demo1 /exten/list
  or
C>wfc386 demo1 /exten/list
```

The contents of the listing file are:

```
WATCOM FORTRAN 77/16 Optimizing Compiler Version 2.0 1997/07/16 09:22:47
Copyright (c) 2002-2022 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.

Options: list,disk,errorfile,extensions,reference,warnings,fpi,oc,of,om,
      os,ot,ox,ml,0,terminal,dependency,fsfloats,gsfloats,libinfo,dt=256,
      align

      1 * This program demonstrates the following features of
      2 * Open Watcom's FORTRAN 77 compiler:
      3 *
      4 *    1. Extensions to the FORTRAN 77 standard are flagged.
      5 *
      6 *    2. Compile time error diagnostics are extensive.  As many
      7 *       errors as possible are diagnosed.
      8 *
      9 *    3. Warning messages are displayed where potential problems
     10 *       can arise.
     11 *
     12         PROGRAM MAIN
     13         DIMENSION A(10)
     14         DO I=1,10
*EXT* DO-05 this DO loop form is not FORTRAN 77 standard
     15             A(I) = I
     16             I = I + 1
                    $
*ERR* DO-07 DO variable cannot be redefined while DO loop is active
     17         ENDLOOP
*ERR* SP-19 ENDLOOP statement does not match with DO statement
     18         GO TO 30
     19         J = J + 1
*WRN* ST-08 this statement will never be executed due to the preceding branch
     20 30      END


Code size (in bytes):              0  Number of errors:                  2
Compile time (in seconds):         0  Number of warnings:                1
Number of statements compiled:     9  Number of extensions:              1
```

As part of the diagnostic capability of Open Watcom F77, a "$" is often used to indicate the particular place in the source line where an error has been detected.

The complete list of Open Watcom F77 diagnostic messages is presented in the appendix entitled "Open Watcom F77 Diagnostic Messages" on page 209.

3.5 Open Watcom FORTRAN 77 INCLUDE File Processing

For information on include file processing, see the section entitled "The INCLUDE Compiler Directive" on page 34 in the chapter entitled "Open Watcom FORTRAN 77 Compiler Directives"

4 The Open Watcom FORTRAN 77 Libraries

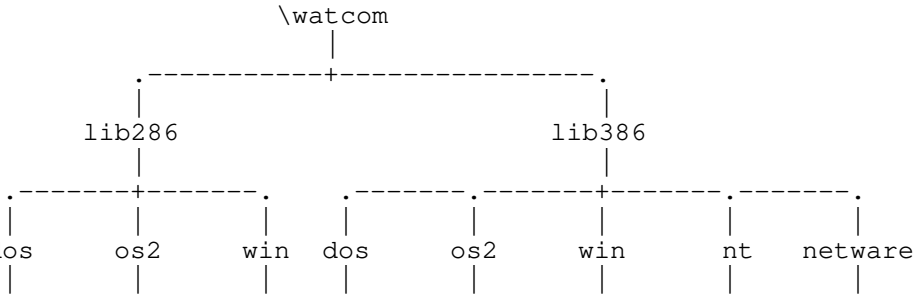
The Open Watcom FORTRAN 77 library routines (intrinsic functions) are described in the *Open Watcom FORTRAN 77 Language Reference* manual. Additional run-time routines are described in the chapter entitled "The Open Watcom F77 Subprogram Library" on page 59. Since Open Watcom FORTRAN 77 supports two major architectures, the 286 architecture (which includes the 8088) and the 386 architecture (which includes the 486 and Pentium processors), libraries are grouped under two major directories.

For the 286 architecture, the processor dependent libraries are placed under the `\WATCOM\LIB286` directory.

For the 386 architecture, the processor dependent libraries are placed under the `\WATCOM\LIB386` directory.

Since Open Watcom FORTRAN 77 also supports several operating systems, including DOS, Windows 3.x, Windows 95, Windows NT, OS/2 and NetWare, system-dependent libraries are grouped under different directories underneath the processor-dependent directories.

System	16-bit applications	32-bit applications
DOS	\WATCOM\LIB286\DOS	\WATCOM\LIB386\DOS
OS/2	\WATCOM\LIB286\OS2	\WATCOM\LIB386\OS2
Windows 3.x	\WATCOM\LIB286\WIN	\WATCOM\LIB386\WIN
Windows NT Windows 95		\WATCOM\LIB386\NT
NetWare		\WATCOM\LIB386\NETWARE



Due to the many code generation strategies possible in the 80x86 family of processors, a number of versions of the libraries are provided. You must use the libraries which coincide with the particular architecture, operating system, and code generation strategy or model that you have selected. For the type of code generation strategy or model that you intend to use, refer to the description of the "m?" memory model compiler option in the chapter entitled "Open Watcom FORTRAN 77 Compiler Options" on page 5. The various code models supported by Open Watcom FORTRAN 77 are described in the chapters entitled "16-bit Memory Models" on page 97 and "32-bit Memory Models" on page 147.

We have selected a simple naming convention for the libraries that are provided with Open Watcom FORTRAN 77. Letters are affixed to the file name to indicate the particular strategy with which the modules in the library have been compiled.

M denotes a version of the 16-bit Open Watcom FORTRAN 77 libraries which have been compiled for the "medium" memory model (big code, small data).

L denotes a version of the 16-bit Open Watcom FORTRAN 77 libraries which have been compiled for the "large" or "huge" memory models (big code, big data or huge data).

7 denotes a version of the Open Watcom FORTRAN 77 libraries which should be used when compiling with the "fpi" or "fpi87" option. Otherwise the libraries have been compiled using the "fpc" compiler option.

S denotes a version of the 32-bit Open Watcom FORTRAN 77 libraries which have been compiled using the "sc" option (stack calling conventions).

The 16-bit Open Watcom FORTRAN 77 libraries are listed below by directory.

Under \WATCOM\LIB286\DOS

```
FLIBM.LIB   (DOS medium model)
FLIB7M.LIB  (DOS medium model, in-line 80x87)
FLIBL.LIB   (DOS large/huge model)
FLIB7L.LIB  (DOS large/huge model, in-line 80x87)
CLIBM.LIB   (DOS i/o system medium model)
CLIBL.LIB   (DOS i/o system large/huge model)
GRAPH.LIB   (DOS graphics support)
```

Under \WATCOM\LIB286\WIN

```
FLIBM.LIB   (Windows medium model)
FLIB7M.LIB  (Windows medium model, in-line 80x87)
FLIBL.LIB   (Windows large/huge model)
FLIB7L.LIB  (Windows large/huge model, in-line 80x87)
CLIBM.LIB   (Windows i/o system medium model)
CLIBL.LIB   (Windows i/o system large/huge model)
WINDOWS.LIB (Windows API library)
```

Under \WATCOM\LIB286\OS2

```
FLIBM.LIB   (OS/2 medium model)
FLIB7M.LIB  (OS/2 medium model, in-line 80x87)
FLIBL.LIB   (OS/2 large/huge model)
FLIB7L.LIB  (OS/2 large/huge model, in-line 80x87)
CLIBM.LIB   (OS/2 i/o system medium model)
CLIBL.LIB   (OS/2 i/o system large/huge model)
DOSPMM.LIB  (Phar Lap 286 PM medium model)
DOSPML.LIB  (Phar Lap 286 PM large/huge model)
```

The 32-bit Open Watcom FORTRAN 77 libraries are listed below.

Under \WATCOM\LIB386\DOS

```
FLIB.LIB     (floating-point calls)
FLIB7.LIB    (in-line 80x87)
FLIBS.LIB    (floating-point calls, stack conventions)
FLIB7S.LIB   (in-line 80x87, stack conventions)
CLIB3R.LIB   (i/o system)
CLIB3S.LIB   (i/o system, stack conventions)
GRAPH.LIB    (DOS graphics support)
```

The graphics library GRAPH.LIB is independent of the argument passing conventions or floating-point model.

Under \WATCOM\LIB386\WIN

```
FLIB.LIB     (floating-point calls)
FLIB7.LIB    (in-line 80x87)
FLIBS.LIB    (floating-point calls, stack conventions)
FLIB7S.LIB   (in-line 80x87, stack conventions)
CLIB3R.LIB   (i/o system)
CLIB3S.LIB   (i/o system, stack conventions)
WIN386.LIB   (32-bit Windows API)
```

Under \WATCOM\LIB386\OS2

```
FLIB.LIB     (floating-point calls)
FLIB7.LIB    (in-line 80x87)
FLIBS.LIB    (floating-point calls, stack conventions)
FLIB7S.LIB   (in-line 80x87, stack conventions)
CLIB3R.LIB   (i/o system)
CLIB3S.LIB   (i/o system, stack conventions)
```

Under \WATCOM\LIB386\NT

```
FLIB.LIB     (floating-point calls)
FLIB7.LIB    (in-line 80x87)
FLIBS.LIB    (floating-point calls, stack conventions)
FLIB7S.LIB   (in-line 80x87, stack conventions)
CLIB3R.LIB   (i/o system)
CLIB3S.LIB   (i/o system, stack conventions)
```

4.1 Open Watcom FORTRAN 77 80x87 Emulator Libraries

One of the following libraries must be used if any of the modules of your application were compiled with the "fpi" option.

16-bit Libraries

```
NOEMU87.LIB
DOS\EMU87.LIB (DOS dependent)
WIN\EMU87.LIB (Windows dependent)
OS2\EMU87.LIB (OS/2 dependent)
```

32-bit Libraries

```
NOEMU387.LIB
DOS\EMU387.LIB (DOS dependent)
WIN\EMU387.LIB (Windows dependent)
OS2\EMU387.LIB (OS/2 dependent)
NT\EMU387.LIB (Windows NT dependent)
```

The "fpi" option causes an 80x87 numeric data processor emulator to be linked into your application. This emulator will decode and emulate 80x87 instructions when an 80x87 is not present in the system or if the environment variable **NO87** has been set (this variable is described below).

If you have compiled your application using the "fpi" option, you can also link with the 16-bit "noemu87.lib" or 32-bit "noemu387.lib" library, depending on which compiler you are using. However, your application will only run on a machine equipped with a 80x87 numeric data processor since the actual emulator is not linked into your application.

When the "fpi87" option is used exclusively, the emulator is not included. In this case, the application must be run on personal computer systems equipped with the numeric data processor.

4.2 The "NO87" Environment Variable

If you have a math coprocessor in your system but you wish to test a version of your application that will use floating-point emulation ("fpi" option) or simulation ("fpc" option), you can define the **NO87** environment variable. The 16-bit application must be compiled using the "fpc" (floating-point calls) option and linked with the appropriate `flib?.lib` library or the "fpi" option (default) and linked with the appropriate `flib7?.lib` and `emu87.lib` libraries. The 32-bit application must be compiled using the "fpc" (floating-point calls) option and linked with the appropriate `flib?.lib` library or the "fpi" option (default) and linked with the appropriate `flib7?.lib` and `emu387.lib` libraries. Using the "SET" command, define the environment variable as follows:

```
C>SET NO87=1
```

Now, when you run your application, the 80x87 will be ignored. To undefine the environment variable, enter the command:

```
C>SET NO87=
```

5 Open Watcom FORTRAN 77 Compiler Directives

5.1 Introduction

A number of compiler directives are available that allow, for example, conditional compilation of source code and the inclusion of source code from other files. A compiler directive is specified by placing a comment character ('c', 'C', or '*') in column one followed by a dollar sign ('$') immediately followed by the compiler directive. The following lists all of the compiler directives available with Open Watcom F77.

1. EJECT
2. INCLUDE
3. PRAGMA
4. DEFINE
5. UNDEFINE
6. IFDEF
7. IFNDEF
8. ENDIF
9. ELSE
10. ELIFDEF
11. ELIFNDEF

These compiler directives will be described in the following sections.

In addition to the above compiler directives, it is also possible to specify certain compiler options in the same way. The following lists these options.

1. [NO]EXTENSIONS
2. [NO]LIST
3. [NO]REFERENCE
4. [NO]WARNINGS

For more information on these options, see the the chapter entitled "Open Watcom FORTRAN 77 Compiler Options" on page 5.

5.2 The EJECT Compiler Directive

This compiler directive causes a form-feed to be generated in the listing file. The listing file is a carriage-control file that is created by the compiler when the "list" compiler option is specified. In the following example, a form-feed character will be generated immediately before the source for subroutine `sub2` and immediately before the source for subroutine `sub3`.

Example:

```
        subroutine sub1
        ! source code
        end
    *$eject
        subroutine sub2
        ! source code
        end
    *$eject
        subroutine sub3
        ! source code
        end
```

5.3 The INCLUDE Compiler Directive

The ***INCLUDE*** compiler directive or ***INCLUDE*** statement may be used to imbed source code into the file being compiled. Either form may be used.

Example:

```
    *$INCLUDE DOS.FI

              INCLUDE 'DOS.FI'
```

When the ***INCLUDE*** statement is used the name of the file must be placed inside single quotes (apostrophes). The file name may include drive, path, and file extension. The default file extension is `.for`.

It is not necessary to include the drive and path specifiers in the file specification when the file resides on a different drive or in a different directory. Open Watcom F77 provides a mechanism for looking up include files which may be located in various directories and disks of the computer system. When the drive and path are omitted from a file specification, Open Watcom F77 searches directories for include files in the following order.

1. First, the current directory is searched.
2. Secondly, each directory listed with the "INCPath" option is searched (in the order that they were specified).
3. Thirdly, each directory listed in the **FINCLUDE** environment variable is searched (in the order that they were specified).

The compiler will search the directories listed with the "INCPath" option or in the **FINCLUDE** environment variable in a manner analogous to that which used by the operating system when searching for programs by using the **PATH** environment variable.

The "INCPath" option takes the following form.

```
    -INCPath=[d:]path;[d:]path...
```

The "SET" command is used to define an **FINCLUDE** environment variable that contains a list of directories. A command of the form

```
    SET FINCLUDE=[d:]path;[d:]path...
```

is issued before running Open Watcom F77 the first time. The brackets indicate that the drive `d:` is optional and the ellipsis indicates that any number of paths may be specified.

We illustrate the use of the ***INCLUDE*** statement in the following example.

```
subroutine ClearScreen()
implicit none
include 'dos.fi'
integer VIDEO_CALL, SCROLL_UP
parameter (VIDEO_CALL=16, SCROLL_UP=6)
DS = ES = FS = GS = 0     ! for safety on 386 DOS extender
AH = SCROLL_UP            ! scroll up
AL = 0                    ! blank entire window
CX = 0                    ! set row,column of upper left
DX = 24*256 + 80          ! set row,column of lower right
BH = 7                    ! attribute "white on black"
call fintr( VIDEO_CALL, regs )
end
```

The third line of this subroutine contains an ***INCLUDE*** statement for the file `DOS.FI`. If the above source code is stored in the file `CLRSCR.FOR` in the current directory then we can issue the following commands to compile the application.

```
C>set finclude=c:\watcom\src\fortran\dos
C>wfc clsscr
```

In the above example, the "SET" command is used to define the **FINCLUDE** environment variable. It specifies that the `\WATCOM\SRC\FORTRAN\DOS` directory is to be searched for include files that cannot be located in the current directory and that have no drive or path specified. The advantage of the **FINCLUDE** environment variable is that drives and paths can be omitted from the INCLUDE file specifications in the source code. This allows the source code to be independent of the disk/directory structure of your computer system.

5.4 The PRAGMA Compiler Directive

This compiler directive is described in the chapters entitled "16-bit Pragmas" on page 115 and "32-bit Pragmas" on page 165.

5.5 The DEFINE Compiler Directive

The DEFINE compiler directive sets the definition status of a *macro* to defined. If a macro does not appear in a DEFINE directive, its definition status is undefined.

Example:

```
*$define debug
```

In the above example, the macro `debug` is defined.

The DEFINE compiler option can also be used to define a macro.

Example:
```
C>wfc /define=debug test
C>wfc386 /define=debug test
```

5.6 The UNDEFINE Compiler Directive

The UNDEFINE compiler directive sets the definition status of a *macro* to undefined.

Example:
```
*$undefine debug
```

In the above example, the definition status of the macro `debug` is set to undefined.

5.7 The IFDEF, IFNDEF and ENDIF Compiler Directive

The IFDEF and IFNDEF compiler directives check the definition status of a macro. If the macro appearing in an IFDEF directive is defined or the macro appearing in an IFNDEF directive is not defined, then all source code up to the corresponding ENDIF compiler directive will be compiled. Otherwise, it will be ignored.

In the following example, the FORTRAN 77 statements represented by `<debugging_statements>` will be compiled.

Example:
```
*$define debug
    ...
*$ifdef debug
    <debugging_statements>
*$endif
```

In the following example, the FORTRAN 77 statements represented by `<debugging_statements>` will not be compiled.

Example:
```
*$undefine debug
    ...
*$ifdef debug
    <debugging_statements>
*$endif
```

In the following example, the FORTRAN 77 statements represented by `<debugging_statements>` will be compiled.

Example:
```
*$undefine debug
    ...
*$ifndef debug
    <debugging statements>
*$endif
```

5.8 The ELSE Compiler Directive

The ELSE compiler directive must be preceded by an IFDEF, IFNDEF, ELSEIFDEF or ELSEIFNDEF compiler directive. If the condition of the preceding compiler directive was satisfied, then all source code between the ELSE compiler directive and the corresponding ENDIF compiler directive will be ignored. If the condition of the preceding compiler directive was not satisfied, then all source code between the ELSE compiler directive and the corresponding ENDIF compiler directive will be compiled.

In the following example, the FORTRAN 77 statements represented by `<debugging_level_2_statements>` will be compiled.

Example:

```
*$undefine debug_level_1
    ...
*$ifdef debug_level_1
    <debugging_level_1_statements>
*$else
    <debugging_level_2_statements>
*$endif
```

5.9 The ELSEIFDEF and ELSEIFNDEF Compiler Directive

The ELSEIFDEF and ELSEIFNDEF compiler directives must be preceded by an IFDEF, IFNDEF, ELSEIFDEF or ELSEIFNDEF compiler directive. If the condition of the preceding compiler directive was satisfied, then all source code between the ELSEIFDEF or ELSEIFNDEF compiler directive and the corresponding ENDIF compiler directive will be ignored. If the condition of the preceding compiler directive was not satisfied, then the definition status of the macro specified in the ELSEIFDEF or ELSEIFNDEF compiler directive is checked. If the macro appearing in the ELSEIFDEF compiler directive is defined, or the macro appearing in the ELSEIFNDEF compiler directive is not defined, then all source up to the next ELSEIFDEF, ELSEIFNDEF, ELSE or ENDIF compiler directive will be compiled.

In the following example, the FORTRAN 77 statements represented by `<debugging_level_2_statements>` will be compiled.

Example:

```
*$define debug_level_2
    ...
*$ifdef debug_level_1
    <debugging_level_1_statements>
*$elseifdef debug_level_2
    <debugging_level_2_statements>
*$endif
```

5.10 Debugging statements ("D" in Column 1)

If the character "D" or "d" appears in column 1, that line will be conditionally compiled depending on the definition status of the macro `__debug__`. Statements that contain a "D" or "d" in column 1 are called debugging statements. If the `__debug__` macro is defined, the line will be compiled; otherwise it will be ignored. The `__debug__` macro can be defined by using the DEFINE compiler directive or the "define" option. In the following example, the "define" option is used to force compilation of debugging statements.

Example:

```
C>wfc /def=__debug__ test
C>wfc386 /def=__debug__ test
```

5.11 General Notes About Compiler Directives

1. Compiler directives must not contain embedded blanks. The following is not a valid ENDIF compiler directive.

   Example:

   ```
   *$end if
   ```

2. Nesting is allowed up to a maximum of 16 levels.

   Example:

   ```
   *$ifdef sym1
       <statements>
   *$ifdef sym2
       <statements>
   *$endif
   *$endif
   ```

3. The macro `__i86__` is a special macro that is defined by the compiler and identifies the target as a 16-bit Intel 80x86 compatible environment.

4. The macro `__386__` is a special macro that is defined by the compiler and identifies the target as a 32-bit Intel 80386 compatible environment.

5. The macro `__stack_conventions__` is a special macro that is defined by the 32-bit compiler when stack conventions are used for code generation. Stack conventions are used when the "sc" or "3s" compiler options are specified.

6. The macro `__fpi__` is a special macro that is defined by the compiler when one of the following floating-point options is specified: "fpi" or "fpi87".

7. The macro `__debug__` is a special macro that can be used to conditionally compile debugging statements. A debugging statement is one that contains the character "D" or "d" in column one.

6 Open Watcom FORTRAN 77 File Handling

This chapter describes the file handling and naming conventions of Open Watcom F77. We discuss *files* and *devices* which are used to store, retrieve and display data. For example, a disk can be used to store a file of student marks. This file is accessible by other programs in order to produce summaries of the data such as marks reports. A device such as a printer can also be treated as if it were a file, although it is only useful for displaying data; an attempt to read information from this device is invalid.

In the following sections, we shall describe:

1. the techniques that Open Watcom F77 adopts for implementing *FORMATTED* and *UNFORMATTED* records and *SEQUENTIAL* and *DIRECT* access to these records,
2. the handling of "print" files,
3. file naming conventions,
4. logical file names,
5. the preconnection of files to units, and
6. special device support.

6.1 Record Access

Two types of record access are supported by Open Watcom F77:

Sequential Sequential access means that records in a file are accessed in order, starting with the first record in the file and proceeding to the last. Sequential access is permitted to records in both variable-length and fixed-length record files.

Direct Direct access means that records in a file are accessed in random order. For example, the fifth record could be accessed, then the second, and then the tenth. Direct access is permitted for fixed-length record files only.

The access method is described using the ***ACCESS=*** specifier of the FORTRAN ***OPEN*** statement. The default access is "SEQUENTIAL".

6.2 Record Format

There are two record formats, "FORMATTED" and "UNFORMATTED", which are supported by Open Watcom F77. The record format is described using the ***FORM=*** specifier of the FORTRAN ***OPEN*** statement. The default format is "FORMATTED" for files connected for sequential access and "UNFORMATTED" for files connected for direct access.

In describing these two formats, we also refer to the two methods of record access, "SEQUENTIAL" and "DIRECT", which are supported by Open Watcom F77.

6.2.1 FORMATTED Records

A *FORMATTED* record is one that contains an arbitrary number of ASCII characters. The end of a record is marked by an ASCII "LF" (line feed) character optionally preceded by an ASCII "CR" (carriage return) character. Thus this special sequence may not appear in the middle of a record.

FORMATTED records may vary in length. If all the records in the file have the same length then the records may be accessed both "sequentially" and "directly". If the records vary in length then it is only possible to access the records sequentially.

For direct access, the length of the records is specified by the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement. The specified length must not include the record separator since it does not form part of the record.

As an extension to the FORTRAN 77 language standard, Open Watcom F77 also supports the use of the ***RECL=*** specifier for sequential access. The maximum length of the records may be specified by the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement. The specified length must not include the record separator since it does not form part of the record. The length is used to allocate a record buffer for sequential access. If the record length is not specified, a default maximum length of 1024 characters is assumed.

6.2.2 UNFORMATTED Records

An *UNFORMATTED* record is one that contains an arbitrary number of binary storage units. The interpretation of the data in such a record depends on the FORTRAN program that is processing the record. An UNFORMATTED record may contain integers, real numbers, character strings, or any other type of FORTRAN data.

UNFORMATTED records may also vary in length. If all records in the file have the same length then the records may be accessed both "sequentially" and "directly". If the records vary in length then it is only possible to access the records sequentially.

When a file containing UNFORMATTED records is accessed sequentially, each record must begin and end with a descriptor that contains the length of the record. The length of the record is represented in 32 bits or 4 bytes (INTEGER*4). The UNFORMATTED records of a file which are written using sequential access will be automatically supplied with the appropriate length descriptors. When such a file is read, it is assumed that each record has the appropriate length descriptors.

Depending on the record length, the output produced by a single unformatted sequential ***WRITE*** statement may cause multiple records to be written. As previously mentioned, each record begins and ends with a length descriptor. The length descriptors for the first record contain the length of the record. The length descriptors for the remaining records contain the length of the record with the high bit (bit 31) set to one. In this way, an unformatted sequential file can be viewed as a number of logical records (a logical record corresponding to the output produced by a ***WRITE*** statement) with each logical record composed of a number of physical records. Files created in this way cannot be accessed directly unless each logical record is composed of a single physical record and each record is the same size.

As an extension to the FORTRAN 77 language standard, Open Watcom F77 also supports the use of the ***RECL=*** specifier for sequential access. The maximum length of the records may be specified by the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement. The specified length must not include the length descriptors since they do not form part of the record. The length is used to allocate a record buffer for

sequential access. If the record length is not specified, a default maximum length of 1024 characters is assumed.

When a file containing UNFORMATTED records is accessed directly, each record must be the same length. In this case, the length of the records is specified by the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement. If the file was originally created with sequential access then the specified length must include any length descriptors which form part of the record. In the direct access mode, no interpretation is placed on any of the data in an UNFORMATTED record and the programmer must account for any record length descriptors which may form part of the record.

Any records which are written using direct access must include record length descriptors if the file is to be accessed sequentially at a later time. As an alternative, you may specify `RECORDTYPE='VARIABLE'` in the FORTRAN ***OPEN*** statement. This specifier is an extension to the FORTRAN 77 language standard and will cause length descriptors to be generated automatically. In this case, the record length should not include the record length descriptors.

6.2.3 Files with no Record Structure

Certain files, for example a file created by a program written in another language, do not have any internal record structure that matches any of the record structures supported by Open Watcom F77. These files are simply streams of data. There are two ways in which these files can be processed.

1. You can use unformatted direct access. In this case, the value specified by the ***RECL=*** specifier in the ***OPEN*** statement determines the amount of data read or written by a ***READ*** or ***WRITE*** statement.

2. Alternatively, you can use unformatted sequential access. In this case, the amount of data read or written to the file is determined by the items in the input/output list of the ***READ*** or ***WRITE*** statement. When using unformatted sequential access, you must specify `RECORDTYPE='FIXED'` to indicate that no record boundaries are present. Otherwise, the default value of `'VARIABLE'` will be used.

6.3 Attributes of Files

The file system does not retain any information on the contents of a file. Unlike more sophisticated file systems, it cannot report whether a file consists of fixed-length or variable-length records, how records are delimited in a file, the maximum length of the records, etc. Therefore, we have provided a mechanism which will allow you to specify additional information about a file. This mechanism should be used when the default assumptions about records in a file are not true for the file in question.

The ***RECORDTYPE=*** specifier of the FORTRAN ***OPEN*** statement can be used to specify additional information about the type of records in the file. This specifier is an extension to the FORTRAN 77 language standard.

The ***RECL=*** specifier of the FORTRAN ***OPEN*** statement can be used to specify additional information about the length of records in the file. When used with sequential access, this specifier is an extension to the FORTRAN 77 language standard.

The ***CARRIAGECONTROL=*** specifier of the FORTRAN ***OPEN*** statement can be used to specify additional information about the handling of ASA carriage control characters for an output file. This specifier is an extension to the FORTRAN 77 language standard.

The ***BLOCKSIZE=*** specifier of the FORTRAN ***OPEN*** statement can be used to specify the size of the internal input/output buffer. A buffer reduces the number of system input/output calls during input/output to a particular file and hence improves the overall performance of a program. The default buffer size is 4K. This specifier is an extension to the FORTRAN 77 language standard.

The following sections describe the attributes of records supported by the Open Watcom F77 run-time system.

6.3.1 Record Type

The ***RECORDTYPE=*** specifier of the FORTRAN ***OPEN*** statement can be used to specify additional information about the type of records in the file. This specifier is an extension to the FORTRAN 77 language standard. The following types may be specified.

```
RECORDTYPE='TEXT'
RECORDTYPE='VARIABLE'
RECORDTYPE='FIXED'
```

TEXT indicates that the file contains variable-length or fixed-length records of ASCII characters separated by an ASCII "LF" (line feed) character optionally preceded with an ASCII "CR" (carriage return) character. By default, the Open Watcom F77 run-time system assumes that ***FORMATTED*** records are of ***TEXT*** format in both the *sequential* and *direct* access modes.

By default, the Open Watcom F77 run-time system uses variable-length record ***TEXT*** files to implement ***FORMATTED*** records in the *sequential* access mode. Of course, all records may be the same length. The record separator is not included in calculating the maximum size of records in the file.

By default, the Open Watcom F77 run-time system uses fixed-length record ***TEXT*** files to implement ***FORMATTED*** records in the *direct* access mode. Each record must be the same length. The record separator is not included in calculating the size of records in the file.

VARIABLE indicates that the file contains variable-length or fixed-length records in which special descriptors are employed to describe the length of each record. The length of each record is contained in a doubleword (INTEGER*4 item) at the beginning and end of the record. These descriptors determine the bounds of the records.

By default, the Open Watcom F77 run-time system uses ***VARIABLE*** format files to implement ***UNFORMATTED*** records in the *sequential* access mode. The length descriptors are required to support the FORTRAN ***BACKSPACE*** statement since no other method exists for determining the bounds of a variable-length unformatted record in a file.

FIXED indicates that the file contains no extra information that determines the record structure. If the file is a direct access file, the value specified by the ***RECL=*** specifier determines the size of each record in the file.

By default, the Open Watcom F77 run-time system uses ***FIXED*** format files to implement ***UNFORMATTED*** records in the *direct* access mode.

If you specify ***FIXED*** with an unformatted sequential file, the size of the records is determined by the items in the input/output list.

6.3.2 Record Size

When access is *direct,* the record length must be specified in the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement.

```
OPEN( UNIT=1, FILE='TEST.DAT', ACCESS='DIRECT', RECL=size, ... )
```

As an extension to the FORTRAN 77 language standard, the record length may also be specified when the access is *sequential.* This should be done whenever access is "sequential" and the maximum record length is greater than the default.

```
OPEN( UNIT=1, FILE='TEST.DAT', ACCESS='SEQUENTIAL', RECL=size, ... )
```

The record length specified by *size* should not include record separators such as CR and LF, nor should it include record length descriptors when sequentially accessing a file containing unformatted records. However, for all files, records longer than the size specified will be truncated. The default record size is 1024. The maximum record size is 65535 for the 16-bit run-time system. Since record buffers are allocated in the dynamic storage region, the size will be restricted to the amount of dynamic storage available.

6.3.3 Print File Attributes

When the first character of each record written to a file will contain an ASA (American Standards Association) carriage control character, the ***CARRIAGECONTROL=*** specifier of the FORTRAN ***OPEN*** statement should be used. This specifier is an extension to the FORTRAN 77 language standard. The ASA character is used for vertical spacing control. The valid characters and their interpretation are:

"1" Advance to Top of Page

"+" Advance Zero Lines (overprint)

" " Advance 1 Line

"0" Advance 2 Lines

"-" Advance 3 Lines

If `CARRIAGECONTROL='YES'` is specified then the Open Watcom F77 run-time system will automatically allocate an extra character at the beginning of a record for the vertical spacing control.

Upon transmitting a record to a file which has the "carriage" attribute, the Open Watcom F77 run-time system will substitute the appropriate ASCII carriage control characters as follows.

"1" Substitute a FF (form feed) for the "1".

"+" Append only a CR (carriage return) to the previous record.

" " Throw away the blank character.

"0" Substitute CR (carriage return) and LF (line feed) for the "0".

"-" Substitute two pairs of CR and LF for the "-".

Any other character in this position will be treated as if a blank character had been found (i.e., it will be discarded).

If the "carriage" attribute is not specified for a file then records will be written to the file without placing any interpretation on the first character position of the record.

6.3.4 Input/Output Buffer Size

The ***BLOCKSIZE=*** specifier is optional. However if you would like to change the default buffer size of 16K for 32-bit applications and 4K for 16-bit applications, you must specify the buffer size in the ***BLOCKSIZE=*** specifier of the ***OPEN*** statement.

```
OPEN( UNIT=1, FILE='TEST.DAT', BLOCKSIZE=1024, ... )
```

6.3.5 File Sharing

On systems that support multi-tasking or networking, it is possible for for a file to be accessed simultaneously by more that one process. There are two specifiers in the ***OPEN*** statement that can be used to control the way in which files are shared between processes.

The ***ACTION=*** specifier indicates the way in which the file is initially accessed. That is, the way in which the first process to open the file accesses the file. The values allowed for the ***ACTION=*** specifier are the following.

'READ' the file is opened for read-only access

'WRITE' the file is opened for write-only access

'READWRITE' the file is opened for both read and write access

The ***SHARE=*** specifier can be used to indicate the manner in which subsequent processes are allowed to access the file while the file is open. The values allowed for the ***SHARE=*** specifier are the following.

'COMPAT' no other process may open the file

'DENYRW' other processes are denied read and write access

'DENYWR' other process are denied write access (allowed read-only access)

'DENYRD' other process are denied read access (allowed write-only access)

'DENYNONE' other processes are allowed read and write access

Let us consider the following scenario. Suppose you want several processes to read a file and prevent any process that is reading the file from changing its contents. We first must establish the method of access for the first process that opens the file. In this case, we want read-only access so the `ACTION='READ'` specifier must be used. Next, we must establish the method of access for subsequent processes. In our example, we do not want any process to make changes to the file. Therefore, we use the `SHARE='DENYWR'` specifier. The file would be opened using the following ***OPEN*** statement.

```
OPEN( UNIT=1, FILE='TEST.DAT', ACTION='READ', SHARE='DENYWR', ... )
```

6.4 File Names in the FAT File System

The FAT file system is supported by DOS and OS/2. OS/2 also supports the High Performance File System (HPFS) which will be discussed in a later section. File naming conventions are used to form file designations in a given file system. The file designation for a FAT file system has the following form.

```
[d:][path]filename[.ext]
```

[] The square brackets denote items which are optional.

d: is the *drive name*. If omitted, the default drive is assumed.

Examples of drive names are: `a:`, `b:`, `c:`, and `d:`.

path is called a "path" specification. The path may be used to refer to files that are stored in sub-directories of the disk. The complete file specification (including drive, path and file name) cannot exceed 143 characters.

Some examples of path specifications are:

```
\plot\
\bench\tools\
\fortran\pgms\
```

Your operating system manuals can tell you more about directories: how to create them, how to store files in them, how to specify a path, etc.

filename is the main part of the file's name. The filename can contain up to 8 characters. If more than 8 characters are used, only the first 8 are meaningful. For example, "COUNTRIES" and "COUNTRIE" are treated as the same name for a file.

ext is an optional *extension* consisting of 1 to 3 characters (e.g., DOC). If an extension is specified, it is separated from the filename by a period. Extensions are normally used to indicate the type of information stored in the file. For example, a file extension of `for` is a common convention for FORTRAN programs.

> ***Note:*** The file specification is case insensitive in that upper and lower case letters can be used interchangeably.

6.4.1 Special DOS Device Names

Certain file names are reserved for devices. These special device names are:

```
CON   the console (or terminal)
AUX   the serial port
COM1  another name for the serial port
COM2  a second serial port
PRN   the parallel printer
LPT1  another name for the printer
LPT2  a second parallel printer
LPT3  a third parallel printer
NUL   nonexistent device
```

When using one of these special device names, no other part of the file designation should be specified. A common mistake is to attempt to create a disk file such as `PRN.DAT` and attempt to write records to it. If you do not have a parallel printer attached to your PC, there may be a long delay before the output operation times out.

6.4.2 Examples of FAT File Specifications

The following are some examples of valid file specifications.

1. The following file designation refers to a file in the current directory of the default disk.

```
OPEN( UNIT=1, FILE='DATA.FIL', ... )
```

2. The following file designation refers to a print file in the current directory of drive `c:`. ASA carriage control characters will be converted to the appropriate ASCII control codes.

```
OPEN( UNIT=2, FILE='c:report.lst',
      CARRIAGECONTROL='YES', ... )
```

3. The file specification below indicates that the file is to have fixed format records of length 80.

```
OPEN( UNIT=3, FILE='final.tst',
      RECL=80, RECORDTYPE='FIXED', ... )
```

4. The file specification below indicates that the file is to have variable format records of maximum length 145.

```
OPEN( UNIT=4, FILE='term.rpt',
      RECL=145, RECORDTYPE='VARIABLE', ... )
```

5. The file designation below indicates that the file resides in the `records` directory of drive `b:`.

```
OPEN( UNIT=5, FILE='b:\records\customers.dat', ... )
```

   Note that the trailing "S" in the file name will be ignored. Thus the following designation is equivalent.

```
OPEN( UNIT=5, FILE='b:\records\customer.dat', ... )
```

6. The file designation below refers to the second serial port.

```
OPEN( UNIT=6, FILE='com2', ... )
```

7. The file designation below refers to a second parallel printer.

```
OPEN( UNIT=7, FILE='lpt2', ... )
```

6.5 File Names in the High Performance File System

OS/2, in addition to supporting the FAT file system, also supports the High Performance File System (HPFS). The rules for forming file names in the High Performance File System are different from those used to form file names in the FAT file system. In HPFS, file names and directory names can be up to 254 characters in length. However, the complete path (including drive, directories and file name) cannot exceed 259 characters. The period is a valid file name character and can appear in a file name or directory name as many times as required; HPFS file names do not require file extensions as in the FAT file system. However, many applications still use the period to denote file extensions.

The HPFS preserves case in file names only in directory listings but ignores case in file searches and other system operations. For example, a directory cannot have more than one file whose names differ only in case.

6.5.1 Special OS/2 Device Names

The OS/2 operating system has reserved certain file names for character devices. These special device names are:

```
CLOCK$          Clock
COM1            First serial port
COM2            Second serial port
COM3            Third serial port
COM4            Fourth serial port
CON             Console keyboard and screen
KBD$            Keyboard
LPT1            First parallel printer
LPT2            Second parallel printer
LPT3            Third parallel printer
MOUSE$          Mouse
NUL             Nonexistent (dummy) device
POINTER$        Pointer draw device (mouse screen support)
PRN             The default printer, usually LPT1
SCREEN$         Screen
```

When using one of these special device names, no other part of the file designation should be specified.

6.5.2 Examples of HPFS File Specifications

The following are some examples of valid file specifications.

1. The following file designation refers to a file in the current directory of the default disk.

```
OPEN( UNIT=1, FILE='DATA.FIL', ... )
```

2. The following file designation refers to a print file in the current directory of drive `c:`. ASA carriage control characters will be converted to the appropriate ASCII control codes.

```
OPEN( UNIT=2, FILE='c:report.lst',
      CARRIAGECONTROL='YES', ... )
```

3. The file specification below indicates that the file is to have fixed format records of length 80.

```
OPEN( UNIT=3, FILE='final.tst',
      RECL=80, RECORDTYPE='FIXED', ... )
```

4. The file specification below indicates that the file is to have variable format records of maximum length 145.

```
OPEN( UNIT=4, FILE='term.rpt',
      RECL=145, RECORDTYPE='VARIABLE', ... )
```

5. The file designation below indicates that the file resides in the `records` directory of drive `b:`.

```
OPEN( UNIT=5, FILE='b:\records\customers.dat', ... )
```

   Note that the trailing "S" in the file name is not ignored as is the case in a FAT file system.

6. The file designation below refers to the second serial port.

```
OPEN( UNIT=6, FILE='com2', ... )
```

7. The file designation below refers to a second parallel printer.

```
OPEN( UNIT=7, FILE='lpt2', ... )
```

6.6 Establishing Connections Between Units and Files

Using Open Watcom F77, FORTRAN unit numbers may range from 0 to 999. Input/output statements such as ***READ*** and ***WRITE*** refer to files by a unit number. All input/output statements except ***OPEN***, ***CLOSE***, and ***INQUIRE*** must refer to a unit that is connected to a file. The Open Watcom F77 run-time system automatically establishes the connection of a unit to a file if no connection previously existed. Any connection between a unit and a file that is established before execution begins is called a preconnection.

The Open Watcom F77 run-time system defines a preconnection of the unit designated by "*" to the standard input and output devices (by this we generally mean the keyboard and screen of the personal computer but input/output can be redirected from/to a file using the standard input/output redirectors "<" and ">" on the command line). This preconnection cannot be altered in any way. Unit "*" is explicitly or implicitly referred to by the following input statements:

```
READ, ...
READ *, ...
READ format-spec, ...
READ(*,...) ...
READ(UNIT=*,...) ...
```

Unit "*" is explicitly or implicitly referred to by the following output statements:

```
PRINT, ...
PRINT *, ...
PRINT format-spec, ...
WRITE(*,...) ...
WRITE(UNIT=*,...) ...
```

The Open Watcom F77 run-time system also defines a preconnection of unit 5 to the standard input device (by this we generally mean the keyboard of the personal computer but input can be redirected from a file using the standard input redirector "<" on the command line).

The Open Watcom F77 run-time system also defines a preconnection of unit 6 to the standard output device (by this we generally mean the screen of the personal computer but output can be redirected to a file using the standard output redirector ">" on the command line).

For all other allowable units, a default preconnection between unit number "nnn" and the file `FORnnn` is assumed when no connection between a unit and a file has been established. *nnn* is a three-digit FORTRAN unit number. Unit 0 is "000", unit 1 is "001", unit 2 is "002", and so on. There is no file extension in this case. In other words, a default file name is constructed for any unit number for which no other connection has been established. Input/output statements of the following forms refer to these units.

```
CLOSE(nnn,...)              OPEN(nnn,...)
CLOSE(UNIT=nnn,...)         OPEN(UNIT=nnn,...)
BACKSPACE nnn               READ(nnn,...)
BACKSPACE(nnn)              READ(UNIT=nnn,...)
BACKSPACE(UNIT=nnn)         REWIND nnn
ENDFILE nnn                 REWIND(nnn)
ENDFILE(nnn)                REWIND(UNIT=nnn)
ENDFILE(UNIT=nnn)           WRITE(nnn,...) ...
INQUIRE(nnn,...)            WRITE(UNIT=nnn,...) ...
INQUIRE(UNIT=nnn,...)
```

Of course, it is unlikely that one would be satisfied with using such undistinguished file names such as `for000`, `for001`, and so on. Therefore, the Open Watcom F77 run-time system provides additional ways of establishing a preconnection between a FORTRAN ***UNIT*** and a file.

The Open Watcom F77 run-time system supports the use of the "SET" command to establish a connection between a unit and a file. The "SET" command is used to create, modify and remove "Environment Variables". The "SET" command must be issued before running a program. The format for a preconnection using the "SET" command is:

```
SET unit=file_spec
```

where	***description***
unit	is a FORTRAN unit number in the range 0 to 999. If this form of the "SET" command is used then FORTRAN unit number *unit* is preconnected to the specified file. FORTRAN input/output statements which refer to the unit number will access the records in the specified file.
file_spec	is the file specification of the preconnected file.

Here are some sample "SET" commands.

Example:

```
C>set 1=input.dat
C>set 2=output.dat
C>set 3=d:\database\customer.fil
```

The above example establishes the following preconnections:

1. Between unit 1 and the file `input.dat` which resides (or will reside) in the current directory.
2. Between unit 2 and the file `output.dat` which resides (or will reside) in the current directory.

3. Between unit 3 and the file `d:\database\customer.fil` which resides (or will reside) in another disk and directory.

Any FORTRAN input/output statements which refer to units 1, 2 or 3 will act upon one of these 3 data files.

Notes:

1. The "SET" command must be issued before running the program.

2. No spaces should be placed before or after the "=" in the "SET" command. The following two examples are quite distinct from each other:

   Example:
   ```
   C>set 55=testbed.dat
   C>set 55 = testbed.dat
   ```

   To verify this, simply enter the two commands and then enter the "SET" command again with no arguments. The current environment strings will be displayed. You should find two entries, one for "55" and one for "55 ".

3. Since the number in front of the "=" is simply a character string, you should not specify any leading zeroes either.

   Example:
   ```
   C>set 01=input.dat
   C>set 1=input.dat
   ```

   In this case, we again have two distinct environment variables. The variable "01" will be ignored by the Open Watcom F77 run-time system.

4. An environment variable will remain in effect until you explicitly remove it or you turn off the personal computer. To discontinue the preconnection between a unit number and a file, you must issue a "SET" command of the following form.

   ```
   C>set <unit>=
   ```

   In the above command, `<unit>` is the unit number for which the preconnection is to be discontinued.

   By omitting the character string after the "=", the environment variable will be removed. For example, to remove the environment variable "01" from the list, reenter the "SET" command specifying everything up to and including the "=" character.

   Example:
   ```
   C>set 01=
   ```

5. Any time you wish to see the current list of environment strings, simply enter the "SET" command with no arguments.

Example:

```
C>set
PROMPT=$d $t $p$_$n$g
COMSPEC=d:\dos\command.com
PATH=G:\;E:\CMDS;C:\WATCOM\BIN;D:\DOS;D:\BIN
LIB=c:\watcom\lib286\dos
1=input.dat
2=output.dat
3=d:\database\customer.fil
```

6. An alternative to preconnecting files is provided by the FORTRAN ***OPEN*** statement which allows files to be connected at execution time.

7. The preconnection of units 5 and 6 may be overridden using preconnection specifications or the FORTRAN ***OPEN*** statement. The precedence of a connection between a unit number and a file is as follows:

Precedence:	***User option:***
Lowest	Preconnection Specifications
Highest	OPEN statement

In other words, the ***OPEN*** statement overrides a preconnection.

6.7 A Preconnection Tutorial

In this section, we will look at some examples of how to establish the link between a file and a FORTRAN unit.

Exhibit 1:

Consider the following example which reads pairs of numbers from a file and writes out the numbers and their sum.

```
* File 'iodemo.for'
10    READ( 1, *, END=99 ) X1, X2
      WRITE( 6, 20 ) X1, X2, X1 + X2
      GO TO 10
20    FORMAT( 3F6.2 )
99    END
```

The FORTRAN ***READ*** statement will read records from a file connected to unit 1. The FORTRAN ***WRITE*** statement will write records to a file connected to unit 6. As we described in the previous section, unit 6 is preconnected by the Open Watcom F77 run-time system to the screen.

What file will be read when the ***READ*** statement refers to unit 1? By default, we know that it will read a file called `for001`. However, suppose the data was actually stored in the file called `numbers.dat`. We can direct the program to read the data in this file by using a "SET" command before running the program.

Example:

```
C>set 1=numbers.dat
C>iodemo
  1.40  2.50  3.90
  3.90  8.70 12.60
  1.10  9.90 11.00
  8.30  7.10 15.40
  8.20  3.50 11.70
```

Exhibit 2:

Suppose that we now wish to write the output from the above program to a disk file instead of the screen. We can do this without modifying the program. Since we know that the ***WRITE*** statement refers to unit 6, we can alter the default preconnection of unit 6 to the screen by issuing another "SET" command.

Example:

```
C>set 6=numbers.rpt
C>iodemo
C>type numbers.rpt
  1.40  2.50  3.90
  3.90  8.70 12.60
  1.10  9.90 11.00
  8.30  7.10 15.40
  8.20  3.50 11.70
```

Now any time a program writes or prints to unit 6, the output will be written to the disk file `numbers.rpt`. If you are going to run other programs, it would be wise to remove the connection between unit 6 and this file so that it is not accidentally overwritten. This can be done by issuing the following command.

Example:

```
C>set 6=
```

You should also do the same for unit 1.

Exhibit 3:

Must we always use "SET" commands to establish the connection between a unit and a file? Suppose that you want to run the program quite often and that you do not want to issue "SET" commands every time. We can do this by modifying the program to include FORTRAN ***OPEN*** statements.

```
* File 'iodemo.for'
      OPEN( 1, FILE='NUMBERS.DAT' )
      OPEN( 6, FILE='NUMBERS.RPT' )
10    READ( 1, *, END=99 ) X1, X2
      WRITE( 6, 20 ) X1, X2, X1 + X2
      GO TO 10
20    FORMAT( 3F6.2 )
99    END
```

This is an example of a connection that is established at execution time. The connection that is established by the ***OPEN*** statement overrides any preconnection that we might have established using a "SET" command. We say that the ***OPEN*** statement has a higher precedence. However, even the ***OPEN*** statement does not have the final word on which files will be accessed. You may wish to read the next section on the Open Watcom F77 run-time system logical file name support to find out why this is so.

6.8 Logical File Name Support

The Open Watcom F77 run-time system supports logical or symbolic file names using the "SET" command. The "SET" command may be used to define a logical file name and its corresponding actual file name. The format for defining a logical file name is as follows:

```
SET name=file_spec
```

where ***description***

name is any character string. The letters in "name" may be specified in upper or lower case. Lower case letters are treated as if they had been specified in upper case. Thus "SYSINPUT" and "sysinput" are equivalent. Note, however, that blank characters must not be specified before and after the "=" character.

file_spec is the file specification of logical file.

Notes and Examples:

1. A logical file name may be used in the ***FILE=*** specifier of the FORTRAN ***OPEN*** and ***INQUIRE*** statements.

   Example:

   ```
   * File 'iodemo.for'
         OPEN( 1, FILE='SYSINPUT' )
   10    READ( 1, *, END=99 ) X1, X2
         WRITE( 6, 20 ) X1, X2, X1 + X2
         GO TO 10
   20    FORMAT( 3F6.2 )
   99    END
   ```

   In the following example, we define the logical file name "SYSINPUT" to correspond to the file `numbers.dat`.

   Example:

   ```
   C>set sysinput=numbers.dat
   C>iodemo
     1.40  2.50  3.90
     3.90  8.70 12.60
     1.10  9.90 11.00
     8.30  7.10 15.40
     8.20  3.50 11.70
   ```

2. If the name in a ***FILE=*** specifier is not included in one of the environment variable names then it is assumed to be the actual name of a file.

Example:

```
OPEN( 2, FILE='SYSOUT' )
```

3. The logical file name feature can also be used to provide additional information regarding the file name at execution time.

   Example:

   ```
   * File 'iodemo.for'
         OPEN( 1, FILE='numbers.dat' )
   10    READ( 1, *, END=99 ) X1, X2
         WRITE( 6, 20 ) X1, X2, X1 + X2
         GO TO 10
   20    FORMAT( 3F6.2 )
   99    END
   ```

   In the following example, the actual location (and name) of the file `numbers.dat` is described through the use of an environment variable.

   Example:

   ```
   C>set numbers.dat=b:\data\input.dat
   C>iodemo
   ```

   As you can see, a logical file name can resemble an actual file name.

   Of course, the entire file name could have been specified in the FORTRAN program.

   Example:

   ```
   OPEN( 1, FILE='b:\data\input.dat' )
   ```

4. Only one level of lookup is performed.

   Example:

   ```
   * File 'iodemo.for'
         OPEN( 1, FILE='sysinput' )
   10    READ( 1, *, END=99 ) X1, X2
         WRITE( 6, 20 ) X1, X2, X1 + X2
         GO TO 10
   20    FORMAT( 3F6.2 )
   99    END
   ```

   This is illustrated by the following commands.

   Example:

   ```
   C>set sysinput=datafile
   C>set datafile=input.dat
   C>iodemo
   ```

   In the above example, unit 1 is connected to the file `datafile` and not the file `input.dat`.

5. Logical file names can be used to direct output normally intended for one device to another device. Consider the following examples.

Example:

```
C>set lpt1=lpt2
```

If the FORTRAN program specifies the name "LPT1" in an ***OPEN*** or ***INQUIRE*** statement, the Open Watcom F77 run-time system will map this name to "LPT2". In an ***INQUIRE*** statement, the ***NAME=*** specifier will return the name "LPT2".

6. As we mentioned earlier, the case of the name does not matter. Upper or lower case can be used interchangeably.

   Example:

   ```
   C>set sysinput=b:\data\input.dat
   C>set SYSINPUT=b:\data\input.dat
   ```

7. No spaces should be placed before or after the "=" in the "SET" command. The following two examples are considered quite distinct from each other:

   Example:

   ```
   C>set sysinput=testbed.dat
   C>set sysinput = testbed.dat
   ```

   This example will define two variables, "SYSINPUT" and "SYSINPUT ".

8. An environment variable will remain in effect until you explicitly remove it or you turn off the personal computer. To remove an environment variable from the list, reenter the "SET" command specifying everything up to and including the "=" character. For example, to remove the definition for "SYSINPUT", the following command can be issued.

   Example:

   ```
   C>set sysinput=
   ```

9. Any time you wish to see the current list of environment strings, simply enter the "SET" command with no arguments.

   Example:

   ```
   C>set
   PROMPT=$d $t $p$_$n$g
   COMSPEC=d:\dos\command.com
   PATH=G:\;E:\CMDS;C:\WATCOM\BIN;D:\DOS;D:\BIN
   LIB=c:\watcom\lib286\dos
   1=input.dat
   2=output.dat
   3=d:\database\customer.fil
   SYSINPUT=b:\data\input.dat
   LPT1=lpt2
   ```

6.9 Terminal or Console Device Support

Input can come from the console or output can be written to the console by using the console device name `con` as the file name. The console can be specified in a "SET" command or through the ***FILE=*** specifier of the FORTRAN ***OPEN*** statement.

The default action for any file is to open the file for both read and write access (i.e., ACTION='READWRITE'). Under Win32, there is a problem accessing the console device `con` for both

read and write access. This problem is overcome by using the ***ACTION=*** specifier in the ***OPEN*** statement. The ***ACTION=*** specifier indicates the way in which the file is initially accessed. The values allowed for the ***ACTION=*** specifier are the following.

'READ' the file is opened for read-only access

'WRITE' the file is opened for write-only access

'READWRITE' the file is opened for both read and write access

To open the console device under Win32, you must specify whether you are going to "READ" or "WRITE" to the file. If you wish to do both reading and writing, then you must use two separate units.

Example:
```
OPEN( UNIT=1, FILE='CON', ACTION='READ')
OPEN( UNIT=2, FILE='CON', ACTION='WRITE')
```

The console can be treated as a carriage control device. This is requested by using the `CARRIAGECONTROL='YES'` specifier of the FORTRAN ***OPEN*** statement.

Example:
```
OPEN( UNIT=1, FILE='con', CARRIAGECONTROL='YES' )
```

Carriage control handling is described in the section entitled "Print File Attributes" on page 43.

The console is not capable of supporting carriage control in a fashion identical to a printer. For example, overprinting of records on the console is destructive in that the previous characters are erased.

End of file is signalled by first pressing the Ctrl/Z key combination and then the line entering key. End of file may be handled by using the ***END=*** specification of the FORTRAN ***READ*** statement.

Example:
```
      READ( UNIT=*, FMT=*, END=100 ) X, Y
      .
      .
      .
100   code to handle "End of File"
```

End of file may also be handled by using the ***IOSTAT=*** specifier of the FORTRAN ***READ*** statement.

Example:
```
READ( UNIT=*, FMT=*, IOSTAT=IOS ) X, Y
IF( IOS .NE. 0 )THEN
code to handle "End of File"
ENDIF
```

6.10 Printer Device Support

Output can be written to a printer by using a printer device name as the file name. A printer can be specified in a "SET" command or through the ***FILE=*** specifier of the FORTRAN ***OPEN*** statement. Several device names may be used:

```
prn or lpt1
lpt2
lpt3
```

The printer can be treated as a carriage control device. This is requested by using the `CARRIAGECONTROL='YES'` specifier of the FORTRAN ***OPEN*** statement.

Example:

```
OPEN( UNIT=1, FILE='prn', CARRIAGECONTROL='YES' )
```

Carriage control handling is described in the section entitled "Print File Attributes" on page 43.

6.11 Serial Device Support

Output can be written to a serial port by using a serial device name as the file name. A serial device can be specified in a "SET" command or through the ***FILE=*** specifier of the FORTRAN ***OPEN*** statement. Three device names may be used:

```
aux or com1
com2
```

The serial device can be treated as a carriage control device. This is requested by using the `CARRIAGECONTROL='YES'` specifier of the FORTRAN ***OPEN*** statement.

Example:

```
OPEN( UNIT=1, FILE='com1', CARRIAGECONTROL='YES' )
```

Carriage control handling is described in the section entitled "Print File Attributes" on page 43.

To set serial characteristics such as speed, parity, and word length, the "MODE" command may be used.

Example:

```
C>mode com1:9600,n,8,1
```

The above example sets serial port 1 to a speed of 9600 BAUD with no parity, a word length of 8 and 1 stop bit.

6.12 File Handling Defaults

The following defaults apply to file specifications:

- The following table indicates the default *record type* for the allowable access methods and forms.

File Access	Form	
	Formatted	Unformatted
Sequential	Text	Variable
Direct	Text	Fixed

Unless the record type of the file does not correspond to the default assumed by Open Watcom F77, the record type attribute should not be specified.

- Unless otherwise stated, the default *record length* for a file is 1024 characters. When access is "direct", the record length must be specified in the ***RECL=*** specifier of the FORTRAN ***OPEN*** statement. The record length may also be specified when the access is "sequential". This should be done whenever access is "sequential" and the maximum record length is greater than the default.

- The default *record access* is "sequential".

- When reading from or writing to a unit for which no preconnection has been specified or no "FILE=" form of the FORTRAN ***OPEN*** statement has been executed, the default *file name* takes the form:

```
FORnnn
```

nnn is a three-digit FORTRAN unit number. Unit 0 is "000", unit 1 is "001", unit 2 is "002", and so on. There is no file extension in this case.

- If the connection between a unit number and a file is discontinued through use of the FORTRAN ***CLOSE*** statement, the same rule for constructing a file name will apply on the next attempt to read from or write to the specified unit.

7 The Open Watcom F77 Subprogram Library

Open Watcom FORTRAN 77 includes additional FORTRAN subprograms which can be called from programs compiled by Open Watcom F77. The following sections describe these subprograms.

7.1 Subroutine FEXIT

The subroutine `FEXIT` allows an application to terminate execution with a return code. It requires one argument of type INTEGER that represents the value to be returned to the system.

Example:

```
INCLUDE 'FSUBLIB.FI'
CALL FEXIT( -1 )
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.2 INTEGER Function FGETCMD

The INTEGER function `FGETCMD` allows an application to obtain the command line from within an executing program.

The function `FGETCMD` requires one argument of type CHARACTER and returns the length of the command line.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER CMDLEN
CHARACTER*128 CMDLIN

CMDLEN = FGETCMD( CMDLIN )
PRINT *, 'Command length = ', CMDLEN
PRINT *, 'Command line   = ', CMDLIN
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. If the argument to FGETCMD is not long enough then only the first part of the command line is returned.

7.3 INTEGER Function FGETENV

The INTEGER function FGETENV allows an application to obtain the value of an environment string from within an executing program.

The function FGETENV requires two arguments of type CHARACTER. The first argument is the character string to look for. FGETENV places the associated environment string value in the second argument and returns the length of the environment string. If no such string is defined, the length returned is zero.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER STRLEN
CHARACTER*80 STRVAL

STRLEN = FGETENV( 'PATH', STRVAL )
PRINT *, 'Environment string length = ', STRLEN
PRINT *, 'Environment string value  = ', STRVAL
END
```

Notes:

1. The FORTRAN include file fsublib.fi, located in the \watcom\src\fortran directory, contains typing and calling information for this subprogram. The \watcom\src\fortran directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. If the second argument to FGETENV is not long enough then only the first part of the value is returned.

7.4 INTEGER Function FILESIZE

The INTEGER function FILESIZE allows an application to determine the size of a file connected to a specified unit.

The function FILESIZE requires one argument of type INTEGER, the unit number and returns the size, in bytes, of the file. If no file is connected to the specified unit, a value of -1 is returned.

Example:

```
INCLUDE 'FSUBLIB.FI'

OPEN( UNIT=1, FILE='sample.fil' )
PRINT *, FILESIZE( 1 )
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.5 Subroutine FINTR and FINTRF

The subroutine `FINTR` and `FINTRF` allow the user to execute any software interrupt from a FORTRAN 77 program.

> ***Note:*** These subroutines are only supported by the DOS and Windows libraries.

The subroutine `FINTR` and `FINTRF` require two arguments.

1. The first argument is an interrupt number. These subroutines will generate the software interrupt given by the this argument. The type must be `INTEGER`.
2. The second argument is an `INTEGER` array of ten elements.

When `FINTR` and `FINTRF` are called, the array contains the values to be assigned to the registers prior to issuing the software interrupt. When control is returned from `FINTR` or `FINTRF`, it contains the values of the registers after the software interrupt has completed. The registers are mapped onto the array `REGS` as follows.

```
              31      0
REGS(1)           EAX
REGS(2)           EBX
REGS(3)           ECX
REGS(4)           EDX
REGS(5)           EBP
REGS(6)           ESI
REGS(7)           EDI
REGS(8)         FS | DS
REGS(9)         GS | ES
REGS(10)       eflags
```

For 16-bit systems (e.g., 8088, 8086, 186, 286), only the low-order 16 bits of each register contain meaningful results.

```
              31      0
REGS(1)             AX
REGS(2)             BX
REGS(3)             CX
REGS(4)             DX
REGS(5)             BP
REGS(6)             SI
REGS(7)             DI
REGS(8)             DS
REGS(9)             ES
REGS(10)         flags
```

Difference between FINTR and FINTRF is that FINTR reset CPU flags before generate the software interrupt, but FINTRF set it from REGS(10) element.

The file dos.fi, located in the \WATCOM\src\fortran\dos directory, defines a set of equivalences for ease of use. The contents of this file are reproduced below.

```
* Define registers: These correspond to the element of an
* array which is to contain the values of the registers.

      integer*4 regd(10), regs(10)
      integer*2 regw(2*10)
      integer*1 regb(4*4)

      integer*4 EAX,EBX,ECX,EDX,EBP,EDI,ESI,EFLAGS
      integer*2 AX,BX,CX,DX,BP,DI,SI,DS,ES,FS,GS,FLAGS
      integer*1 AH,AL,BH,BL,CH,CL,DH,DL
      equivalence (regd,regs),(regd,regw),(regd,regb),
     1(EAX,regd(1)), (EBX,regd(2)), (ECX,regd(3)), (EDX,regd(4)),
     2(EBP,regd(5)), (EDI,regd(6)), (ESI,regd(7)), (EFLAGS,regd(10)),
     3(AX,regw(1)),  (BX,regw(3)),  (CX,regw(5)),  (DX,regw(7)),
     4(BP,regw(9)),  (DI,regw(11)), (SI,regw(13)), (DS,regw(15)),
     5(FS,regw(16)), (ES,regw(17)), (GS,regw(18)), (FLAGS,regw(19)),
     6(AL,regb(1)),  (AH,regb(2)),  (BL,regb(5)),  (BH,regb(6)),
     7(CL,regb(9)),  (CH,regb(10)), (DL,regb(13)), (DH,regb(14))
```

The following is extracted from the "CALENDAR" program. It demonstrates the use of the FINTR subroutine.

```
      subroutine ClearScreen()
*$noextensions
      implicit none

      include 'dos.fi'

* Define BIOS functions.

      integer VIDEO_CALL, SCROLL_UP
      parameter (VIDEO_CALL=16, SCROLL_UP=6)

      DS = ES = FS = GS = 0
      AH = SCROLL_UP            ! scroll up
      AL = 0                    ! blank entire window
      CX = 0                    ! set row,column of upper left
      DX = 24*256 + 80          ! set row,column of lower right
      BH = 7                    ! attribute "white on black"
      call fintr( VIDEO_CALL, regs )
      end
```

7.6 INTEGER Function FLUSHUNIT

The INTEGER function FLUSHUNIT flushes the internal input/output buffer for a specified unit. Each file, except special devices such as con, has an internal buffer. Buffered input/output is much more efficient since it reduces the number of system calls which are usually quite expensive. For example, many ***WRITE*** operations may be required before filling the internal file buffer and data is physically transferred to the file.

This function is particularly useful for applications that call non-FORTRAN subroutines or functions that wish to perform input/output to a FORTRAN file.

The function FLUSHUNIT requires one argument, the unit number, of type INTEGER. It returns an INTEGER value representing the return code of the input/output operation. A return value of 0 indicates success; otherwise an error occurred.

The following example will flush the contents of the internal input/output buffer for unit 7.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER ISTAT

ISTAT = FLUSHUNIT( 7 )
IF( ISTAT .NE. 0 )THEN
    PRINT *, 'Error in FLUSHUNIT'
END IF

END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.7 INTEGER Function FNEXTRECL

The INTEGER function `FNEXTRECL` reports the record length of the next unformatted record to be read sequentially from the specified unit.

The function `FNEXTRECL` requires one argument, the unit number, of type INTEGER. It returns an INTEGER value representing the size of the next record to be read.

The following example creates an unformatted file and then reads the records in the file sequentially.

Example:

```
      INCLUDE 'FSUBLIB.FI'

      CHARACTER*80 INPUT

      OPEN(UNIT=2, FILE='UNFORM.TXT', FORM='UNFORMATTED',
     & ACCESS='SEQUENTIAL' )
      WRITE( UNIT=2 ) 'A somewhat longish first record'
      WRITE( UNIT=2 ) 'A short second record'
      WRITE( UNIT=2 ) 'A very, very much longer third record'
      CLOSE( UNIT=2 )

      OPEN(UNIT=2, FILE='UNFORM.TXT', FORM='UNFORMATTED',
     & ACCESS='SEQUENTIAL' )

      I = FNEXTRECL( 2 )
      PRINT *, 'Record length=', I
      READ( UNIT=2 ) INPUT(1:I)
      PRINT *, INPUT(1:I)

      I = FNEXTRECL( 2 )
      PRINT *, 'Record length=', I
      READ( UNIT=2 ) INPUT(1:I)
      PRINT *, INPUT(1:I)

      I = FNEXTRECL( 2 )
      PRINT *, 'Record length=', I
      READ( UNIT=2 ) INPUT(1:I)
      PRINT *, INPUT(1:I)
      CLOSE( UNIT=2 )
      END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.8 INTEGER Function FSIGNAL

The INTEGER function `FSIGNAL` allows your application to respond to certain events that occur during execution.

Event	*Meaning*
SIGBREAK	an interactive attention (Ctrl/Break on keyboard) is signalled
SIGFPE	an erroneous floating-point operation occurs (such as division by zero, overflow and underflow)
SIGILL	illegal instruction encountered
SIGINT	an interactive attention (Ctrl/C on keyboard) is signalled

SIGSEGV an illegal memory reference is detected

SIGTERM a termination request is sent to the program

SIGIDIVZ integer division by zero

SIGIOVFL integer overflow

The function FSIGNAL requires two arguments. The first argument is an INTEGER argument and must be one of the events described above. The second argument, called the handler, is one of the following.

1. a subprogram that is called when the event occurs
2. the value SIG_DFL, causing the default action to be taken when the event occurs
3. the value SIG_IGN, causing the event to be ignored

FSIGNAL returns SIG_ERR if the request could not be processed, or the previous event handler.

Example:

```
INCLUDE 'FSIGNAL.FI'

EXTERNAL BREAK_HANDLER
LOGICAL BREAK_FLAG
COMMON BREAK_FLAG
BREAK_FLAG = .FALSE.
CALL FSIGNAL( SIGBREAK, BREAK_HANDLER )
WHILE( .NOT. VOLATILE( BREAK_FLAG ) ) CONTINUE
PRINT *, 'Program Interrupted'
END

SUBROUTINE BREAK_HANDLER()
LOGICAL BREAK_FLAG
COMMON BREAK_FLAG
BREAK_FLAG = .TRUE.
END
```

Notes:

1. The FORTRAN include file `fsignal.fi` contains typing and calling information for FSIGNAL and should be included when using this function. This file is located in the `\watcom\src\fortran` directory. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. The intrinsic function VOLATILE is used to indicate that the reference to the variable `break_flag` is volatile. A volatile reference prevents the compiler from caching a variable in a register. In this case, we want to retrieve the value of `break_flag` from memory each time the loop is iterated.

7.9 INTEGER Function FSPAWN

The INTEGER function FSPAWN allows an application to run another program as a subprocess. When the program completes, execution is returned to the invoking application. There must be enough available free memory to start the subprocess.

The function FSPAWN requires two arguments of type CHARACTER. The first argument is a character string representing the name of the program to be run. The string must end in a NULL character (i.e., a character with the binary value 0).

The second argument is a character string argument list to be passed to the program. The first character of the second argument must contain, in binary, the length of the remainder of the argument list. For example, if the argument is the string "HELLO" then the first character would be CHAR(5) and the remaining characters would be "HELLO" (see the example below).

FSPAWN returns an INTEGER value representing the status of subprocess execution. If the value is negative then the program could not be run. If the value is positive then the value represents the program's return code.

Example:

```
          INCLUDE 'FSUBLIB.FI'
          INTEGER CMDLEN, STATUS
          CHARACTER CMD*128, CMDLIN*128

    * COMSPEC will tell us where DOS 'COMMAND.COM' is hiding
          CMDLEN = FGETENV( 'COMSPEC', CMD )
          CMD(CMDLEN+1:CMDLEN+1) = CHAR( 0 )

          CMDLIN = '/c dir *.for'
          CMDLIN(13:13) = CHAR( 0 )

          STATUS = FSPAWN( CMD, CMDLIN )
          PRINT *, 'Program status = ', STATUS
          END
```

Notes:

1. The FORTRAN include file fsublib.fi, located in the \watcom\src\fortran directory, contains typing and calling information for this subprogram. The \watcom\src\fortran directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. The INTEGER function FSYSTEM, which is described in a later section, implements a more general form of the example given above. We recommend its use.

7.10 INTEGER Function FSYSTEM

The INTEGER function FSYSTEM allows an application to run another program or execute an operating system command.

The function FSYSTEM requires one argument of type CHARACTER. This argument represents a operating system command or a program name together with any arguments. FSYSTEM returns an INTEGER value representing the status of subprocess execution. If the value is negative, the operating

system command interpreter or shell could not be run (an attempt is made to invoke the system command interpreter to run the program). If the value is positive, the value represents the program's return code.

In the following example, a "COPY" command is executed and then a hypothetical sorting program is run.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER STATUS

STATUS = FSYSTEM( 'COPY *.FOR \BACKUP\FOR\SRC' )
PRINT *, 'Status of COPY command = ', STATUS
STATUS = FSYSTEM( 'SORTFILE/IN=INP.DAT/OUT=OUT.DAT' )
PRINT *, 'Status of SORT program = ', STATUS
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.11 Subroutine FTRACEBACK

The subroutine `FTRACEBACK` allows your application to generate a run-time traceback. The application must be compiled with the "DEBUG" or "TRACE" option. It is useful when you wish to disclose a problem in an application and provide an informative report of where the problem occurred in the application.

The `FTRACEBACK` subroutine requires no arguments. The `FTRACEBACK` subroutine does not terminate program execution.

Example:

```
SUBROUTINE READREC( UN )

INCLUDE 'FSUBLIB.FI'

INTEGER UN
INTEGER RLEN
CHARACTER*35 INPUT

RLEN = FNEXTRECL( UN )
IF( RLEN .GT. 35 )THEN
  PRINT *, 'Error: Record too long', RLEN
  CALL FTRACEBACK
  STOP
ELSE
  PRINT *, 'Record length=', RLEN
  READ( UNIT=UN ) INPUT(1:RLEN)
  PRINT *, INPUT(1:RLEN)
ENDIF
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.12 Subroutine GETDAT

The subroutine `GETDAT` allows an application to obtain the current date.

The subroutine `GETDAT` has three arguments of type `INTEGER*2`. When control is returned from `GETDAT`, they contain the year, month and day of the current date.

The following program prints the current date in the form "YY-MM-DD".

Example:

```
        INCLUDE 'FSUBLIB.FI'
        INTEGER*2 YEAR, MONTH, DAY
        CALL GETDAT( YEAR, MONTH, DAY )
        PRINT 100, YEAR, MONTH, DAY
100     FORMAT( 1X, I4, '-', I2.2, '-', I2.2 )
        END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. The arguments to `GETDAT` must be of type INTEGER*2 in order to obtain correct results.

7.13 Subroutine GETTIM

The subroutine `GETTIM` allows an application to obtain the current time.

The subroutine `GETTIM` has four arguments of type `INTEGER*2`. When control is returned from `GETTIM`, they contain the hours, minutes, seconds, and hundredths of seconds of the current time.

The following program prints the current time in the form "HH:MM:SS.TT".

Example:

```
        INCLUDE 'FSUBLIB.FI'
        INTEGER*2 HRS, MINS, SECS, HSECS
        CALL GETTIM( HRS, MINS, SECS, HSECS )
        PRINT 100, HRS, MINS, SECS, HSECS
100     FORMAT( 1X, I2.2, ':', I2.2, ':', I2.2, '.', I2.2 )
        END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. The arguments to `GETTIM` must be of type INTEGER*2 in order to obtain correct results.

7.14 INTEGER Function GROWHANDLES

The INTEGER function `GROWHANDLES` allows an application to increase the maximum number of files that can be opened. It requires one argument of type INTEGER representing the maximum number of files that can be opened and returns an INTEGER value representing the actual limit. The actual limit may differ from the specified limit. For example, memory constraints or system parameters may be such that the request cannot be satisfied.

The following example attempts to increase the limit on the number of open files to sixty-four.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER NEW_LIMIT

NEW_LIMIT = GROWHANDLES( 64 )

END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.15 Functions IARGC and IGETARG

The function `IARGC` allows an application to determine the number of arguments (including the program name) used to invoke the program. The function `IGETARG` can be used to retrieve an argument.

Arguments supplied to a program are assigned indices. Argument zero is the program name, argument one is the first argument, etc. The function `IGETARG` requires two arguments. The first argument is the index of the argument to retrieve and is of type INTEGER. The second argument is of type CHARACTER and is used to return the argument. The size of the argument (number of characters) is returned.

Example:

```
      INCLUDE 'FSUBLIB.FI'
      CHARACTER*128 ARG
      INTEGER ARGC, ARGLEN

      ARGC = IARGC()
      ARGLEN = IGETARG( 0, ARG )
      PRINT *, 'Program name is ', ARG(1:ARGLEN)
      DO I = 1, ARGC - 1
          ARGLEN = IGETARG( I, ARG )
          PRINT '(A, I2, 2A)', 'Argument ', I, ' is ',
     1                          ARG(1:ARGLEN)
      END DO
      END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

7.16 Math Error Functions

Math error functions are called when an error is detected in a math library function. For example, if the second argument to the `AMOD` intrinsic function is zero, a math error function will be called. A number of math error functions are defined in the FORTRAN run-time libraries and perform default actions when an error is detected. These actions typically produce an error message to the screen.

It is possible to replace the FORTRAN run-time library version of the math error functions with your own versions. The file `_matherr.for` located in the `\watcom\src\fortran` directory can be used as a template for defining your own math error functions. The following functions represent the set of math error functions.

1. The function `__imath2err` is called for math functions of type INTEGER that take two arguments of type INTEGER. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the first argument passed to the math function and the third argument is a pointer to the second argument passed to the math function. The error function returns a value that is then used as the return value for the math function.
2. The function `__amath1err` is called for math functions of type REAL that take one argument of type REAL. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the argument passed to the math function. The error function returns a value that is then used as the return value for the math function.
3. The function `__amath2err` is called for math functions of type REAL that take two arguments of type REAL. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the first argument passed to the math function and the third argument is a pointer to the second argument passed to the math function. The error function returns a value that is then used as the return value for the math function.
4. The function `__math1err` is called for math functions of type DOUBLE PRECISION that take one argument of type DOUBLE PRECISION. The first argument represents the error

information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the argument passed to the math function. The error function returns a value that is then used as the return value for the math function.

5. The function `__math2err` is called for math functions of type DOUBLE PRECISION that take two arguments of type DOUBLE PRECISION. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the first argument passed to the math function and the third argument is a pointer to the second argument passed to the math function. The error function returns a value that is then used as the return value for the math function.
6. The function `__zmath2err` is called for math functions of type COMPLEX that take two arguments of type COMPLEX. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the first argument passed to the math function and the third argument is a pointer to the second argument passed to the math function. The error function returns a value that is then used as the return value for the math function.
7. The function `__qmath2err` is called for math functions of type DOUBLE COMPLEX that take two arguments of type DOUBLE COMPLEX. The first argument represents the error information and is an argument of type INTEGER that is passed by value. The second argument is a pointer to the first argument passed to the math function and the third argument is a pointer to the second argument passed to the math function. The error function returns a value that is then used as the return value for the math function.

The include file `mathcode.fi` is included by the file `_matherr.for` and is located in the `\watcom\src\fortran` directory. It defines the information that is contained in the error information argument that is passed to all math error functions.

7.17 INTEGER Function SEEKUNIT

The INTEGER function `SEEKUNIT` permits seeking to a particular byte offset within a file connected to a FORTRAN unit. The file must be opened with the following attributes:

```
FORM='UNFORMATTED'
ACCESS='SEQUENTIAL'
RECORDTYPE='FIXED'
```

The function `SEEKUNIT` requires three arguments of type INTEGER, the unit number, the offset to seek, and the type of positioning to do. The seek positioning may be absolute (indicated by 0) or relative to the current position (indicated by 1). It returns an INTEGER value representing the new offset in the file. A returned value of -1 indicates that the function call failed.

This function is particularly useful for applications that wish to change the input/output position for a file connected to a unit.

The following example will set the current input/output position of the file connected to the specified unit.

Example:

```
      EXTERNAL SEEKUNIT
      INTEGER SEEKUNIT
      INTEGER SEEK_SET, SEEK_CUR
      PARAMETER (SEEK_SET=0, SEEK_CUR=1)

      INTEGER POSITION
      CHARACTER*80 RECORD

      OPEN( UNIT=8, FILE='file', FORM='UNFORMATTED',
     1        ACCESS='SEQUENTIAL', RECORDTYPE='FIXED' )
      POSITION = SEEKUNIT( 8, 10, SEEK_SET )
      IF( POSITION .NE. -1 )THEN
          PRINT *, 'New position is', POSITION
          READ( UNIT=8 ) RECORD
          PRINT *, RECORD
      ENDIF
      END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. A value of -1 is returned if the requested positioning cannot be done.

7.18 INTEGER Function SETJMP/Subroutine LONGJMP

The INTEGER function `SETJMP` saves the current executing environment, making it possible to restore that environment by subsequently calling the `LONGJMP` subroutine. For example, it is possible to implement error handling by using `SETJMP` to record the point to which a return will occur following an error. When an error is detected in a called subprogram, that subprogram uses `LONGJMP` to jump back to the recorded position. The original subprogram which called `SETJMP` must still be active (it cannot have returned to the subprogram which called it).

The `SETJMP` function requires one argument. The argument is a structure of type `jmp_buf` and is used to save the current environment. The return value is an integer and is zero when initially called. It is non-zero if the return is the result of a call to the `LONGJMP` subroutine. An ***IF*** statement is often used to handle these two cases. This is demonstrated in the following example.

Example:

```
include 'fsignal.fi'
include 'setjmp.fi'
record /jmp_buf/ jmp_buf
common jmp_buf
external break_handler
integer rc
call fsignal( SIGBREAK, break_handler )
rc = setjmp( jmp_buf )
if( rc .eq. 0 )then
    call do_it()
else
    print *, 'abnormal termination:', rc
endif
end

subroutine do_it()
loop
end loop
end

subroutine break_handler()
include 'setjmp.fi'
record /jmp_buf/ jmp_buf
common jmp_buf
call longjmp( jmp_buf, -1 )
end
```

Notes:

1. The FORTRAN include file `setjmp.fi` contains typing and calling information for `SETJMP` and `LONGJMP` and must be included. Similarly, `fsignal.fi` must be included when using the `FSIGNAL` function. These files are located in the `\watcom\src\fortran` directory. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate these include files.

7.19 INTEGER Function SETSYSHANDLE

The INTEGER function `SETSYSHANDLE` allows an application to set the system file handle for a specified unit.

The function `SETSYSHANDLE` requires an argument of type INTEGER, the unit number, and an argument of type INTEGER*2, the handle, and returns an INTEGER value representing the success or fail status of the function call. A returned value of -1 indicates that the function call failed and 0 indicates that the function call succeeded.

This function is particularly useful for applications that wish to set the system file handle for a unit. The system file handle may have been obtained from a non-FORTRAN subroutine or function.

The following example will set the system file handle for a paricular unit.

Example:

```
INCLUDE 'FSUBLIB.FI'
INTEGER STDIN, STDOUT
PARAMETER (STDIN=0, STDOUT=1)

OPEN( UNIT=8, FORM='FORMATTED' )
I = SYSHANDLE( 8 )
PRINT *, 'Old handle was', I
I = SETSYSHANDLE( 8, STDOUT )
IF( I .EQ. 0 )THEN
    WRITE( UNIT=8, FMT=* ) 'Output to UNIT 8 which is stdout'
ENDIF
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. A value of -1 is returned if the unit is not connected to a file.

3. Units 5 and 6 are preconnected to the standard input and standard output devices respectively.

7.20 INTEGER*2 Function SYSHANDLE

The INTEGER*2 function `SYSHANDLE` allows an application to obtain the system file handle for a specified unit.

The function `SYSHANDLE` requires one argument of type INTEGER, the unit number. and returns an INTEGER*2 value representing the system file handle.

This function is particularly useful for applications that wish to pass the system file handle to non-FORTRAN subroutines or functions that wish to perform input/output to a FORTRAN 77 file.

The following example will print the system file handles for the standard input and standard output devices.

Example:

```
INCLUDE 'FSUBLIB.FI'

PRINT *, 'Unit 5 file handle is', SYSHANDLE( 5 )
PRINT *, 'Unit 6 file handle is', SYSHANDLE( 6 )
END
```

Notes:

1. The FORTRAN include file `fsublib.fi`, located in the `\watcom\src\fortran` directory, contains typing and calling information for this subprogram. The `\watcom\src\fortran` directory should be included in the **FINCLUDE** environment variable so that the compiler can locate the include file.

2. A value of -1 is returned if the unit is not connected to a file.

3. Units 5 and 6 are preconnected to the standard input and standard output devices respectively.

7.21 REAL Function URAND

The REAL function URAND returns pseudo-random numbers in the range (0,1).

The function URAND requires one argument of type INTEGER, the initial seed. The seed can contain any integer value. URAND returns a REAL value which is a pseudo-random number in the range (0.0,1.0).

In the following example, 100 random numbers are printed.

Example:

```
REAL URAND
INTEGER SEED

SEED = 75347
DO I = 1, 100
   PRINT *, URAND( SEED )
ENDDO
END
```

Notes:

1. Upon each invocation of URAND, the seed argument is updated by the random number generator. Therefore, the argument must not be a constant and, once the seed value has been set, it must *not* be modified by the programmer.

7.22 Default Windowing Functions

The functions described in the following sections provide the capability to manipulate attributes of various windows created by Open Watcom's default windowing system for Microsoft Windows 3.x, Windows 95, Windows NT, and IBM OS/2. A simple default windowing FORTRAN application can be built using the following command(s):

16-bit Windows
```
C>wfl [fn1] [fn2] ...  /bw /windows /l=windows
```

32-bit Windows
```
C>wfl386 [fn1] [fn2] ...  /bw /l=win386
C>wbind -n [fn1]
```

32-bit Windows NT or Windows 95
```
C>wfl386 [fn1] [fn2] ...  /bw /l=nt_win
```

32-bit OS/2 Presentation Manager
```
C>wfl386 [fn1] [fn2] ...  /bw /l=os2v2_pm
```

Note: At present, a restriction in Windows NT prevents you from opening the console device (CON) for both read and write access. Therefore, it is not possible to open additional windows for both input and output under Windows NT. They must be either read-only or write-only windows.

7.22.1 dwfDeleteOnClose

```
integer function dwfDeleteOnClose( unit )
integer unit
```

The dwfDeleteOnClose function tells the console window that it should close itself when the corresponding file is closed. The argument `unit` is the unit number associated with the opened console.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfDeleteOnClose function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )
      END
```

7.22.2 dwfSetAboutDlg

```
integer function dwfSetAboutDlg( title, text )
character*(*) title
character*(*) text
```

The dwfSetAboutDlg function sets the "About" dialog box of the default windowing system. The argument `title` is a character string that will replace the current title. If `title` is CHAR(0) then the title will not be replaced. The argument `text` is a character string which will be placed in the "About" box. To get multiple lines, embed a new line character (CHAR(13)) after each logical line in the string. If `text` is CHAR(0), then the current text in the "About" box will not be replaced.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfSetAboutDlg function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )
      END
```

7.22.3 dwfSetAppTitle

```
integer function dwfSetAppTitle( title )
character*(*) title
```

The dwfSetAppTitle function sets the main window's title. The argument `title` is a character string that will replace the current title.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfSetAppTitle function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )
      END
```

7.22.4 dwfSetConTitle

```
integer function dwfSetConTitle( unit, title )
integer unit
character*(*) title
```

The dwfSetConTitle function sets the console window's title which corresponds to the unit number passed to it. The argument `unit` is the unit number associated with the opened console. The argument `title` is the character string that will replace the current title.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfSetConTitle function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )
      END
```

7.22.5 dwfShutDown

```
integer function dwfShutDown()
```

The dwfShutDown function shuts down the default windowing I/O system. The application will continue to execute but no windows will be available for output. Care should be exercised when using this function since any subsequent output may cause unpredictable results.

When the application terminates, it will not be necessary to manually close the main window.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfShutDown function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )
      rc = dwfShutDown()

*   do more computing that does not involve console input/output
*         .
*         .
*         .

      END
```

7.22.6 dwfYield

```
integer function dwf veld()
```

The dwfYield function yields control back to the operating system, thereby giving other processes a chance to run.

This function is one of the support functions that can be called from an application using Open Watcom's default windowing support.

The dwfYield function returns 1 if it was successful and 0 if not.

Example:

```
      PROGRAM main
      INCLUDE 'FSUBLIB.FI'

      INTEGER rc
      CHARACTER response
      INTEGER i

      rc = dwfSetAboutDlg( 'Hello World About Dialog',
     1                     'About Hello World'//CHAR(13)//
     2                     'Copyright 1994 by WATCOM'//CHAR(13) )
      rc = dwfSetAppTitle( 'Hello World Application Title' )
      rc = dwfSetConTitle( 5, 'Hello World Console Title' )
      PRINT *, 'Hello World'
      OPEN( unit=3, file='CON' )
      rc = dwfSetConTitle( 3, 'Hello World Second Console Title' )
      rc = dwfDeleteOnClose( 3 )
      WRITE( unit=3, fmt=* ) 'Hello to second console'
      WRITE( unit=3, fmt=* ) 'Press Enter to close this console'
      READ( unit=3, fmt='(A)', end=100, err=100 ) response
100   CLOSE( unit=3 )

      DO i = 0, 1000
          rc = dwfYield()
*         do CPU-intensive calculation
*         .
*         .
*         .
      ENDDO
      PRINT *, i

      END
```

8 Data Representation On x86-based Platforms

This chapter describes the internal or machine representation of the basic types supported by Open Watcom F77. The following table summarizes these data types.

Data Type	Size (in bytes)	FORTRAN 77 Standard
LOGICAL	4	
LOGICAL*1	1	(extension)
LOGICAL*4	4	(extension)
INTEGER	4	
INTEGER*1	1	(extension)
INTEGER*2	2	(extension)
INTEGER*4	4	(extension)
REAL	4	
REAL*4	4	(extension)
REAL*8	8	(extension)
DOUBLE PRECISION	8	
COMPLEX	8	
COMPLEX*8	8	(extension)
COMPLEX*16	16	(extension)
DOUBLE COMPLEX	16	(extension)
CHARACTER	1	
CHARACTER*n	n	

8.1 LOGICAL*1 Data Type

An item of type **LOGICAL*1** occupies 1 byte of storage. It can only have two values, namely .TRUE. (a value of 1) and .FALSE. (a value of 0).

8.2 LOGICAL and LOGICAL*4 Data Types

An item of type **LOGICAL** or **LOGICAL*4** occupies 4 bytes of storage. It can only have two values, namely .TRUE. (a value of 1) and .FALSE. (a value of 0).

8.3 INTEGER*1 Data Type

An item of type **INTEGER*1** occupies 1 byte of storage. Its value is in the following range. An integer *n* can be represented in 1 byte if

```
-128 <= n <= 127
```

8.4 INTEGER*2 Data Type

An item of type **INTEGER*2** occupies 2 bytes of storage. An integer *n* can be represented in 2 bytes if

```
-32768 <= n <= 32767
```

8.5 INTEGER and INTEGER*4 Data Types

An item of type **INTEGER** or **INTEGER*4** occupies 4 bytes of storage (one numeric storage unit). An integer *n* can be represented in 4 bytes if

```
-2147483648 <= n <= 2147483647
```

8.6 REAL and REAL*4 Data Types

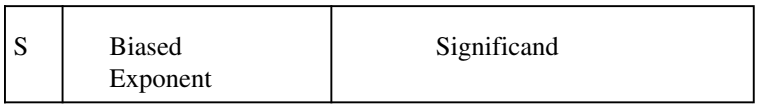
An item of type **REAL** or **REAL*4** is an approximate representation of a real number and occupies 4 bytes (one numeric storage unit). If `m` is the magnitude of a real number `x`, then `x` can be approximated if

```
 -126          128
2     <= m < 2
```

or in more approximate terms if

```
1.175494e-38 <= m <= 3.402823e38
```

Items of type **REAL** or **REAL*4** are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



31 30-23 22-0

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 127 (i.e., exponent value 1 represents 2**-126; exponent value 127 represents 2**0; exponent value 254 represents 2**127; etc.). The exponent field is 8 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 23 bits long.

Zero A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

8.7 DOUBLE PRECISION and REAL*8 Data Types

An item of type **DOUBLE PRECISION** or **REAL*8** is an approximate representation of a real number, occupies 8 bytes (two numeric storage units) and has precision greater than or equal to that of an item of type **REAL** or **REAL*4**. If `m` is the magnitude of a real number `x`, then `x` can be approximated if

```
 -1022          1024
2       <= m < 2
```

or in more approximate terms if

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Items of type **DOUBLE PRECISION** or **REAL*8** are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.

S	Biased Exponent	Significand

63 62-52 51-0

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 1023 (i.e., exponent value 1 represents 2**-1022; exponent value 1023 represents 2**0; exponent value 2046 represents 2**1023; etc.). The exponent field is 11 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 52 bits long.

Zero A double precision zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

8.8 COMPLEX, COMPLEX*8, and DOUBLE COMPLEX Data Types

An item of type **COMPLEX** or **COMPLEX*8** is an approximate representation of a complex number. The representation is an ordered pair of real numbers, the first representing the real part of the complex number and the second representing the imaginary part of the complex number. Each item of type **COMPLEX** or **COMPLEX*8** occupies 8 bytes (two consecutive numeric storage units), the first being the real part and the second the imaginary part. The approximation of the real and imaginary parts of a complex number is the same degree of approximation used for items of type **REAL**.

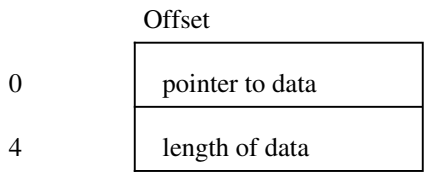
8.9 COMPLEX*16 Data Type

An item of type **COMPLEX*16** is an approximate representation of a complex number. The representation is an ordered pair of real numbers, the first representing the real part of the complex number and the second representing the imaginary part of the complex number. Each item of type **COMPLEX*16** occupies 16 bytes (four consecutive numeric storage units), the first two being the real part and the last two the imaginary part. The approximation of the real and imaginary parts of a complex number is the same degree of approximation used for items of type **DOUBLE PRECISION**.

8.10 CHARACTER Data Type

An item of type **CHARACTER** represents a sequence of characters. Each character occupies 1 byte of storage (1 character storage unit). The length of an item of type **CHARACTER** is the number of characters it contains. Each character is assigned an integer that represents its position. Characters are numbered from 1 to *n* starting from the left, *n* being the number of characters.

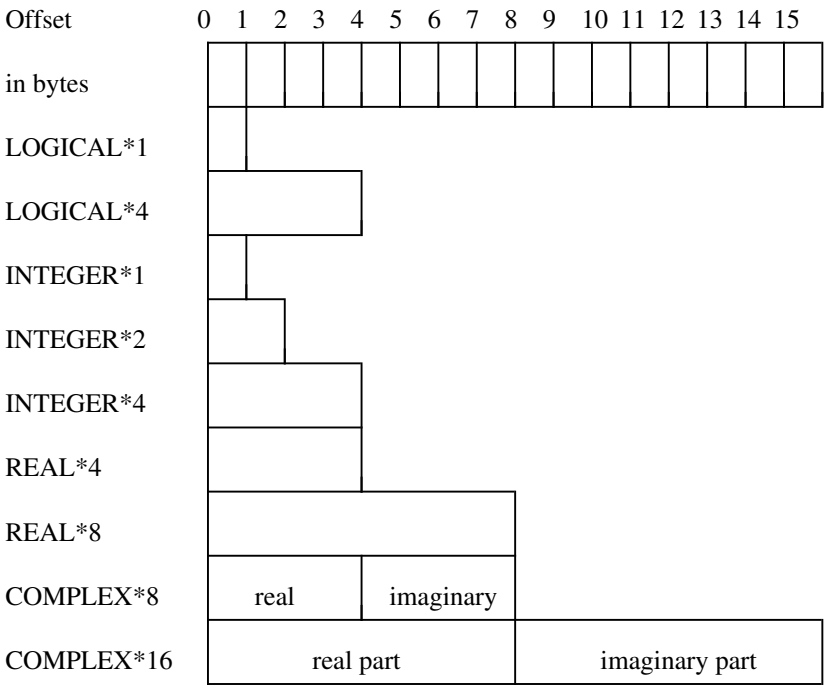
Items of type **CHARACTER** are represented by a *string descriptor*. A string descriptor has the following format.



The pointer to the actual data is a 32-bit offset in the default data segment. The length is represented as a 32-bit unsigned integer.

8.11 Storage Organization of Data Types

The following illustrates the relative size of the data types in terms of bytes. **LOGICAL** is equivalent to **LOGICAL*4**, **INTEGER** is equivalent to **INTEGER*4**, **DOUBLE PRECISION** is equivalent to **REAL*8**, and **COMPLEX** is equivalent to **COMPLEX*8**. If the "short" option is used, **LOGICAL** is equivalent to **LOGICAL*1** and **INTEGER** is equivalent to **INTEGER*2**.



8.12 Floating-point Accuracy On x86-based Platforms

There are a number of issues surrounding floating-point accuracy, calculations, exceptions, etc. on the x86-based personal computer platform that we will address in the following sections. Some result from differences in the behaviour of standard-conforming FORTRAN 77 compilers. Other result from idiosyncrasies of the IEEE Standard 754 floating-point that is supported on the x86 platform.

Some FORTRAN 77 compilers extend the precision of single-precision constants in DATA statement initialization lists when the corresponding variable is double precision. This is permitted by the FORTRAN 77 Standard. Open Watcom FORTRAN 77, however, does not do this. This is illustrated by the following example.

Example:

```
double precision pi1, pi2
data pi1 /3.141592653589793/
data pi2 /3.141592653589793d0/
write(unit=*,fmt='(1x,z16,1x,f18.15)') pi1, pi1
write(unit=*,fmt='(1x,z16,1x,f18.15)') pi2, pi2
end
```

The output produces two very different results for our pi variables. The variable `PI1` is initialized with a single precision (i.e., REAL) constant.

```
400921FB60000000  3.141592741012573
400921FB54442D18  3.141592653589793
```

A single precision datum has 23 bits in the mantissa; a double precision datum has 52 bits in the mantissa. Hence `PI1` has 29 fewer bits of accuracy in the mantissa (the difference between 52 and 23) since it is initialized with a single precision constant. You can verify this by examining the hexadecimal output of the two pi's. The bottom 29 bits of the mantissa in `PI1` are all zero.

To be on the safe side, the rule is always use double precision constants (even in DATA statements) if you want as much accuracy as possible.

This behaviour treats DATA statement initialization as equivalent to simple assignment as shown in the following example.

Example:

```
          double precision pi1, pi2
          pi1 = 3.141592653589793
          pi2 = 3.141592653589793d0
          write(unit=*,fmt='(1x,z16,1x,f18.15)') pi1, pi1
          write(unit=*,fmt='(1x,z16,1x,f18.15)') pi2, pi2
          end
```

The output follows:

```
 400921FB60000000  3.141592741012573
 400921FB54442D18  3.141592653589793
```

A second consideration is illustrated by the next example. On some computer architectures, there is no difference in the exponent range between single and double precision floating-point representation. One such architecture is the IBM mainframe computer (e.g., IBM System/370). When a double precision result is assigned to a single precision (REAL) variable, only precision in the mantissa is lost.

The x86 platform uses the IEEE Standard 754 floating-point representation. In this representation, the range of exponent values is greater in double precision than in single precision. As described in the section entitled "REAL and REAL*4 Data Types" on page 82, the range for single precision (REAL, REAL*4) numbers is:

```
1.175494e-38 <= m <= 3.402823e38
```

On the other hand, the range for double precision (DOUBLE PRECISION, REAL*8) numbers is:

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Double precision is described in the section entitled "DOUBLE PRECISION and REAL*8 Data Types" on page 83. So you can see that a number like 1.0E234 can easily be represented in double precision but not in single precision since the maximum positive exponent value for single precision is 38.

8.13 Floating-point Exceptions On x86-based Platforms

The following types of exceptions can be enabled/disabled on PC's with an 80x87 floating-point unit (either a real FPU or a true emulator).

DENORMAL The result has become denormalized. When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number. By providing a significand with leading zeros, the range of possible negative exponents can be extended by the number of bits in the significand. Each leading zero is a bit of lost accuracy, so the extended exponent range is obtained by reducing significance.

ZERODIVIDE A division by zero was attempted. A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

OVERFLOW The result has overflowed. The correct answer is finite, but has a magnitude too great to be represented in the destination floating-point format.

UNDERFLOW The result has numerically underflowed. The correct answer is non-zero but has a magnitude too small to be represented as a normal number in the destination floating-point format. IEEE Standard 754 specifies that an attempt be made to represent the number as a denormal. This denormalization may result in a loss of significant bits from the significand.

PRECISION A calculation does not return an exact answer. This exception is usually masked (disabled) and ignored. It is used in extremely critical applications, when the user must know if the results are exact. The precision exception is called "inexact" in IEEE Standard 754.

INVALID This is the exception condition that covers all cases not covered by the other exceptions. Included are FPU stack overflow and underflow, NAN inputs, illegal infinite inputs, out-of-range inputs, and inputs in unsupported formats.

Which exceptions does Open Watcom FORTRAN 77 catch and which ones does it ignore by default? We can determine the answer to this with the following program.

```
* This program uses the C Library routine "_control87"
* to obtain the math coprocessor exception mask.

      implicit none
      include 'fsignal.fi'

      character*8 status
      integer fp_cw, bits

      fp_cw = _control87( 0, 0 )
      bits = IAND( fp_cw, MCW_EM )
      print '(a,1x,z4)', 'Interrupt exception mask', bits
      print *,'Invalid operation exception ', status(bits, EM_INVALID)
      print *,'Denormalized exception ', status(bits, EM_DENORMAL)
      print *,'Divide by 0 exception ', status(bits, EM_ZERODIVIDE)
      print *,'Overflow exception ', status(bits, EM_OVERFLOW)
      print *,'Underflow exception ', status(bits, EM_UNDERFLOW)
      print *,'Precision exception ', status(bits, EM_PRECISION)
      end

      character*8 function status( bits, mask )
      integer bits, mask

      if( IAND(bits,mask) .eq. 0 ) then
        status = 'enabled'
      else
        status = 'disabled'
      endif
      end
```

If you compile and run this program, the following output is produced.

```
Interrupt exception mask 0032
Invalid operation exception enabled
Denormalized exception disabled
Divide by 0 exception enabled
Overflow exception enabled
Underflow exception disabled
Precision exception disabled
```

So, by default, the Open Watcom FORTRAN 77 run-time system will catch "invalid operation", "divide by 0", and "overflow" exceptions. It ignores "denormal", "underflow", and "precision" exceptions. Thus calculations that produce very small results trend towards zero. Also, calculations that produce inexact results (a very common occurrence in floating-point calculations) are allowed to continue.

Suppose that you were interested in flagging calculations that result in denormalized or underflowed results. To do this, we need to enable both DENORMAL and UNDERFLOW exceptions. This following program illustrates how to do this.

```
*$ifdef __386__
*$ifdef __stack_conventions__
*$pragma aux _clear87 "!"
*$else
*$pragma aux _clear87 "!_"
*$endif
*$else
*$pragma aux _clear87 "!_"
*$endif

      implicit none
      include 'fsignal.fi'

      character*8 status
      integer fp_cw, fp_mask, bits

*     get rid of any errors so we don't cause an instant exception
      call _clear87

*     fp_mask determines the bits to enable and/or disable
      fp_mask = 0
     1      + EM_DENORMAL
     2      + EM_UNDERFLOW

*     fp_cw determines whether to enable(0) or disable(1)
*     (in this case, nothing is disabled)
      fp_cw = '0000'x

      fp_cw = _control87( fp_cw, fp_mask )

      bits = IAND( fp_cw, MCW_EM )
      print '(a,1x,z4)', 'Interrupt exception mask', bits
      print *,'Invalid operation exception ', status(bits, EM_INVALID)
      print *,'Denormalized exception ', status(bits, EM_DENORMAL)
      print *,'Divide by 0 exception ', status(bits, EM_ZERODIVIDE)
      print *,'Overflow exception ', status(bits, EM_OVERFLOW)
      print *,'Underflow exception ', status(bits, EM_UNDERFLOW)
      print *,'Precision exception ', status(bits, EM_PRECISION)
      end

      character*8 function status( bits, mask )
      integer bits, mask

      if( IAND(bits,mask) .eq. 0 ) then
        status = 'enabled'
      else
        status = 'disabled'
      endif
      end
```

If you compile and run this program, the following output is produced.

```
Interrupt exception mask 0020
Invalid operation exception enabled
Denormalized exception enabled
Divide by 0 exception enabled
Overflow exception enabled
Underflow exception enabled
Precision exception disabled
```

8.14 Compiler Options Relating to Floating-point

Let us take the program that we developed in the previous section and test it out. If you introduce the variable `FLT` to the program and calculate the expression "2e-38 x 2e-38", you would expect to see 0.0 printed when underflow exceptions are disabled and a run-time diagnostic when underflow exceptions are enabled. The statements that you would add are show in the following.

```
      real flt

      flt=2e-38
      print *, flt*flt

* code to enable exceptions goes here

      print *, flt*flt

      end
```

If you compile the modified program with default options and run it, the result is as follows.

```
        0.0000000
Interrupt exception mask 0020
Invalid operation exception enabled
Denormalized exception enabled
Divide by 0 exception enabled
Overflow exception enabled
Underflow exception enabled
Precision exception disabled
        0.0000000
```

This is not what we expected. Evaluation of the second expression did not produce the run-time diagnostic that we expected. The reason this happened is related to the compiler's processing of the source code. By default, the compiler optimized the generated code by evaluating the expression "2e-38 x 2e-38" at compile time producing 0.0 as the result (due to the underflow).

```
      flt=2e-38
      print *, flt*flt

reduces to

      print *, 2e-28*2e-38

which further reduces to

      print *, 0.0
```

Recompile the program using the "OP" option and run it. The result is as follows.

```
       0.0000000
Interrupt exception mask 0020
Invalid operation exception enabled
Denormalized exception enabled
Divide by 0 exception enabled
Overflow exception enabled
Underflow exception enabled
Precision exception disabled
*ERR* KO-03 floating-point underflow
```

The use of the "OP" option will force the result to be stored in memory after each FORTRAN statement is executed. Thus, the source code is not optimized across statements. Compile-time versus run-time evaluation of expressions can lead to different results. It is very instructive to compile and then run your application with a variety of compile-time options to see the effect of optimizations. See the chapter entitled "Open Watcom FORTRAN 77 Compiler Options" on page 5 for more information on compiler options.

Before we end this section, there is another important aspect of floating-point exceptions to consider. A floating-point exception is triggered upon the execution of the next FPU instruction following the one that caused the exception.

```
implicit none

real*4 a
real*8 b

b=12.0d123
a=b*b
b=1.0
a=b/2.0
print *, a, b
end
```

Compile this program with the "OP" and "DEBUG" options and then run it. The result is displayed next.

```
*ERR* KO-02 floating-point overflow
 - Executing line 9 in file pi4.for
```

Line 9 is the line containing the statement `a=b/2.0` which could not possibly be responsible for an overflow. However, it contains the first floating-point instruction following the instruction in line 7 where the overflow actually occurred. To see this, it helps to disassemble the object file.

```
      a=b*b
0029    B8 07 00 00 00      mov       eax,0x00000007
002E    E8 00 00 00 00      call      RT@SetLine
0033    DD 45 F4            fld       qword ptr -0xc[ebp]
0036    D8 C8               fmul      st,st
0038    D9 5D FC            fstp      dword ptr -0x4[ebp]
      b=1.0
003B    B8 09 00 00 00      mov       eax,0x00000009
0040    E8 00 00 00 00      call      RT@SetLine
0045    31 DB               xor       ebx,ebx
0047    89 5D F4            mov       -0xc[ebp],ebx
004A    C7 45 F8 00 00 F0 3F
                            mov       dword ptr -0x8[ebp],0x3ff00000
      a=b/2.0
0051    B8 0A 00 00 00      mov       eax,0x0000000a
0056    E8 00 00 00 00      call      RT@SetLine
005B    DD 45 F4            fld       qword ptr -0xc[ebp]
005E    DC 0D 08 00 00 00
                            fmul      qword ptr L$2
0064    D9 5D FC            fstp      dword ptr -0x4[ebp]
```

The overflow occurred when the "fstp" was executed but is signalled when the subsequent "fld" is executed. The overflow could also be signalled while executing down in a run-time routine. This behaviour of the FPU can be somewhat exasperating.

8.15 Floating-point Exception Handling

In certain situations, you want to handle floating-point exceptions in the application itself rather than let the run-time system terminate your application. The following example illustrates how to do this by installing a FORTRAN subroutine as a floating-point exception handler.

```
      implicit none
      include 'fsignal.fi'

      real flt
      external fpehandler
      integer      signal_count, signal_number, signal_type
      common /fpe/ signal_count, signal_number, signal_type

*     begin the signal handling process for floating-point exceptions
      call fsignal( SIGFPE, fpehandler )
*
*   main body of application goes here
*
      flt = 2.0
      print *, 'number of signals', volatile( signal_count )
      print *, flt / 0.0
      print *, 'number of signals', volatile( signal_count )

      end

*$ifdef __386__
*$ifdef __stack_conventions__
*$pragma aux _clear87 "!"
*$else
*$pragma aux _clear87 "!_"
*$endif
*$else
*$pragma aux _clear87 "!_"
*$endif

*$pragma aux fpehandler parm( value, value )

      subroutine fpehandler( sig_num, fpe_type )

      implicit none

*     sig_num and fpe_type are passed by value, not by reference
      integer sig_num, fpe_type

      include 'fsignal.fi'

      integer      signal_count, signal_number, signal_type
      common /fpe/ signal_count, signal_number, signal_type
*     we could add this to our common block
      integer      signal_split( FPE_INVALID:FPE_IOVERFLOW )

      signal_count = signal_count + 1
      signal_number = sig_num
      signal_type = fpe_type

*     floating-point exception types

*     FPE_INVALID         = 129 (0)
*     FPE_DENORMAL        = 130 (1)
*     FPE_ZERODIVIDE      = 131 (2)
*     FPE_OVERFLOW        = 132 (3)
*     FPE_UNDERFLOW       = 133 (4)
*     FPE_INEXACT         = 134 (5)
*     FPE_UNEMULATED      = 135 (6)
*     FPE_SQRTNEG         = 136 (7)
*     undefined           = 138 (8)
*     FPE_STACKOVERFLOW   = 137 (9)
*     FPE_STACKUNDERFLOW  = 138 (10)
*     FPE_EXPLICITGEN     = 139 (11)
*     FPE_IOVERFLOW       = 140 (12)

*     log the type of error for interest only */
      signal_split( fpe_type ) =
     1signal_split( fpe_type ) + 1

*     get rid of any errors
      call _clear87
```

```
*     resignal for more exceptions
      call fsignal( SIGFPE, fpehandler )

*     if we don't then a subsequent exception will
*     cause an abnormal program termination

      end
```

Note the use of the `VOLATILE` intrinsic function to obtain up-to-date contents of the variable `SIGNAL_COUNT`.

16-bit Topics

9 16-bit Memory Models

9.1 Introduction

This chapter describes the various 16-bit memory models supported by Open Watcom F77. Each memory model is distinguished by two properties; the code model used to implement subprogram calls and the data model used to reference data.

9.2 16-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to subprograms are made with *near calls.* In a near call, the destination address is 16 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library subprograms, must be less than 64K. Open Watcom F77 does not support the small code model.

A big code model is one in which all calls to subprograms are made with *far calls.* In a far call, the destination address is 32 bits (a segment value and an offset relative to the segment value). This model allows the size of the code comprising your program to exceed 64K.

9.3 16-bit Data Models

There are three data models;

1. the small data model,
2. the big data model and
3. the huge data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 16 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 64K.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 32 bits (a segment value and an offset relative to the segment value). This removes the 64K limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Open Watcom F77 assumes that the offset portion of a far pointer will not be incremented beyond 64K. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 64K bytes. For example, an array containing 40,000 integers does not fit into the big data model. An object such as this should be described as *huge.*

A huge data model is one in which all references to data are made with far pointers. This is similar to the big data model. However, in the huge data model, incrementing a far pointer will adjust the offset *and* the segment if necessary. The limit on the size of an object pointed to by a far pointer imposed by the big data model is removed in the huge data model.

Notes:

1. The huge data model has the same characteristics as the big data model, but formal array arguments are assumed to exceed 64K bytes. You should use the huge data model whenever any arrays in your application exceed 64K bytes in size.

2. If your program contains less than 64K of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

3. The huge data model should be used only if needed. The code generated in the huge data model is not very efficient since a run-time routine is called in order to increment far pointers. This increases the size of the code significantly and increases execution time.

9.4 Summary of 16-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Open Watcom F77.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
medium	big	small	far	near
large	big	big	far	far
huge	big	huge	far	huge

9.5 Mixed 16-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that includes arrays which are larger than 64K bytes.

For example, a medium memory model application that uses some arrays which exceed 64K bytes in total size can be described as a mixed memory model. In an application such as this, most of the data is in a 64K segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model.

9.6 Linking Applications for the Various 16-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model.

Library	Memory model	Floating-point model
flibm.lib	/mm	/fpc
flibl.lib	/ml, /mh	/fpc
flib7m.lib	/mm	/fpi, /fpi87
flib7l.lib	/ml, /mh	/fpi, /fpi87
clibm.lib	/mm	/fpc, /fpi, /fpi87
clibl.lib	/ml, /mh	/fpc, /fpi, /fpi87
mathm.lib	/mm,	/fpc
mathl.lib	/ml, /mh	/fpc
math87m.lib	/mm,	/fpi, /fpi87
math87l.lib	/ml, /mh	/fpi, /fpi87
emu87.lib	/mm, /ml, /mh	/fpi
noemu87.lib	/mm, /ml, /mh	/fpi87

9.7 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"
6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom F77.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<subprogram>_TEXT" where <subprogram> is the name of a subprogram.

2. The "FAR_DATA" class consists of the following:

   (a) arrays whose size exceeds the data threshold in large data memory models (the data threshold is 256 bytes unless changed using the "dt" compiler option)

   (b) equivalenced variables in large data memory models

10 16-bit Assembly Language Considerations

10.1 Introduction

This chapter will deal with the following topics.

1. The memory layout of a program compiled by Open Watcom F77.
2. The method for passing arguments and returning values.
3. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Open Watcom F77 "fpi", "fpi87" or "fpi387" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" or "fpi387" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Open Watcom F77 "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

10.2 Calling Conventions

The following sections describe the method used by Open Watcom F77 to pass arguments.

The FORTRAN 77 language specifically requires that arguments be passed by reference. This means that instead of passing the value of an argument, its address is passed. This allows a called subprogram to modify the value of the actual arguments. The following illustrates the method used to pass arguments.

Type of Argument	Method Used to Pass Argument
non-character constant	address of constant
non-character expression	address of value of expression
non-character variable	address of variable
character constant	address of string descriptor
character expression	address of string descriptor
character variable	address of string descriptor
non-character array	address of array
non-character array element	address of array
character array	address of string descriptor
character array element	address of string descriptor
character substring	address of string descriptor
subprogram	address of subprogram
alternate return specifier	no argument passed
user-defined structure	address of structure

When passing a character array as an argument, the string descriptor contains the address of the first element of the array and the length of an element of the array.

The address of arguments are either passed in registers or on the stack. The registers used to pass the address of arguments to a subprogram are AX, BX, CX and DX. The address of arguments are passed in the following way.

1. For memory models with a big data model, address of arguments consist of a 16-bit offset and a 16-bit segment. Hence, two registers are required to pass the address of an argument. The first argument will be passed in registers DX:AX with register DX containing the segment and register AX containing the offset. The second argument will be passed in registers CX:BX with register CX containing the segment and register BX containing the offset.

2. For memory models with a small data model, address of arguments consists of only a 16-bit offset into the default data segment. Hence, only a single register is required to pass the address of an argument. The first argument is passed in register AX, the second argument is passed in register DX, the third argument is passed in register BX, and the fourth argument is passed in register CX.

3. For any remaining arguments, their address is passed on the stack. Note that addresses of arguments are pushed on the stack from right to left.

10.2.1 Processing Function Return Values with no 80x87

The way in which function values are returned is also dependent on the data type of the function. The following describes the method used to return function values.

1. **LOGICAL*1** values are returned in register AL.

2. **LOGICAL*4** values are returned in registers DX:AX.

3. **INTEGER*1** values are returned in register AL.

4. **INTEGER*2** values are returned in register AX.

5. **INTEGER*4** values are returned in registers DX:AX.

6. **REAL*4** values are returned in registers DX:AX.

7. **REAL*8** values are returned in registers AX:BX:CX:DX.

8. For **COMPLEX*8** functions, space is allocated on the stack by the caller for the return value. Register SI is set to point to the destination of the result. The called function places the result at the location pointed to by register SI.

9. For **COMPLEX*16** functions, space is allocated on the stack by the caller for the return value. Register SI is set to point to the destination of the result. The called function places the result at the location pointed to by register SI.

10. For **CHARACTER** functions, an additional argument is passed. This argument is the address of the string descriptor for the result. Note that the address of the string descriptor can be passed in any of the registers that are used to pass actual arguments.

11. For functions that return a user-defined structure, space is allocated on the stack by the caller for the return value. Register SI is set to point to the destination of the result. The called function places the result at the location pointed to by register SI. Note that a structure of size 1, 2 or 4 bytes is returned in register AL, AX or DX:AX respectively.

10.2.2 Processing Function Return Values Using an 80x87

The following describes the method used to return function values when your application is compiled using the "fpi87" or "fpi" option.

1. For **REAL*4** functions, the result is returned in floating-point register ST(0).
2. For **REAL*8** functions, the result is returned in floating-point register ST(0).
3. All other function values are returned in the way described in the previous section.

10.2.3 Processing Alternate Returns

Alternate returns are processed by the caller and are only allowed in subroutines. The called subroutine places the value specified in the ***RETURN*** statement in register AX. Note that the value returned in register AX is ignored if there are no alternate return specifiers in the actual argument list.

10.2.4 Alternate Method of Passing Character Arguments

As previously described, character arguments are passed using string descriptors. Recall that a string descriptor contains a pointer to the actual character data and the length of the character data. When passing character data, both a pointer and length are required by the subprogram being called. When using a string descriptor, this information can be passed using a single argument, namely the pointer to the string descriptor.

An alternate method of passing character arguments is also supported and is selected when the "nodescriptor" option is specified. In this method, the pointer to the character data and the length of the character data are passed as two separate arguments. The character argument lengths are appended to the end of the actual argument list.

Let us consider the following example.

```
INTEGER A, C
CHARACTER B, D
CALL SUB( A, B, C, D )
```

In the above example, the first argument is of type INTEGER, the second argument is of type CHARACTER, the third argument is of type INTEGER, and the fourth argument is of type CHARACTER. If the character arguments were passed by descriptor, the argument list would resemble the following.

1. The first argument would be the address of `A`.
2. The second argument would be the address of the string descriptor for `B`.
3. The third argument would be the address of `C`.
4. The fourth argument would be the address of the string descriptor for `D`.

If we specified the "nodescriptor" option, the argument list would be as follows.

1. The first argument would be the address of `A`.
2. The second argument would be the address of the character data for `B`.
3. The third argument would be the address of `C`.
4. The fourth argument would be the address of the character data for `D`.
5. A hidden argument for the length of `B` would be the fifth argument.
6. A hidden argument for the length of `D` would be the sixth argument.

Note that the arguments corresponding to the length of the character arguments are passed as INTEGER*2 arguments.

10.2.4.1 Character Functions

By default, when a character function is called, a hidden argument is passed at the end of the actual argument list. This hidden argument is a pointer to the string descriptor used for the return value of the character function. When the alternate method of passing character arguments is specified by using the "nodescriptor" option, the string descriptor for the return value is passed to the function as two hidden arguments, similar to the way character arguments were passed. However the two hidden arguments for the return value of the character function are placed at the beginning of the actual argument list. The first argument is the the pointer to the storage immediately followed by the size of the storage.

10.3 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"

6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Open Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom F77.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<subprogram>_TEXT" where <subprogram> is the name of a subprogram.

2. The "FAR_DATA" class consists of the following:

   (a) arrays whose size exceeds the data threshold in large data memory models (the data threshold is 256 bytes unless changed using the "dt" compiler option)

   (b) equivalenced variables in large data memory models

10.4 Writing Assembly Language Subprograms

When writing assembly language subprograms, use the following guidelines.

1. All used registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to be saved and restored if you are compiling your application with the "sr" option.

2. The direction flag must be clear before returning to the caller.

3. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, register CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

4. In a small data model, segment register DS contains the segment address of the default data segment (group "DGROUP"). In a big data model, segment register SS (not DS) contains the segment address of the default data segment (group "DGROUP").

5. When writing assembly language subprograms for the small code model, you must declare them as "near". If you wish to write assembly language subprograms for the big code model, you must declare them as "far".

6. Use the ".8087" pseudo-op so that floating-point constants are in the correct format.

7. The called subprogram must remove arguments that were passed on the stack in the "ret" instruction.

8. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

Consider the following example.

```
      INTEGER HRS, MINS, SECS, HSECS
      CALL GETTIM( HRS, MINS, SECS, HSECS )
      PRINT 100, HRS, MINS, SECS, HSECS
100   FORMAT( 1X,I2.2,':',I2.2,':',I2.2,'.',I2.2 )
      END
```

GETTIM is an assembly language subroutine that gets the current time. It requires four integer arguments. The arguments are passed by reference so that the subroutine can return the hour, minute, seconds and hundredths of a second for the current time. These arguments will be passed to GETTIM in the following way.

1. The address of the first argument will be passed in registers DX:AX.
2. The address of the second argument will be passed in registers CX:BX.
3. The address of the third argument will be passed on the stack.
4. The address of the fourth argument will be passed on the stack.

The following is an assembly language subprogram which implements GETTIM.

Large Memory Model (big code, big data)

```
GETTIM_TEXT segment byte public 'CODE'
        assume  CS:GETTIM_TEXT
        public  GETTIM
GETTIM  proc    far
        push    DI          ; save register(s)
        push    ES          ; ...
        push    DS          ; ...
        push    BP          ; get addressability to arguments
        mov     BP,SP       ; ...
        mov     ES,DX       ; ES:DI points to hours
        mov     DI,AX       ; ...
        mov     DS,CX       ; DS:BX points to minutes
        mov     AH,2ch      ; set DOS "get time" function
        int     21h         ; issue DOS function call
        mov     AL,CH       ; get hours
        cbw                 ; ...
        mov     ES:[DI],AX  ; return hours
        sub     AX,AX       ; ...
        mov     ES:2[DI],AX ; ...
        mov     AL,CL       ; get minutes
        cbw                 ; ...
        mov     [BX],AX     ; return minutes
        sub     AX,AX       ; ...
        mov     2[BX],AX    ; ...
        mov     DS,14[BP]   ; get address of seconds
        mov     DI,12[BP]   ; ...
        mov     AL,DH       ; get seconds
        cbw                 ; ...
        mov     [DI],AX     ; return seconds
        sub     AX,AX       ; ...
        mov     2[DI],AX    ; ...
        mov     DS,18[BP]   ; get address of ticks
        mov     DI,16[BP]   ; ...
        mov     AL,DL       ; get ticks
        cbw                 ; ...
```

```
        cwd                 ; ...
        mov     [DI],AX     ; return ticks
        mov     2[DI],DX    ; ...
        pop     BP          ; restore register(s)
        pop     DS          ; ...
        pop     ES          ; ...
        pop     DI          ; ...
        ret     8           ; return
GETTIM  endp
GETTIM_TEXT ends

        end
```

Notes:

1. Two arguments were passed on the stack so a "ret 8" instruction is used to return to the caller.

2. Registers AX, BX, CX and DX were not saved and restored since they were used to pass arguments. However, registers DS, ES, DI and BP were modified in the subprogram and hence must be saved and restored.

Let us look at the stack upon entry to `GETTIM`.

Large Model (big code, big data)

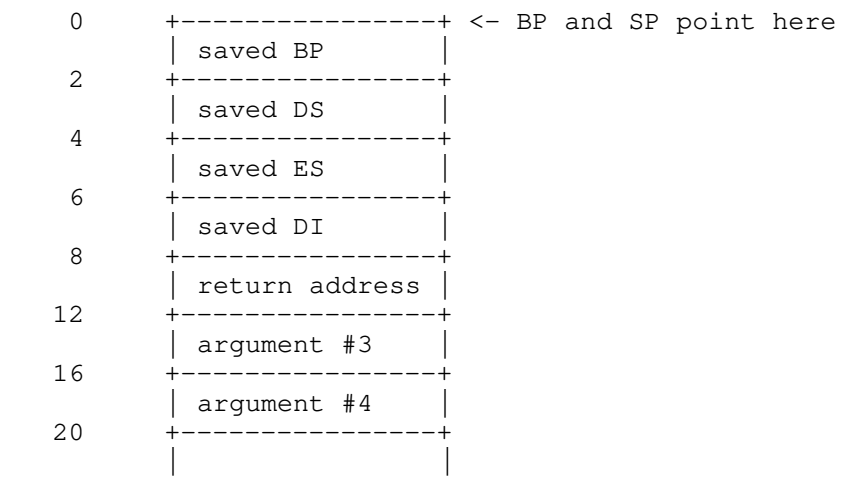
```
Offset
    0      +-----------------+ <- SP points here
           | return address  |
    4      +-----------------+
           | argument #3     |
    8      +-----------------+
           | argument #4     |
   12      +-----------------+
           |                 |
```

Notes:

1. The top element of the stack is a segment/offset pair forming a 32-bit return address. Hence, the third argument will be at offset 4 from the top of the stack and the fourth argument at offset 8.

Register SP cannot be used as a base register to address the arguments on the stack. Register BP is normally used to address arguments on the stack. Upon entry to the subroutine, registers that are modified (except those used to pass arguments) are saved and register BP is set to point to the stack. After performing this prologue sequence, the stack looks like this.

Large Model (big code, big data)



As the above diagram shows, the third argument is at offset 12 from register BP and the fourth argument is at offset 16.

10.4.1 Returning Values from Assembly Language Functions

The following illustrates the way function values are to be returned from assembly language functions.

1. A **LOGICAL*1** function.

```
L1_TEXT segment byte public 'CODE'
        assume  CS:L1_TEXT
        public  L1
L1      proc    far
        mov     AL,1
        ret
L1      endp
L1_TEXT ends
        end
```

2. A **LOGICAL*4** function.

```
L4_TEXT segment byte public 'CODE'
        assume  CS:L4_TEXT
        public  L4
L4      proc    far
        mov     AX,0
        cwd
        ret
L4      endp
L4_TEXT ends
        end
```

3. An **INTEGER*1** function.

```
I1_TEXT segment byte public 'CODE'
        assume  CS:I1_TEXT
        public  I1
I1      proc    far
```

```
        mov     AL,73
        ret
I1      endp
I1_TEXT ends
        end
```

4. An **INTEGER*2** function.

```
I2_TEXT segment byte public 'CODE'
        assume  CS:I2_TEXT
        public  I2
I2      proc    far
        mov     AX,7143
        ret
I2      endp
I2_TEXT ends
        end
```

5. An **INTEGER*4** function.

```
I4_TEXT segment byte public 'CODE'
        assume  CS:I4_TEXT
        public  I4
I4      proc    far
        mov     AX,383
        cwd
        ret
I4      endp
I4_TEXT ends
        end
```

6. A **REAL*4** function.

```
.8087

DGROUP  group R4_DATA

R4_TEXT segment byte public 'CODE'
        assume  CS:R4_TEXT
        assume  SS:DGROUP
        public  R4
R4      proc    far
        mov     AX,word ptr SS:R4Val
        mov     DX,word ptr SS:R4Val+2
        ret
R4      endp
R4_TEXT ends
R4_DATA segment byte public 'DATA'
R4Val   dd 1314.3
R4_DATA ends

        end
```

7. A **REAL*8** function.

```
.8087

DGROUP  group R8_DATA

R8_TEXT segment byte public 'CODE'
        assume  CS:R8_TEXT
        assume  SS:DGROUP
        public  R8
R8      proc    far
        mov     DX,word ptr SS:R8Val
```

```
        mov     CX,word ptr SS:R8Val+2
        mov     BX,word ptr SS:R8Val+4
        mov     AX,word ptr SS:R8Val+6
        ret
R8      endp
R8_TEXT ends
R8_DATA segment byte public 'DATA'
R8Val   dq 103.3
R8_DATA ends

        end
```

8. A **COMPLEX*8** function.

```
.8087

DGROUP  group C8_DATA

C8_TEXT segment byte public 'CODE'
        assume  CS:C8_TEXT
        assume  SS:DGROUP
        public  C8
C8      proc    far
        push    DI
        push    ES
        xchg    DI,SI
        push    SS
        pop     ES
        mov     SI,offset SS:C8Val
        movsw
        movsw
        movsw
        movsw
        pop     ES
        pop     DI
        ret
C8      endp
C8_TEXT ends

C8_DATA segment byte public 'DATA'
C8Val   dd 2.2
        dd 2.2
C8_DATA ends

        end
```

9. A **COMPLEX*16** function.

```
.8087

DGROUP  group C16_DATA

C16_TEXT segment byte public 'CODE'
        assume  CS:C16_TEXT
        assume  SS:DGROUP
        public  C16
C16     proc    far
        push    DI
        push    ES
        push    CX
        xchg    DI,SI
        push    SS
        pop     ES
        mov     SI,offset SS:C16Val
        mov     CX,8
        repe    movsw
        pop     CX
        pop     ES
        pop     DI
        ret
```

```
C16     endp
C16_TEXT ends

C16_DATA segment byte public 'DATA'
C16Val  dq 3.3
        dq 3.3
C16_DATA ends

        end
```

10. A **CHARACTER** function.

```
CHR_TEXT segment byte public 'CODE'
        assume  CS:CHR_TEXT
        public  CHR
CHR     proc    far
        push    DI
        push    ES
        mov     ES,DX
        mov     DI,AX
        les     DI,ES:[DI]
        mov     byte ptr ES:[DI],'F'
        pop     ES
        pop     DI
        ret
CHR     endp
CHR_TEXT ends

        end
```

11. A function returning a user-defined structure.

```
DGROUP  group STRUCT_DATA

STRUCT_TEXT segment byte public 'CODE'
        assume  CS:STRUCT_TEXT
        assume  SS:DGROUP
        public  C16
STRUCT  proc    far
        push    DI
        push    ES
        push    CX
        xchg    DI,SI
        push    SS
        pop     ES
        mov     SI,offset SS:StructVal
        mov     CX,4
        repe    movsw
        pop     CX
        pop     ES
        pop     DI
        ret
STRUCT  endp
STRUCT_TEXT ends

STRUCT_DATA segment byte public 'DATA'
StructVal dd 7
          dd 3
STRUCT_DATA ends

        end
```

If you are using an 80x87 to return floating-point values, only assembly language functions of type **REAL*4** and **REAL*8** need to be modified.

1. A **REAL*4** function using an 80x87.

```
.8087

DGROUP  group R4_DATA

R4_TEXT segment byte public 'CODE'
        assume  CS:R4_TEXT
        assume  SS:DGROUP
        public  R4
R4      proc    far
        fld     dword ptr SS:R4Val
        ret
R4      endp
R4_TEXT ends
R4_DATA segment byte public 'DATA'
R4Val   dd 1314.3
R4_DATA ends

        end
```

2. A **REAL*8** function using an 80x87.

```
.8087

DGROUP  group R8_DATA

R8_TEXT segment byte public 'CODE'
        assume  CS:R8_TEXT
        assume  SS:DGROUP
        public  R8
R8      proc    far
        fld     qword ptr SS:R8Val
        ret
R8      endp
R8_TEXT ends
R8_DATA segment byte public 'DATA'
R8Val   dq 103.3
R8_DATA ends

        end
```

Notes:

1. The ".8087" pseudo-op must be specified so that all floating-point constants are generated in 8087 format.

2. When returning values on the stack, remember to use a segment override to the stack segment (SS).

The following is an example of a Open Watcom F77 program calling the above assembly language subprograms.

```
logical l1*1, l4*4
integer i1*1, i2*2, i4*4
real r4*4, r8*8
complex c8*8, c16*16
character chr
structure /coord/
    integer x, y
end structure
record /coord/ struct
print *, l1()
print *, l4()
print *, i1()
print *, i2()
print *, i4()
print *, r4()
print *, r8()
print *, c8()
print *, c16()
print *, chr()
print *, struct()
end
```

11 16-bit Pragmas

11.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas can be used to direct the Open Watcom F77 code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Open Watcom F77.

- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the FORTRAN 77 language level. The code generator can use this information to generate more efficient code.

- Any sequence of in-line machine language instructions, including DOS and BIOS function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. A pragma operator of the form, ***_Pragma*** ("string-literal") is an alternative method of specifying ***pragma*** directives.

For example, the following two statements are equivalent.

```
_Pragma( "library (\"kernel32.lib\")" )
#pragma library ("kernel32.lib")
```

The ***_Pragma*** operator can be used in macro definition.

```
# define LIBRARY(X) PRAGMA(library (#X))
# define PRAGMA(X) _Pragma(#X)
LIBRARY(kernel32.lib)  // same as #pragma library ("kernel32.lib")
```

The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc]	The item *abc* is optional.
{abc}	The item *abc* may be repeated zero or more times.
a|b|c	One of *a, b* or *c* may be specified.
a ::= b	The item *a* is defined in terms of *b*.
(a)	Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify default libraries
- pragmas that provide auxiliary information used for code generation

11.2 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

The backslash character ('\') is used to continue a pragma on the next line. Text following the backslash character is ignored. The line continuing the pragma must start with a comment character ('c', 'C' or '*').

11.2.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
*$pragma aux y <attrs_1>
code in which symbol y is referenced
code in which symbol z is referenced
*$pragma aux default <attrs_2>
*$pragma aux x <attrs_3>
```

Auxiliary attributes are assigned to `x`, `y` and `z` in the following way.

1. Symbol `x` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>` and `<attrs_3>`.

2. Symbol `y` is assigned the initial default attributes merged with the attributes specified by `<attrs_1>`.

3. Symbol `z` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>`.

11.2.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
*$pragma aux ( sym, alias )
```

where ***description***

sym is any valid FORTRAN 77 identifier.

alias is the alias name and is any valid FORTRAN 77 identifier.

Consider the following example.

```
*$pragma aux value_args parm (value)
*$pragma aux ( rtn, value_args )
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed by value.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
*$pragma aux ( alias ) sym aux_attrs
```

where ***description***

alias is the alias name and is any valid FORTRAN 77 identifier.

sym is any valid FORTRAN 77 identifier.

aux_attrs are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
*$pragma aux WC "*_" parm (value)
*$pragma aux (WC) rtn1
*$pragma aux (WC) rtn2
*$pragma aux (WC) rtn3
```

The routines `rtn1`, `rtn2` and `rtn3` assume the same attributes as the alias name `WC` which defines the calling convention used by the Open Watcom C compiler. Whenever calls are made to `rtn1`, `rtn2` and `rtn3`, the Open Watcom C calling convention will be used. Note that arguments must be passed by value. By default, Open Watcom F77 passes arguments by reference.

Note that if the attributes of `WC` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1`, `rtn2` and `rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

> ***WARNING!*** The alias name `WC` is just another symbol. If `WC` appeared in your source code, it would assume the attributes specified in the pragma for `WC`.

11.2.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` defines the calling convention used by Microsoft compilers.

__fastcall `__fastcall` defines the calling convention used by Microsoft compilers.

__fortran `__fortran` defines the calling convention used by Open Watcom FORTRAN compilers.

__pascal `__pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

__stdcall `__stdcall` defines the calling convention used by Microsoft compilers.

__watcall `__watcall` defines the calling convention used by Open Watcom compilers.

The following describes the attributes of the above alias names.

11.2.3.1 Predefined "__cdecl" Alias

```
*$pragma aux __cdecl "_*" \
c           parm caller [] \
c           value struct float struct routine [ax] \
c           modify [ax bx cx dx es]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register AX.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

11.2.3.2 Predefined "__pascal" Alias

```
*$pragma aux __pascal "^" \
c           parm reverse routine [] \
c           value struct float struct caller [] \
c           modify [ax bx cx dx es]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register AX will contain address of the space allocated for the return value.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

11.2.3.3 Predefined "__watcall" Alias

```
*$pragma aux __watcall "*_" \
c           parm routine [ax bx cx dx] \
c           value struct caller
```

Notes:

1. Symbol names are followed by an underscore character.

2. Arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left. The calling routine will remove the arguments if any were pushed on the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space is put into SI register. The called routine then places the return value there. Upon returning from the call, register AX will contain address of the space allocated for the return value.

4. Floating-point values are returned using 80x86 registers ("fpc" option) or using 80x87 floating-point registers ("fpi" or "fpi87" option).

5. All registers must be preserved by the called routine.

11.2.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
*$pragma aux sym obj_name
```

where	*description*
sym	is any valid FORTRAN 77 identifier.
obj_name	is any character string enclosed in double quotes.

When specifying `obj_name`, some characters have a special meaning:

where	*description*
*	is unmodified symbol name
^	is symbol name converted to uppercase
!	is symbol name converted to lowercase
#	is a placeholder for "@nnn", where nnn is size of all function parameters on the stack; it is ignored for functions with variable argument lists, or for symbols that are not functions
\	next character is treated as literal

Several examples of source to object form symbol name translation follow: By default, the upper case version "MYRTN" or "MYVAR" is placed in the object file.

In the following example, the name "MyRtn" will be replaced by "MYRTN_" in the object file.

```
*$pragma aux MyRtn "^_"
```

In the following example, the name "MyVar" will be replaced by "_MYVAR" in the object file.

```
*$pragma aux MyVar "_^"
```

In the following example, the lower case version "myrtn" will be placed in the object file.

```
*$pragma aux MyRtn "!"
```

In the following example, the name "MyRtn" will be replaced by "_MyRtn@nnn" in the object file. "nnn" represents the size of all function parameters.

```
*$pragma aux MyRtn "_*#"
```

In the following example, the name "MyRtn" will be replaced by "_MyRtn#" in the object file.

```
*$pragma aux MyRtn "_*\#"
```

The default mapping for all symbols can also be changed as illustrated by the following example.

```
*$pragma aux default "_^_"
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

11.2.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a subprogram is to be called.

```
*$pragma aux sym far
    or
*$pragma aux sym near
    or
*$pragma aux sym = in_line

in_line ::= { const | "asm" | (float fpinst) }
```

where	*description*
sym	is a subprogram name.
const	is a valid FORTRAN 77 hexadecimal constant.
fpinst	is a sequence of bytes that forms a valid 80x87 instruction. The keyword ***float*** must precede `fpinst` so that special fixups are applied to the 80x87 instruction.
asm	is an assembly language instruction or directive.

In the following example, Open Watcom F77 will generate a far call to the subprogram `myrtn`.

```
*$pragma aux myrtn far
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Open Watcom F77 will generate a near call to the subprogram `myrtn`.

```
*$pragma aux myrtn near
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Open Watcom F77 will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line subprogram.

```
*$pragma aux mode4 =    \
*    zb4 z00            \ mov AH,0
*    zb0 z04            \ mov AL,4
*    zcd z10            \ int 10h
*    modify [ AH AL ]
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0       ; select function "set mode"
mov   AL,4       ; specify mode (mode 4)
int   10H        ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your FORTRAN 77 program.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
*$pragma aux mode4 =    \
*    "mov AH,0"         \
*    "mov AL,4"         \
*    "int 10H"          \
*    modify [ AH AL ]
```

If a sequence of in-line assembly language instructions contains 80x87 floating-point instructions, each floating-point instruction must be preceded by "float". Note that this is only required if you have specified the "fpi" compiler option; otherwise it will be ignored.

The following example generates the 80x87 "square root" instruction.

```
*$pragma aux mysqrt parm( value ) [8087] = \
*            float zd9fa
```

11.2.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a subprogram that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified subprogram.

```
*$pragma aux sym parm loadds
```

where ***description***

sym is a subprogram name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified subprogram.

```
*$pragma aux sym loadds
```

where ***description***

sym is a subprogram name.

11.2.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link libraries are exported using the Open Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
*$pragma aux sym export
```

where ***description***

sym is a subprogram name.

11.2.5.3 Defining Windows Callback Functions

When compiling a Microsoft Windows application, you must use the "windows" option so that special prologue/epilogue sequences are generated. Furthermore, callback functions require larger prologue/epilogue sequences than those generated when the "windows" compiler option is specified. The following form of the auxiliary pragma will cause a callback prologue/epilogue sequence to be generated for a callback function when compiled using the "windows" option.

```
*$pragma aux sym export
```

where ***description***

sym is a callback function name.

11.2.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Open Watcom F77 is to use for calling subprograms. This is particularly useful when interfacing to subprograms that have been compiled by other compilers or subprograms written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
*$pragma aux sym parm { arg_info | pop_info | reverse {reg_set} }

arg_info ::= ( arg_attr {, arg_attr} )

arg_attr ::= value [v_attr]
                 | reference [r_attr]
                 | data_reference [d_attr]

v_attr ::= far | near | *1 | *2 | *4 | *8

r_attr ::= [far | near] [descriptor | nodescriptor]

d_attr ::= [far | near]

pop_info ::= caller | routine
```

where	*description*
sym	is a subprogram name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.2.6.1 Passing Arguments to non-FORTRAN Subprograms

When calling a subprogram written in a different language, it may be necessary to provide the arguments in a form different than the default methods used by Open Watcom F77. For example, C functions require scalar arguments to be passed by value instead of by reference. For information on the methods Open Watcom F77 uses to pass arguments, see the chapter entitled "Assembly Language Considerations".

The following form of the auxiliary pragma can be used to alter the default calling mechanism used for passing arguments.

```
*$pragma aux sym parm ( arg_attr {, arg_attr} )

arg_attr ::= value [v_attr]
                 | reference [r_attr]
                 | data_reference [d_attr]

v_attr ::= far | near | *1 | *2 | *4 | *8

r_attr ::= [far | near] [descriptor | nodescriptor]

d_attr ::= [far | near]
```

where ***description***

sym is a subprogram name.

REFERENCE specifies that arguments are to be passed by reference. For non-character arguments, the address is a pointer to the data. For character arguments, the address is a pointer to a string descriptor. See the chapter entitled "Assembly Language Considerations" for a description of a string descriptor. This is the default calling mechanism. If "NEAR" or "FAR" is specified, a near pointer or far pointer is passed regardless of the memory model used at compile-time.

If the "DESCRIPTOR" attribute is specified, a pointer to the string descriptor is passed. This is the default. If the "NODESCRIPTOR" attribute is specified, a pointer to the the actual character data is passed instead of a pointer to the string descriptor.

DATA_REFERENCE specifies that arguments are to be passed by data reference. For non-character items, this is identical to passing by reference. For character items, a pointer to the actual character data (instead of the string descriptor) is passed. If "NEAR" or "FAR" is specified, a near pointer or far pointer is passed regardless of the memory model used at compile-time.

VALUE specifies that arguments are to be passed by value. Character arguments are treated specially when passed by value. Instead of passing a pointer to a string descriptor, a pointer to the actual character data is passed. See the chapter entitled "Assembly Language Considerations" for a description of a string descriptor.

Notes:

1. Arrays and subprograms are always passed by reference, regardless of the argument attribute specified.

2. When character arguments are passed by reference, the address of a string descriptor is passed. The string descriptor contains the address of the actual character data and the number of characters. When character arguments are passed by value or data reference, the address of the actual character data is passed instead of the address of a string descriptor. Character arguments are passed by value by specifying the "VALUE" or "DATA_REFERENCE" attribute. If "NEAR" or "FAR" is specified, a near pointer or far pointer to the character data is passed regardless of the memory model used at compile-time.

3. When complex arguments are passed by value, the real part and the imaginary part are passed as two separate arguments.

4. When an argument is a user-defined structure and is passed by value, a copy of the structure is made and passed as an argument.

5. For scalar arguments, arguments of type **INTEGER*1**, **INTEGER*2**, **INTEGER*4** ct , **REAL** or **DOUBLE PRECISION**, a length specification can be specified when the "VALUE" attribute is specified to pass the argument by value. This length specification refers to the size of the argument; the compiler will convert the actual argument to a type that matches the size. For example, if an argument of type **REAL** is passed to a subprogram that has an argument attribute of "VALUE*8", the argument will be converted to **DOUBLE PRECISION**. If an argument of type **DOUBLE PRECISION** is passed to a subprogram that has an argument attribute of "VALUE*4", the argument will be converted to **REAL**. If an argument of type **INTEGER*4** is passed to a subprogram that has an argument attribute of "VALUE*2" or VALUE*1, the argument will be converted to **INTEGER*2** or **INTEGER*1**. If an argument of type **INTEGER*2** is passed to a subprogram that has an argument attribute of "VALUE*4 or VALUE*1", the argument will be converted to **INTEGER*4** or **INTEGER*1**. If an argument of type **INTEGER*1** is passed to a subprogram that has an argument attribute of "VALUE*4 or VALUE*2", the argument will be converted to **INTEGER*4** or **INTEGER*2**.

6. If the number of arguments exceeds the number of entries in the argument-attribute list, the last attribute will be assumed for the remaining arguments.

Consider the following example.

```
*$pragma aux printf "*_" parm (value) caller []
      character cr/z0d/, nullchar/z00/
      call printf( 'values: %ld, %ld'//cr//nullchar,
     1             77, 31410 )
      end
```

The C "printf" function is called with three arguments. The first argument is of type **CHARACTER** and is passed as a C string (address of actual data terminated by a null character). The second and third arguments are passed by value. Also note that "printf" is a function that takes a variable number of arguments, all passed on the stack (an empty register set was specified), and that the caller must remove the arguments from the stack.

11.2.6.2 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular subprogram.

```
*$pragma aux sym parm {reg_set}
```

where	*description*
sym	is a subprogram name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **REAL** and **DOUBLE PRECISION** are always pushed on the stack when the "fpi" or "fpi87" option is used.

DOUBLE PRECISION

Arguments of type **DOUBLE PRECISION**, when passed by value, can only be passed in the following register combination: AX:BX:CX:DX. For example, if the following register set was specified for a routine having an argument of type **DOUBLE PRECISION**,

```
[AX BX SI DI]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **DOUBLE PRECISION** is supported only when the "fpc" option is used. Note that this argument passing method does not include arguments of type **COMPLEX*8** or user-defined structures whose size is 8 bytes when these arguments are passed by value.

far pointer

A far pointer can only be passed in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. For example, if a far pointer is passed to a function with the following register set,

```
[ES BP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set. Far pointers are used to pass arguments by reference in a big data memory model.

INTEGER*4, REAL

The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **INTEGER*4**, when passed by value) are: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX and BX:AX. For example, if the following register set was specified for a routine with one argument of type **INTEGER*4**,

```
[ES DI]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method includes arguments of type **REAL** but only when the "fpc" option is used.

INTEGER*2

The only registers that will be assigned to 2-byte arguments (e.g., arguments of type **INTEGER*2** when passed by value or arguments passed by reference in a small data memory model) are: AX, BX, CX, DX, SI and DI. For example, if the following register set was specified for a routine with one argument of type **INTEGER*2**,

```
[BP]
```

the argument would be pushed on the stack since a valid register combination for 2-byte arguments is not contained in the register set.

INTEGER*1 Arguments whose size is 1 byte (e.g., arguments of type **INTEGER*1** when passed by value) are promoted to 2 bytes and are then assigned registers as if they were 2-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [AX BX CX DX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX.

3. If you are compiling for a memory model with a small data model, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

Consider the following example.

```
*$pragma aux myrtn parm (value) \
*                       [ax bx cx dx] [bp si]
```

Suppose `myrtn` is a routine with 3 arguments each of type **INTEGER**. Note that the arguments are passed by value.

1. The first argument will be passed in the register pair DX:AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be pushed on the stack since BP:SI is not a valid register pair for arguments of type **INTEGER**.

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
*$pragma aux myrtn parm (value) \
*                       [ax bx cx dx] [si di]
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **INTEGER** and the second and third of type **INTEGER**. Note that all arguments are passed by value.

1. The first argument will be passed in the register AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be passed in the register set DI:SI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [AX BX CX DX] is used.

11.2.6.3 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a subprogram, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register SI, the second argument to be passed in register DI and the third argument to be passed in register CX, the following auxiliary pragma can be used.

```
*$pragma aux mycopy parm (value) \
*                        [SI] [DI] [CX]
      character*10  dst
      call mycopy( dst, '0123456789', 10 )
      ...
      end
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

11.2.6.4 Passing Arguments to In-Line Subprograms

For subprograms whose code is generated by Open Watcom F77 and whose argument list is described by an auxiliary pragma, Open Watcom F77 has some freedom in choosing how arguments are assigned to registers. Since the code for in-line subprograms is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Open Watcom F77 assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line subprogram called `scrollactivepgup`.

```
*$pragma aux scrollactivepgup =        \
*   "mov AH,6"                         \
*   "int 10h"                          \
*   parm (value)                       \
*        [ch] [cl] [dh] [dl] [al] [bh] \
*   modify [ah]
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Open Watcom F77 will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **INTEGER**, it would first be converted to **INTEGER*1** before assigning it to register CH. Similarly, if an in-line subprogram required its argument in register pair DX:AX and the argument was of type **INTEGER*2**, the argument would be converted to **INTEGER*4** before assigning it to register pair DX:AX.

In general, Open Watcom F77 assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **INTEGER*1**.
2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **INTEGER*2**.
3. A register set consisting of two 16-bit registers (4 bytes) is assigned a type of **INTEGER*4**.
4. A register set consisting of four 16-bit registers (8 bytes) is assigned a type of **DOUBLE PRECISION**.

If the size of an integer argument is larger than the size specified by the register set, the argument will be truncated to the required size. If the size of an integer argument is smaller than the size specified by the register set, the argument will be padded (to the left) with zeros.

11.2.6.5 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
*$pragma aux sym parm (caller | routine)
```

where ***description***

sym is a subprogram name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

Consider the following example. It describes the pragma required to call the C "printf" function.

```
*$pragma aux printf "*_" parm (value) caller []
      character cr/z0d/, nullchar/z00/
      call printf( 'value is %ld'//cr//nullchar,
     1             7143 )
      end
```

The first argument must be passed as a C string, a pointer to the actual character data terminated by a null character. By default, the address of a string descriptor is passed for arguments of type **CHARACTER**. See the chapter entitled "Assembly Language Considerations" for more information on string descriptors. The second argument is of type **INTEGER** and is passed by value. Also note that "printf" is a function that takes a variable number of arguments, all pushed on the stack (an empty register set was specified).

11.2.6.6 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
*$pragma aux sym parm reverse
```

where	*description*
sym	is a subprogram name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for subprograms that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a subprogram.

```
*$pragma aux rtn parm reverse []
```

11.2.7 Describing Subprogram Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
*$pragma aux sym value {no8087 | reg_set | struct_info}
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where	*description*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.2.7.1 Returning Subprogram Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
*$pragma aux sym value reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

Note that the method described below for returning values of type **REAL** or **DOUBLE PRECISION** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set.*

1-byte	For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.
2-byte	For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.
4-byte	For 4-byte return values (except far pointers), only the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, registers DX:AX will be used. This form of the auxiliary pragma is legal for functions of type **REAL** when using the "fpc" option only.
far pointer	For functions that return far pointers, the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. If no register set is specified, the registers DX:AX will be used.
8-byte	For 8-byte return values (including functions of type **DOUBLE PRECISION**), only the following register combination is allowed: AX:BX:CX:DX. If no register set is specified, the registers AX:BX:CX:DX will be used. This form of the auxiliary pragma is legal for functions of type **DOUBLE PRECISION** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

11.2.7.2 Returning Structures and Complex Numbers

Typically, structures and complex numbers are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register SI to point to it. The called routine then places the return value at the location pointed to by register SI.

Complex numbers are not scalars but rather an ordered pair of real numbers. One can also view complex numbers as a *structure* containing two real numbers.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
*$pragma aux sym value struct (caller | routine) reg_set
```

where ***description***

sym is a subprogram name.

reg_set is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register AX by the called routine. It is assumed that the memory for the return value is allocated from the stack segment (the stack segment is contained in segment register SS).

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: AX, DX, BX, CX, SI or DI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, register pair DX:AX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register SI to it.

```
*$pragma aux sym value struct struct
```

where	*description*
sym	is a subprogram name.

11.2.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **REAL** or **DOUBLE PRECISION** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register SI to it.

```
*$pragma aux sym value struct float
```

where	*description*
sym	is a function name.

In other words, floating-point values are to be returned in the same way complex numbers are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **REAL** will be returned in registers DX:AX. Function return values whose type is **DOUBLE PRECISION** will be returned in registers AX:BX:CX:DX. This is the default method for the "fpc" option.

```
*$pragma aux sym value no8087
```

where	*description*
sym	is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
*$pragma aux sym value [8087]
```

where	*description*
sym	is a function name.

11.2.8 A Subprogram that Never Returns

The following form of the auxiliary pragma can be used to describe a subprogram that does not return to the caller.

```
*$pragma aux sym aborts
```

where ***description***

sym is a subprogram name.

Consider the following example.

```
*$pragma aux exitrtn aborts
      ...
      call exitrtn()
      end
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Open Watcom F77 generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn`.

11.2.9 Describing How Subprograms Use Variables in Common

The following form of the auxiliary pragma can be used to describe a subprogram that does not modify any variable that appears in a common block defined by the caller.

```
*$pragma aux sym modify nomemory
```

where ***description***

sym is a subprogram name.

Consider the following example.

```
integer i
common /blk/ i
while( i .lt. 1000 )do
    i = i + 383
endwhile
call myrtn()
i = i + 13143
end

block data
common /blk/ i
integer i/1033/
end
```

To compile the above program, "rtn.for", we issue the following command.

```
C>wfc rtn -mm -d1
C>wfc386 rtn -d1
```

The "d1" compiler option is specified so that the object file produced by Open Watcom F77 contains source line information.

We can generate a file containing a disassembly of `rtn.obj` by issuing the following command.

```
C>wdis rtn -l -s -r
```

The "s" option is specified so that the listing file produced by the Open Watcom Disassembler contains source lines taken from `rtn.for`. The listing file `rtn.lst` appears as follows.

Let us add the following auxiliary pragma to the source file.

```
*$pragma aux myrtn modify nomemory
```

If we compile the source file with the above pragma and disassemble the object file using the Open Watcom Disassembler, we get the following listing file.

```
Module: rtn.for
Group: 'DGROUP' _DATA,LDATA,CDATA,BLK

Segment: 'FMAIN_TEXT' BYTE  00000024 bytes

*$pragma aux myrtn modify nomemory
        integer*2 i
        common /blk/ i
 0000  52                FMAIN           push    dx
 0001  8b 16 00 00                       mov     dx,L3

        while( i .lt. 1000 )do
 0005  81 fa e8 03       L1              cmp     dx,03e8H
 0009  7d 06                             jge     L2

            i = i + 383
        endwhile
 000b  81 c2 7f 01                       add     dx,017fH
 000f  eb f4                             jmp     L1

        call myrtn()
 0011  89 16 00 00       L2              mov     L3,dx
 0015  9a 00 00 00 00                    call    far MYRTN

        i = i + 13143
 001a  81 c2 57 33                       add     dx,3357H
 001e  89 16 00 00                       mov     L3,dx

        end

        block data
        common /blk/ i
        integer*2 i/1033/
        end
 0022  5a                                pop     dx
 0023  cb                                retf

No disassembly errors
```

```
List of external symbols

Symbol
----------------
MYRTN            00000016
------------------------------------------------------------

Segment: 'BLK' PARA  00000002 bytes
 0000  09 04                  L3             - ..

No disassembly errors

------------------------------------------------------------
List of public symbols

SYMBOL          GROUP           SEGMENT          ADDRESS
-----------------------------------------------------------
FMAIN                           FMAIN_TEXT       00000000

------------------------------------------------------------
```

Notice that the value of `i` is in register DX after completion of the "while" loop. After the call to `myrtn`, the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any variable that appears in a common block defined by `Rtn` and hence register DX contains the correct value of `i`.

The preceding auxiliary pragma deals with routines that modify variables in common. Let us consider the case where routines reference variables in common. The following form of the auxiliary pragma can be used to describe a subprogram that does not reference any variable that appears in a common block defined by the caller.

```
*$pragma aux sym parm nomemory modify nomemory
```

where ***description***

sym is a subprogram name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
*$pragma aux myrtn parm nomemory modify nomemory
```

If you now compile our source file and disassemble the object file using WDIS, the result is the following listing file.

```
Module: rtn.for
Group: 'DGROUP' _DATA,LDATA,CDATA,BLK

Segment: 'FMAIN_TEXT' BYTE  00000020 bytes

*$pragma aux myrtn parm nomemory modify nomemory
        integer*2 i
        common /blk/ i
 0000  52               FMAIN           push    dx
 0001  8b 16 00 00                      mov     dx,L3
```

```
       while( i .lt. 1000 )do
 0005  81 fa e8 03       L1              cmp     dx,03e8H
 0009  7d 06                             jge     L2

           i = i + 383
       endwhile
 000b  81 c2 7f 01                       add     dx,017fH
 000f  eb f4                             jmp     L1

       call myrtn()
 0011  9a 00 00 00 00    L2              call    far MYRTN

       i = i + 13143
 0016  81 c2 57 33                       add     dx,3357H
 001a  89 16 00 00                       mov     L3,dx

       end

       block data
       common /blk/ i
       integer*2 i/1033/
       end
 001e  5a                                pop     dx
 001f  cb                                retf

No disassembly errors

List of external symbols

Symbol
----------------
MYRTN            00000012
------------------------------------------------------------

Segment: 'BLK' PARA  00000002 bytes
 0000  09 04                  L3              - ..

No disassembly errors

------------------------------------------------------------
List of public symbols

SYMBOL          GROUP           SEGMENT          ADDRESS
------------------------------------------------------------
FMAIN                           FMAIN_TEXT       00000000

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register DX before calling `myrtn`. The auxiliary pragma informs the compiler that `myrtn` does not reference any variable that appears in a common block defined by `myrtn` so updating `i` was not necessary before calling `myrtn`.

11.2.10 Describing the Registers Modified by a Subprogram

The following form of the auxiliary pragma can be used to describe the registers that a subprogram will use without saving.

```
*$pragma aux sym modify [exact] reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

Specifying a register set informs Open Watcom F77 that the registers belonging to the register set are modified by the subprogram. That is, the value in a register before calling the subprogram is different from its value after execution of the subprogram.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called subprogram. Also, since the AX register is frequently used to return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called subprogram does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called subprogram.

```
*$pragma aux sym modify exact reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

The above form of the auxiliary pragma tells Open Watcom F77 not to assume that the registers used to pass arguments will be modified by the called subprogram. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:

```
*$ifdef __386__
*$pragma aux GetSP = value [esp] modify exact []
*$else
*$pragma aux GetSP = value [sp] modify exact []
*$endif

      program main
      integer GetSP
      print *, 'Current SP =', GetSP()
      end
```

11.2.11 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile subprograms. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

11.2.11.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a subprogram is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to subprograms.

```
*$pragma aux sym parm {reg_set}
```

where ***description***

sym is a subprogram name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

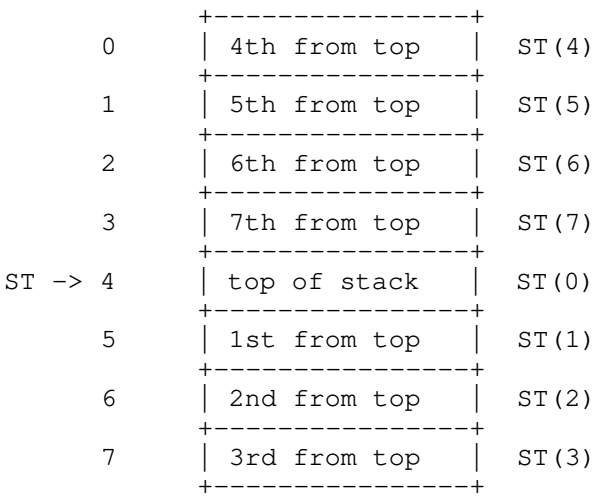
Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).



Starting with version 9.5, the Open Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
*$pragma aux myrtn parm (value) [8087]

      real x
      double precision y
      integer*2 i
      integer j
      x = 7.7
      i = 7
      y = 77.77
      j = 77
      call myrtn( x, i, y, j )
      end
```

`myrtn` is an assembly language subprogram that requires four arguments. The first argument of type **REAL** (4 bytes), the second argument is of type **INTEGER*2** (2 bytes), the third argument is of type **DOUBLE PRECISION** (8 bytes) and the fourth argument is of type **INTEGER*4** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **REAL**, will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
*$pragma aux myrtn parm [ax 8087]
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **REAL** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register AX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **DOUBLE PRECISION**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

11.2.11.2 Using the 80x87 to Return Subprogram Values

The following form of the auxiliary pragma can be used to describe a subprogram that returns a floating-point value in ST(0).

```
*$pragma aux sym value reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set containing the string "8087", i.e. [8087].

11.2.11.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a subprogram unless the "fpr" option is used to generate backward compatible code (older Open Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified subprogram.

```
*$pragma aux sym modify reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set containing the string "8087", i.e. [8087].

This instructs Open Watcom F77 to save any local variables that are located in the 80x87 cache before calling the specified routine.

32-bit Topics

12 32-bit Memory Models

12.1 Introduction

This chapter describes the various 32-bit memory models supported by Open Watcom F77. Each memory model is distinguished by two properties; the code model used to implement subprogram calls and the data model used to reference data.

12.2 32-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to subprograms are made with *near calls.* In a near call, the destination address is 32 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library subprograms, must be less than 4GB.

A big code model is one in which all calls to subprograms are made with *far calls.* In a far call, the destination address is 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This model allows the size of the code comprising your program to exceed 4GB.

> ***Note:*** If your program contains less than 4GB of code, you should use a memory model that employs the small code model. This will result in smaller and faster code since near calls are smaller instructions and are processed faster by the CPU.

12.3 32-bit Data Models

There are two data models;

1. the small data model and
2. the big data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 32 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 4GB.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This removes the 4GB limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Open Watcom F77 assumes that the offset portion of a far pointer will not be incremented beyond 4GB. The compiler will assign an object to a new segment if the grouping of data in a segment will

cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 4GB.

> ***Note:*** If your program contains less than 4GB of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

12.4 Summary of 32-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Open Watcom F77.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
flat	small	small	near	near
small	small	small	near	near
medium	big	small	far	near
compact	small	big	near	far
large	big	big	far	far

12.5 Flat Memory Model

In the flat memory model, the application's code and data must total less than 4GB in size. Segment registers CS, DS, SS and ES point to the same linear address space (this does not imply that the segment registers contain the same value). That is, a given offset in one segment refers to the same memory location as that offset in another segment. Essentially, a flat model operates as if there were no segments.

12.6 Mixed 32-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that includes arrays which are larger than 4GB.

For example, a medium memory model application that uses some arrays which exceed 4GB in total size can be described as a mixed memory model. In an application such as this, most of the data is in a 4GB segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model.

12.7 Linking Applications for the Various 32-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model. Currently, only libraries for the flat/small memory model are provided. The following table lists the run-time libraries used by FORTRAN 77 and the compiler options that cause their use.

1. The "Library" column specified the library name.
2. The "Memory model" column indicates the compiler options that specify the memory model of the library.
3. The "Floating-point column" indicates the compiler options that specify the floating-point model of the library.
4. The "Calling convention" column indicates the compiler option that specifies the calling convention of the library (register-based or stack-based).

Library	Memory model	Floating-point model	Calling convention
flib.lib	/mf, /ms	/fpc	
flibs.lib	/mf, /ms	/fpc	/sc
flib7.lib	/mf, /ms	/fpi, /fpi87	
flib7s.lib	/mf, /ms	/fpi, /fpi87	/sc
clib3r.lib	/mf, /ms	/fpc, /fpi, /fpi87	
clib3r.lib	/mf, /ms	/fpc, /fpi, /fpi87	/sc
math387r.lib	/mf, /ms	/fpi, /fpi87	
math387s.lib	/mf, /ms	/fpi, /fpi87	/sc
math3r.lib	/mf, /ms	/fpc	
math3s.lib	/mf, /ms	/fpc	/sc
emu387.lib	/mf, /ms	/fpi	
noemu387.lib	/mf, /ms	/fpi87	

12.8 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom F77.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<subprogram>_TEXT" where <subprogram> is the name of a subprogram.

2. The "FAR_DATA" class consists of the following:

   (a) arrays whose size exceeds the data threshold in large data memory models (the data threshold is 256 bytes unless changed using the "dt" compiler option)

   (b) equivalenced variables in large data memory models

13 32-bit Assembly Language Considerations

13.1 Introduction

This chapter will deal with the following topics.

1. The memory layout of a program compiled by Open Watcom F77.
2. The method for passing arguments and returning values.
3. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Open Watcom F77 "fpi", "fpi87" or "fpi287" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" or "fpi287" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Open Watcom F77 "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

13.2 Calling Conventions

The following sections describe the method used by Open Watcom F77 to pass arguments.

The FORTRAN 77 language specifically requires that arguments be passed by reference. This means that instead of passing the value of an argument, its address is passed. This allows a called subprogram to modify the value of the actual arguments. The following illustrates the method used to pass arguments.

Type of Argument	Method Used to Pass Argument
non-character constant	address of constant
non-character expression	address of value of expression
non-character variable	address of variable
character constant	address of string descriptor
character expression	address of string descriptor
character variable	address of string descriptor
non-character array	address of array
non-character array element	address of array
character array	address of string descriptor
character array element	address of string descriptor
character substring	address of string descriptor
subprogram	address of subprogram
alternate return specifier	no argument passed
user-defined structure	address of structure

When passing a character array as an argument, the string descriptor contains the address of the first element of the array and the length of an element of the array.

The address of arguments are either passed in registers or on the stack. The registers used to pass the address of arguments to a subprogram are EAX, EBX, ECX and EDX. The address of arguments are passed in the following way.

1. The first argument is passed in register EAX, the second argument is passed in register EDX, the third argument is passed in register EBX, and the fourth argument is passed in register ECX.

2. For any remaining arguments, their address is passed on the stack. Note that addresses of arguments are pushed on the stack from right to left.

13.2.1 Stack-Based Calling Convention

The previous section described a register-based calling convention in which registers were used to pass arguments to subprograms. A stack-based calling convention is another method that can be used to pass arguments. The calling convention is selected when the "sc" compiler option is specified.

The most significant difference between the stack-based calling convention and the register-based calling convention is the way the arguments are passed. When using the stack-based calling conventions, no registers are used to pass arguments. Instead, all arguments are passed on the stack.

13.2.2 Processing Function Return Values with no 80x87

The way in which function values are returned is also dependent on the data type of the function. The following describes the method used to return function values.

1. **LOGICAL*1** values are returned in register AL.

2. **LOGICAL*4** values are returned in register EAX.

3. **INTEGER*1** values are returned in register AL.

4. **INTEGER*2** values are returned in register AX.

5. **INTEGER*4** values are returned in register EAX.

6. **REAL*4** values are returned in register EAX.

7. **REAL*8** values are returned in registers EDX:EAX.

8. For **COMPLEX*8** functions, space is allocated on the stack by the caller for the return value. Register ESI is set to point to the destination of the result. The called function places the result at the location pointed to by register ESI.

9. For **COMPLEX*16** functions, space is allocated on the stack by the caller for the return value. Register ESI is set to point to the destination of the result. The called function places the result at the location pointed to by register ESI.

10. For **CHARACTER** functions, an additional argument is passed. This argument is the address of the string descriptor for the result. Note that the address of the string descriptor can be passed in any of the registers that are used to pass actual arguments.

11. For functions that return a user-defined structure, space is allocated on the stack by the caller for the return value. Register ESI is set to point to the destination of the result. The called function places the result at the location pointed to by register ESI. Note that a structure of size 1, 2 or 4 bytes is returned in register AL, AX or EAX respectively.

> ***Note:*** The way in which a function returns its value does not change when the stack-based calling convention is used.

13.2.3 Processing Function Return Values Using an 80x87

The following describes the method used to return function values when your application is compiled using the "fpi87" or "fpi" option.

1. For **REAL*4** functions, the result is returned in floating-point register ST(0).
2. For **REAL*8** functions, the result is returned in floating-point register ST(0).
3. All other function values are returned in the way described in the previous section.

> ***Note:*** When the stack-based calling convention is used, floating-point values are not returned using the 80x87. **REAL*4** values are returned in register EAX. **REAL*8** values are returned in registers EDX:EAX.

13.2.4 Processing Alternate Returns

Alternate returns are processed by the caller and are only allowed in subroutines. The called subroutine places the value specified in the ***RETURN*** statement in register EAX. Note that the value returned in register EAX is ignored if there are no alternate return specifiers in the actual argument list.

> ***Note:*** The way in which a alternate returns are processed does not change when the stack-based calling convention is used.

13.2.5 Alternate Method of Passing Character Arguments

As previously described, character arguments are passed using string descriptors. Recall that a string descriptor contains a pointer to the actual character data and the length of the character data. When passing character data, both a pointer and length are required by the subprogram being called. When using a string descriptor, this information can be passed using a single argument, namely the pointer to the string descriptor.

An alternate method of passing character arguments is also supported and is selected when the "nodescriptor" option is specified. In this method, the pointer to the character data and the length of the character data are passed as two separate arguments. The character argument lengths are appended to the end of the actual argument list.

Let us consider the following example.

```
INTEGER A, C
CHARACTER B, D
CALL SUB( A, B, C, D )
```

In the above example, the first argument is of type INTEGER, the second argument is of type CHARACTER, the third argument is of type INTEGER, and the fourth argument is of type CHARACTER. If the character arguments were passed by descriptor, the argument list would resemble the following.

1. The first argument would be the address of `A`.
2. The second argument would be the address of the string descriptor for `B`.
3. The third argument would be the address of `C`.
4. The fourth argument would be the address of the string descriptor for `D`.

If we specified the "nodescriptor" option, the argument list would be as follows.

1. The first argument would be the address of `A`.
2. The second argument would be the address of the character data for `B`.
3. The third argument would be the address of `C`.
4. The fourth argument would be the address of the character data for `D`.
5. A hidden argument for the length of `B` would be the fifth argument.
6. A hidden argument for the length of `D` would be the sixth argument.

Note that the arguments corresponding to the length of the character arguments are passed as INTEGER*4 arguments.

13.2.5.1 Character Functions

By default, when a character function is called, a hidden argument is passed at the end of the actual argument list. This hidden argument is a pointer to the string descriptor used for the return value of the character function. When the alternate method of passing character arguments is specified by using the "nodescriptor" option, the string descriptor for the return value is passed to the function as two hidden arguments, similar to the way character arguments were passed. However the two hidden arguments for the

return value of the character function are placed at the beginning of the actual argument list. The first argument is the the pointer to the storage immediately followed by the size of the storage.

13.3 Memory Layout

The following describes the segment ordering of an application linked by the Open Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Open Watcom F77.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<subprogram>_TEXT" where <subprogram> is the name of a subprogram.

2. The "FAR_DATA" class consists of the following:

   (a) arrays whose size exceeds the data threshold in large data memory models (the data threshold is 256 bytes unless changed using the "dt" compiler option)

   (b) equivalenced variables in large data memory models

13.4 Writing Assembly Language Subprograms

When writing assembly language subprograms, use the following guidelines.

1. All used registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to be saved and restored if you are compiling your application with the "sr" option.

2. The direction flag must be clear before returning to the caller.

3. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, register CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

4. In a small data model, segment register DS contains the segment address of the default data segment (group "DGROUP"). In a big data model, segment register SS (not DS) contains the segment address of the default data segment (group "DGROUP").

5. When writing assembly language subprograms for the small code model, you must declare them as "near". If you wish to write assembly language subprograms for the big code model, you must declare them as "far".

6. Use the ".8087" pseudo-op so that floating-point constants are in the correct format.

7. The called subprogram must remove arguments that were passed on the stack in the "ret" instruction.

8. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

Consider the following example.

```
      INTEGER HRS, MINS, SECS, HSECS
      CALL GETTIM( HRS, MINS, SECS, HSECS )
      PRINT 100, HRS, MINS, SECS, HSECS
100   FORMAT( 1X,I2.2,':',I2.2,':',I2.2,'.',I2.2 )
      END
```

`GETTIM` is an assembly language subroutine that gets the current time. It requires four integer arguments. The arguments are passed by reference so that the subroutine can return the hour, minute, seconds and hundredths of a second for the current time. These arguments will be passed to `GETTIM` in the following way.

1. The address of the first argument will be passed in register EAX.
2. The address of the second argument will be passed in register EDX.
3. The address of the third argument will be passed in register EBX.
4. The address of the fourth argument will be passed in register ECX.

The following is an assembly language subprogram which implements `GETTIM`.

Small or Flat Memory Model (small code, small data)

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  GETTIM
GETTIM  proc    near
        push    EAX       ; save registers modified by
        push    ECX       ; ... DOS function call
        push    EDX       ; ...
        mov     AH,2ch    ; set DOS "get time" function
        int     21h       ; issue DOS function call
        movzx   EAX,DH    ; get seconds
        mov     [EBX],EAX ; return seconds
        pop     EBX       ; get address of minutes
        movzx   EAX,CL    ; get minutes
        mov     [EBX],EAX ; return minutes
        pop     EBX       ; get address of ticks
        movzx   EAX,DL    ; get ticks
        mov     [EBX],EAX ; return ticks
        pop     EBX       ; get address of hours
        movzx   EAX,CH    ; get hours
        mov     [EBX],EAX ; return hours
        ret               ; return
GETTIM  endp
_TEXT   ends

        end
```

Notes:

1. No arguments were passed on the stack so a simple "ret" instruction is used to return to the caller. If a single argument was passed on the stack, a "ret 4" instruction would be required to return to the caller.

2. Registers EAX, EBX, ECX and EDX were not saved and restored since they were used to pass arguments.

13.4.1 Using the Stack-Based Calling Convention

When writing assembly language subprograms that use the stack-based calling convention, use the following guidelines.

1. All used registers, except registers EAX, ECX and EDX must be saved on entry and restored on exit. Also, if segment registers ES and DS are used, they must be saved on entry and restored on exit. Note that segment registers only have to be saved and restored if you are compiling your application with the "sr" option.
2. The direction flag must be clear before returning to the caller.
3. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, register CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.
4. In a small data model, segment register DS contains the segment address of the default data segment (group "DGROUP"). In a big data model, segment register SS (not DS) contains the segment address of the default data segment (group "DGROUP").
5. When writing assembly language subprograms for the small code model, you must declare them as "near". If you wish to write assembly language subprograms for the big code model, you must declare them as "far".
6. Use the ".8087" pseudo-op so that floating-point constants are in the correct format.
7. The caller will remove arguments that were passed on the stack.

8. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

Consider the following example.

```
      INTEGER HRS, MINS, SECS, HSECS
      CALL GETTIM( HRS, MINS, SECS, HSECS )
      PRINT 100, HRS, MINS, SECS, HSECS
100   FORMAT( 1X,I2.2,':',I2.2,':',I2.2,'.',I2.2 )
      END
```

GETTIM is an assembly language subroutine that gets the current time. It requires four integer arguments. The arguments are passed by reference so that the subroutine can return the hour, minute, seconds and hundredths of a second for the current time. These arguments will be passed to GETTIM on the stack.

The following is an assembly language subprogram which implements GETTIM.

Small or Flat Memory Model (small code, small data)

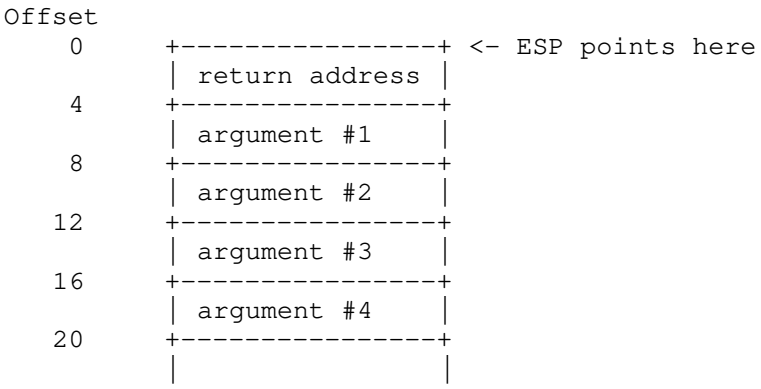
```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  GETTIM
GETTIM  proc    near
        push    EBP         ; save registers
        mov     EBP,ESP     ; ...
        push    ESI         ; ...
        mov     AH,2ch      ; set DOS "get time" function
        int     21h         ; issue DOS function call
        movzx   EAX,CH      ; get hours
        mov     ESI,8[EBP]  ; get address of hours
        mov     [ESI],EAX   ; return hours
        movzx   EAX,CL      ; get minutes
        mov     ESI,12[BP]  ; get address of minutes
        mov     [ESI],EAX   ; return minutes
        movzx   EAX,DH      ; get seconds
        mov     ESI,16[BP]  ; get address of seconds
        mov     [ESI],EAX   ; return seconds
        movzx   EAX,DL      ; get ticks
        mov     ESI,20[BP]  ; get address of ticks
        mov     [ESI],EAX   ; return ticks
        pop     ESI         ; restore registers
        mov     ESP,EBP     ; ...
        pop     EBP         ; ...
        ret                 ; return
GETTIM  endp
_TEXT   ends

        end
```

Notes:

1. The four arguments that were passed on the stack will be removed by the caller.

2. Registers ESI and EBP were saved and restored since they were used in GETTIM.

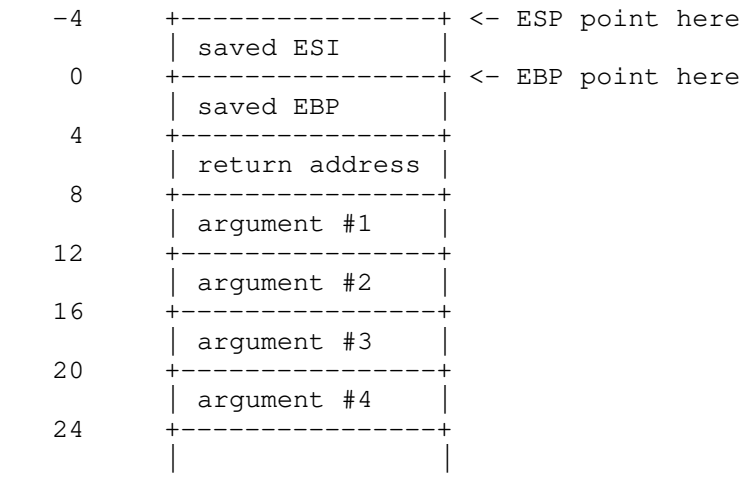
Let us look at the stack upon entry to GETTIM.



Notes:

1. The top element of the stack is a the 32-bit return address. The first argument is at offset 4 from the top of the stack, the second argument at offset 8, the third argument at offset 12, and the fourth argument at offset 16.

Register EBP is normally used to address arguments on the stack. Upon entry to the subroutine, registers that are modified (except registers EAX, ECX and EDX) are saved and register EBP is set to point to the stack. After performing this prologue sequence, the stack looks like this.



As the above diagram shows, the first argument is at offset 8 from register EBP, the second argument is at offset 12, the third argument is at offset 16, and the fourth argument is at offset 20.

13.4.2 Returning Values from Assembly Language Functions

The following illustrates the way function values are to be returned from assembly language functions.

> ***Note:*** The way in which a function returns its value does not change when the stack-based calling convention is used.

1. A **LOGICAL*1** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  L1
L1      proc    near
        mov     AL,1
        ret
L1      endp
_TEXT   ends
        end
```

2. A **LOGICAL*4** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  L4
L4      proc    near
        mov     EAX,0
        ret
L4      endp
_TEXT   ends
        end
```

3. An **INTEGER*1** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  I1
I1      proc    near
        mov     AL,73
        ret
I1      endp
_TEXT   ends
        end
```

4. An **INTEGER*2** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  I2
I2      proc    near
        mov     AX,7143
        ret
I2      endp
_TEXT   ends
        end
```

5. An **INTEGER*4** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  I4
I4      proc    near
        mov     EAX,383
        ret
I4      endp
_TEXT   ends
        end
```

6. A **REAL*4** function.

```
.8087

DGROUP  group R4_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  R4
R4      proc    near
        mov     EAX,dword ptr R4Val
        ret
R4      endp
_TEXT   ends

R4_DATA segment byte public 'DATA'
R4Val   dd 1314.3
R4_DATA ends

        end
```

7. A **REAL*8** function.

```
.8087

DGROUP  group R8_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  R8
R8      proc    near
        mov     EAX,dword ptr R8Val
        mov     EDX,dword ptr R8Val+4
        ret
R8      endp
_TEXT   ends

R8_DATA segment byte public 'DATA'
R8Val   dq 103.3
R8_DATA ends

        end
```

8. A **COMPLEX*8** function.

```
.8087

DGROUP  group C8_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  C8
C8      proc    near
        push    EAX
        mov     EAX,C8Val
        mov     [ESI],EAX
        mov     EAX,C8Val+4
        mov     4[ESI],EAX
        pop     EAX
        ret
C8      endp
_TEXT   ends

C8_DATA segment byte public 'DATA'
C8Val   dd 2.2
        dd 2.2
C8_DATA ends

        end
```

9. A **COMPLEX*16** function.

```
.8087

DGROUP  group C16_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  C16
C16     proc    near
        push    EAX
        mov     EAX,dword ptr C16Val
        mov     [ESI],EAX
        mov     EAX,dword ptr C16Val+4
        mov     4[ESI],EAX
        mov     EAX,dword ptr C16Val+8
        mov     8[ESI],EAX
        mov     EAX,dword ptr C16Val+12
        mov     12[ESI],EAX
        pop     EAX
        ret
C16     endp
_TEXT   ends
C16_DATA segment byte public 'DATA'
C16Val  dq 3.3
        dq 3.3
C16_DATA ends

        end
```

10. A **CHARACTER** function.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  CHR
CHR     proc    near
        push    EAX
        mov     EAX,[EAX]
        mov     byte ptr [EAX],'F'
        pop     EAX
        ret
CHR     endp
_TEXT   ends
        end
```

Remember, if you are using stack calling conventions (i.e., you specified the "sc" compiler option), arguments will be passed on the stack. The above character function must be modified as follows.

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  CHR
CHR     proc    near
        push    EAX
        mov     EAX,8[ESP]
        mov     EAX,[EAX]
        mov     byte ptr [EAX],'F'
        pop     EAX
        ret
CHR     endp
_TEXT   ends
        end
```

11. A function returning a user-defined structure.

```
DGROUP  group STRUCT_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  STRUCT
STRUCT  proc    near
        push    EAX
        mov     EAX,dword ptr StructVal
        mov     [ESI],EAX
        mov     EAX,dword ptr StructVal+4
        mov     4[ESI],EAX
        pop     EAX
        ret
STRUCT  endp
_TEXT   ends

STRUCT_DATA segment byte public 'DATA'
StructVal dd 7
          dd 3
STRUCT_DATA ends

        end
```

If you are using an 80x87 to return floating-point values, only **REAL*4** and **REAL*8** assembly language functions need to be modified. *Remember, this does not apply if you are using the stack-based calling convention.*

1. A **REAL*4** function using an 80x87.

```
.8087

DGROUP  group R4_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  R4
R4      proc    near
        fld     dword ptr R4Val
        ret
R4      endp
_TEXT   ends

R4_DATA segment byte public 'DATA'
R4Val   dd 1314.3
R4_DATA ends

        end
```

2. A **REAL*8** function using an 80x87.

```
.8087

DGROUP  group R8_DATA

_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        assume  DS:DGROUP
        public  R8
R8      proc    near
        fld     qword ptr R8Val
        ret
R8      endp
_TEXT   ends
```

```
R8_DATA segment byte public 'DATA'
R8Val   dq 103.3
R8_DATA ends

        end
```

The following is an example of a Open Watcom F77 program calling the above assembly language subprograms.

```
logical l1*1, l4*4
integer i1*1, i2*2, i4*4
real r4*4, r8*8
complex c8*8, c16*16
character chr
structure /coord/
    integer x, y
end structure
record /coord/ struct
print *, l1()
print *, l4()
print *, i1()
print *, i2()
print *, i4()
print *, r4()
print *, r8()
print *, c8()
print *, c16()
print *, chr()
print *, struct()
end
```

14 32-bit Pragmas

14.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas can be used to direct the Open Watcom F77 code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Open Watcom F77.

- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the FORTRAN 77 language level. The code generator can use this information to generate more efficient code.

- Any sequence of in-line machine language instructions, including DOS and BIOS function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. A pragma operator of the form, ***_Pragma*** ("string-literal") is an alternative method of specifying ***pragma*** directives.

For example, the following two statements are equivalent.

```
_Pragma( "library (\"kernel32.lib\")" )
#pragma library ("kernel32.lib")
```

The ***_Pragma*** operator can be used in macro definition.

```
# define LIBRARY(X) PRAGMA(library (#X))
# define PRAGMA(X) _Pragma(#X)
LIBRARY(kernel32.lib)  // same as #pragma library ("kernel32.lib")
```

The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc]	The item *abc* is optional.
{abc}	The item *abc* may be repeated zero or more times.
a|b|c	One of *a, b* or *c* may be specified.
a ::= b	The item *a* is defined in terms of *b*.
(a)	Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify default libraries
- pragmas that provide auxiliary information used for code generation

14.2 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

The backslash character ('\') is used to continue a pragma on the next line. Text following the backslash character is ignored. The line continuing the pragma must start with a comment character ('c', 'C' or '*').

14.2.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
*$pragma aux y <attrs_1>
code in which symbol y is referenced
code in which symbol z is referenced
*$pragma aux default <attrs_2>
*$pragma aux x <attrs_3>
```

Auxiliary attributes are assigned to `x`, `y` and `z` in the following way.

1. Symbol `x` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>` and `<attrs_3>`.

2. Symbol `y` is assigned the initial default attributes merged with the attributes specified by `<attrs_1>`.

3. Symbol `z` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>`.

14.2.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
*$pragma aux ( sym, alias )
```

where ***description***

sym is any valid FORTRAN 77 identifier.

alias is the alias name and is any valid FORTRAN 77 identifier.

Consider the following example.

```
*$pragma aux value_args parm (value)
*$pragma aux ( rtn, value_args )
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed by value.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
*$pragma aux ( alias ) sym aux_attrs
```

where ***description***

alias is the alias name and is any valid FORTRAN 77 identifier.

sym is any valid FORTRAN 77 identifier.

aux_attrs are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
*$pragma aux WC "*_" parm (value)
*$pragma aux (WC) rtn1
*$pragma aux (WC) rtn2
*$pragma aux (WC) rtn3
```

The routines `rtn1`, `rtn2` and `rtn3` assume the same attributes as the alias name `WC` which defines the calling convention used by the Open Watcom C compiler. Whenever calls are made to `rtn1`, `rtn2` and `rtn3`, the Open Watcom C calling convention will be used. Note that arguments must be passed by value. By default, Open Watcom F77 passes arguments by reference.

Note that if the attributes of `WC` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1`, `rtn2` and `rtn3`, we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

> ***WARNING!*** The alias name `WC` is just another symbol. If `WC` appeared in your source code, it would assume the attributes specified in the pragma for `WC`.

14.2.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` defines the calling convention used by Microsoft compilers.

__fastcall `__fastcall` defines the calling convention used by Microsoft compilers.

__fortran `__fortran` defines the calling convention used by Open Watcom FORTRAN compilers.

__pascal `__pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

__stdcall `__stdcall` defines a special calling convention used by the Win32 API functions.

__syscall `__syscall` defines the calling convention used by the 32-bit OS/2 API functions.

__watcall `__watcall` defines the calling convention used by Open Watcom compilers.

The following describes the attributes of the above alias names.

14.2.3.1 Predefined "__cdecl" Alias

```
*$pragma aux __cdecl "_*" \
c           parm caller [] \
c           value struct float struct routine [eax] \
c           modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register EAX.

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

14.2.3.2 Predefined "__pascal" Alias

```
*$pragma aux __pascal "^" \
c           parm reverse routine [] \
c           value struct float struct caller [] \
c           modify [eax ebx ecx edx]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value.

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

14.2.3.3 Predefined "__stdcall" Alias

```
*$pragma aux __stdcall "_*@nnn" \
c           parm routine [] \
c           value struct struct caller [] \
c           modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. All C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted.

3. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The called routine will remove the arguments from the stack.

4. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

5. Registers EAX, ECX and EDX are not saved and restored when a call is made.

14.2.3.4 Predefined "__syscall" Alias

```
*$pragma aux __syscall "*" \
c           parm caller [] \
c           value struct struct caller [] \
c           modify [eax ecx edx]
```

Notes:

1. Symbols names are not modified, that is, they are not adorned with leading or trailing underscores.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

14.2.3.5 Predefined "__watcall" Alias (register calling convention)

```
*$pragma aux __watcall "*_" \
c           parm routine [eax ebx ecx edx] \
c           value struct caller
```

Notes:

1. Symbol names are followed by an underscore character.

2. Arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left. The calling routine will remove the arguments if any were pushed on the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space is put into ESI register. The called routine then places the return value there. Upon returning from the call, register EAX will contain address of the space allocated for the return value.

4. Floating-point values are returned using 80x86 registers ("fpc" option) or using 80x87 floating-point registers ("fpi" or "fpi87" option).

5. All registers must be preserved by the called routine.

14.2.3.6 Predefined "__watcall" Alias (stack calling convention)

```
*$pragma aux __watcall "*" \
c           parm caller [] \
c           value no8087 struct caller \
c           modify [eax ecx edx 8087]
```

Notes:

1. All symbols appear in object form as they do in source form.
2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.
3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value.
4. Floating-point values are returned only using 80x86 registers.
5. Registers EAX, ECX and EDX are not preserved by the called routine.
6. Any local variables that are located in the 80x87 cache are not preserved by the called routine.

14.2.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
*$pragma aux sym obj_name
```

where	*description*
sym	is any valid FORTRAN 77 identifier.
obj_name	is any character string enclosed in double quotes.

When specifying `obj_name`, some characters have a special meaning:

where	*description*
*	is unmodified symbol name
^	is symbol name converted to uppercase
!	is symbol name converted to lowercase
#	is a placeholder for "@nnn", where nnn is size of all function parameters on the stack; it is ignored for functions with variable argument lists, or for symbols that are not functions

\ next character is treated as literal

Several examples of source to object form symbol name translation follow: By default, the upper case version "MYRTN" or "MYVAR" is placed in the object file.

In the following example, the name "MyRtn" will be replaced by "MYRTN_" in the object file.

```
*$pragma aux MyRtn "^_"
```

In the following example, the name "MyVar" will be replaced by "_MYVAR" in the object file.

```
*$pragma aux MyVar "_^"
```

In the following example, the lower case version "myrtn" will be placed in the object file.

```
*$pragma aux MyRtn "!"
```

In the following example, the name "MyRtn" will be replaced by "_MyRtn@nnn" in the object file. "nnn" represents the size of all function parameters.

```
*$pragma aux MyRtn "_*#"
```

In the following example, the name "MyRtn" will be replaced by "_MyRtn#" in the object file.

```
*$pragma aux MyRtn "_*\#"
```

The default mapping for all symbols can also be changed as illustrated by the following example.

```
*$pragma aux default "_^_"
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

14.2.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a subprogram is to be called.

```
*$pragma aux sym far
    or
*$pragma aux sym far16
    or
*$pragma aux sym near
    or
*$pragma aux sym = in_line

in_line ::= { const | "asm" }
```

where	*description*
sym	is a subprogram name.
const	is a valid FORTRAN 77 hexadecimal constant.
asm	is an assembly language instruction or directive.

In the following example, Open Watcom F77 will generate a far call to the subprogram `myrtn`.

```
*$pragma aux myrtn far
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Open Watcom F77 will generate a near call to the subprogram `myrtn`.

```
*$pragma aux myrtn near
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Open Watcom F77 will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line subprogram.

```
*$pragma aux mode4 =    \
*    zb4 z00            \ mov AH,0
*    zb0 z04            \ mov AL,4
*    zcd z10            \ int 10h
*    modify [ AH AL ]
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0       ; select function "set mode"
mov   AL,4       ; specify mode (mode 4)
int   10H        ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your FORTRAN 77 program.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
*$pragma aux mode4 =    \
*    "mov AH,0"         \
*    "mov AL,4"         \
*    "int 10H"          \
*    modify [ AH AL ]
```

The `__far16` attribute should only be used on systems that permit the calling of 16-bit code from 32-bit code. Currently, the only supported operating system that allows this is 32-bit OS/2. If you have any libraries of subprograms or APIs that are only available as 16-bit code and you wish to access these subprograms and APIs from 32-bit code, you must specify the `__far16` attribute. If the `__far16`

attribute is specified, the compiler will generate special code which allows the 16-bit code to be called from 32-bit code. Note that a `__far16` function must be a function whose attributes are those specified by one of the alias names `__cdecl` or `__pascal`. These alias names will be described in a later section.

The file `bsesub.fap` in the `\watcom\src\fortran\os2` directory contains examples of pragmas that use the `far16` attribute to describe the 16-bit VIO, KBD and MOU subsystems available in 32-bit OS/2.

14.2.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a subprogram that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified subprogram.

```
*$pragma aux sym parm loadds
```

where ***description***

sym is a subprogram name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified subprogram.

```
*$pragma aux sym loadds
```

where ***description***

sym is a subprogram name.

14.2.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link libraries are exported using the Open Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
*$pragma aux sym export
```

where	*description*
sym	is a subprogram name.

14.2.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Open Watcom F77 is to use for calling subprograms. This is particularly useful when interfacing to subprograms that have been compiled by other compilers or subprograms written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
*$pragma aux sym parm { arg_info | pop_info | reverse {reg_set} }

arg_info ::= ( arg_attr {, arg_attr} )

arg_attr ::= value [v_attr]
                 | reference [r_attr]
                 | data_reference [d_attr]

v_attr ::= far | near | *1 | *2 | *4 | *8

r_attr ::= [far | near] [descriptor | nodescriptor]

d_attr ::= [far | near]

pop_info ::= caller | routine
```

where	*description*
sym	is a subprogram name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

14.2.6.1 Passing Arguments to non-FORTRAN Subprograms

When calling a subprogram written in a different language, it may be necessary to provide the arguments in a form different than the default methods used by Open Watcom F77. For example, C functions require scalar arguments to be passed by value instead of by reference. For information on the methods Open Watcom F77 uses to pass arguments, see the chapter entitled "Assembly Language Considerations".

The following form of the auxiliary pragma can be used to alter the default calling mechanism used for passing arguments.

```
*$pragma aux sym parm ( arg_attr {, arg_attr} )

arg_attr ::= value [v_attr]
                 | reference [r_attr]
                 | data_reference [d_attr]

v_attr ::= far | near | *1 | *2 | *4 | *8

r_attr ::= [far | near] [descriptor | nodescriptor]

d_attr ::= [far | near]
```

where ***description***

sym is a subprogram name.

REFERENCE specifies that arguments are to be passed by reference. For non-character arguments, the address is a pointer to the data. For character arguments, the address is a pointer to a string descriptor. See the chapter entitled "Assembly Language Considerations" for a description of a string descriptor. This is the default calling mechanism. If "NEAR" or "FAR" is specified, a near pointer or far pointer is passed regardless of the memory model used at compile-time.

If the "DESCRIPTOR" attribute is specified, a pointer to the string descriptor is passed. This is the default. If the "NODESCRIPTOR" attribute is specified, a pointer to the the actual character data is passed instead of a pointer to the string descriptor.

DATA_REFERENCE specifies that arguments are to be passed by data reference. For non-character items, this is identical to passing by reference. For character items, a pointer to the actual character data (instead of the string descriptor) is passed. If "NEAR" or "FAR" is specified, a near pointer or far pointer is passed regardless of the memory model used at compile-time.

VALUE specifies that arguments are to be passed by value. Character arguments are treated specially when passed by value. Instead of passing a pointer to a string descriptor, a pointer to the actual character data is passed. See the chapter entitled "Assembly Language Considerations" for a description of a string descriptor.

Notes:

1. Arrays and subprograms are always passed by reference, regardless of the argument attribute specified.

2. When character arguments are passed by reference, the address of a string descriptor is passed. The string descriptor contains the address of the actual character data and the number of characters. When character arguments are passed by value or data reference, the address of the actual character data is passed instead of the address of a string descriptor. Character arguments are passed by value by specifying the "VALUE" or "DATA_REFERENCE" attribute. If "NEAR" or "FAR" is specified, a near pointer or far pointer to the character data is passed regardless of the memory model used at compile-time.

3. When complex arguments are passed by value, the real part and the imaginary part are passed as two separate arguments.

4. When an argument is a user-defined structure and is passed by value, a copy of the structure is made and passed as an argument.

5. For scalar arguments, arguments of type **INTEGER*1**, **INTEGER*2**, **INTEGER*4** ct , **REAL** or **DOUBLE PRECISION**, a length specification can be specified when the "VALUE" attribute is specified to pass the argument by value. This length specification refers to the size of the argument; the compiler will convert the actual argument to a type that matches the size. For example, if an argument of type **REAL** is passed to a subprogram that has an argument attribute of "VALUE*8", the argument will be converted to **DOUBLE PRECISION**. If an argument of type **DOUBLE PRECISION** is passed to a subprogram that has an argument attribute of "VALUE*4", the argument will be converted to **REAL**. If an argument of type **INTEGER*4** is passed to a subprogram that has an argument attribute of "VALUE*2" or VALUE*1, the argument will be converted to **INTEGER*2** or **INTEGER*1**. If an argument of type **INTEGER*2** is passed to a subprogram that has an argument attribute of "VALUE*4 or VALUE*1", the argument will be converted to **INTEGER*4** or **INTEGER*1**. If an argument of type **INTEGER*1** is passed to a subprogram that has an argument attribute of "VALUE*4 or VALUE*2", the argument will be converted to **INTEGER*4** or **INTEGER*2**.

6. If the number of arguments exceeds the number of entries in the argument-attribute list, the last attribute will be assumed for the remaining arguments.

Consider the following example.

```
*$pragma aux printf "*_" parm (value) caller []
      character cr/z0d/, nullchar/z00/
      call printf( 'values: %ld, %ld'//cr//nullchar,
     1             77, 31410 )
      end
```

The C "printf" function is called with three arguments. The first argument is of type **CHARACTER** and is passed as a C string (address of actual data terminated by a null character). The second and third arguments are passed by value. Also note that "printf" is a function that takes a variable number of arguments, all passed on the stack (an empty register set was specified), and that the caller must remove the arguments from the stack.

14.2.6.2 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular subprogram.

```
*$pragma aux sym parm {reg_set}
```

where	*description*
sym	is a subprogram name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **REAL** and **DOUBLE PRECISION** are always pushed on the stack when the "fpi" or "fpi87" option is used.

DOUBLE PRECISION

Arguments of type **DOUBLE PRECISION**, when passed by value, can only be passed in one of the following register pairs: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. For example, if the following register set was specified for a routine having an argument of type **DOUBLE PRECISION**,

```
[EBP EBX]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **DOUBLE PRECISION** is supported only when the "fpc" option is used. Note that this argument passing method does not include arguments of type **COMPLEX*8** or user-defined structures whose size is 8 bytes when these arguments are passed by value.

far pointer

A far pointer can only be passed in one of the following register pairs: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. For example, if a far pointer is passed to a function with the following register set,

```
[ES EBP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set. Far pointers are used to pass arguments by reference in a big data memory model.

INTEGER

The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **INTEGER** when passed by value or arguments passed by reference in a small data memory model) are: EAX, EBX, ECX, EDX, ESI and EDI. For example, if the following register set was specified for a routine with one argument of type **INTEGER**,

```
[EBP]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method also includes arguments of type **REAL** but only when the "fpc" option is used.

INTEGER*1, INTEGER*2

Arguments whose size is 1 byte or 2 bytes (e.g., arguments of type **INTEGER*1** and **INTEGER*2** as well as 2-byte structures when passed by value) are promoted to 4 bytes and are then assigned registers as if they were 4-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [EAX EBX ECX EDX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX. Specifying register EAX implies that register AX has been specified. Specifying register EBX implies that register BX has been specified. Specifying register ECX implies that register CX has been specified. Specifying register EDX implies that register DX has been specified. Specifying register EDI implies that register DI has been specified. Specifying register ESI implies that register SI has been specified. Specifying register EBP implies that register BP has been specified. Specifying register ESP implies that register SP has been specified.

3. If you are compiling for a memory model with a small data model, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

4. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

Consider the following example.

```
*$pragma aux myrtn parm (value) \
*                       [eax ebx ecx edx] [ebp esi]
```

Suppose `myrtn` is a routine with 3 arguments each of type **DOUBLE PRECISION**. Note that the arguments are passed by value.

1. The first argument will be passed in the register pair EDX:EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be pushed on the stack since EBP:ESI is not a valid register pair for arguments of type **DOUBLE PRECISION**.

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
*$pragma aux myrtn parm (value) \
*                       [eax ebx ecx edx] [esi edi]
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **INTEGER** and the second and third of type **DOUBLE PRECISION**. Note that all arguments are passed by value.

1. The first argument will be passed in the register EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be passed in the register set EDI:ESI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [EAX EBX ECX EDX] is used.

14.2.6.3 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a subprogram, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register ESI, the second argument to be passed in register EDI and the third argument to be passed in register ECX, the following auxiliary pragma can be used.

```
*$pragma aux mycopy parm (value) \
*                        [ESI] [EDI] [ECX]
      character*10  dst
      call mycopy( dst, '0123456789', 10 )
      ...
      end
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

14.2.6.4 Passing Arguments to In-Line Subprograms

For subprograms whose code is generated by Open Watcom F77 and whose argument list is described by an auxiliary pragma, Open Watcom F77 has some freedom in choosing how arguments are assigned to registers. Since the code for in-line subprograms is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Open Watcom F77 assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line subprogram called `scrollactivepgup`.

```
*$pragma aux scrollactivepgup =        \
*   "mov AH,6"                         \
*   "int 10h"                          \
*   parm (value)                       \
*        [ch] [cl] [dh] [dl] [al] [bh] \
*   modify [ah]
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Open Watcom F77 will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **INTEGER**, it would first be converted to **INTEGER*1** before assigning it to register CH. Similarly, if an in-line subprogram required its argument in register EAX and the argument was of type **INTEGER*2**, the argument would be converted to **INTEGER*4** before assigning it to register EAX.

In general, Open Watcom F77 assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **INTEGER*1**.

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **INTEGER*2**.

3. A register set consisting of a single 32-bit register (4 bytes) is assigned a type of **INTEGER*4**.

4. A register set consisting of two 32-bit registers (8 bytes) is assigned a type of **DOUBLE PRECISION**.

If the size of an integer argument is larger than the size specified by the register set, the argument will be truncated to the required size. If the size of an integer argument is smaller than the size specified by the register set, the argument will be padded (to the left) with zeros.

14.2.6.5 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
*$pragma aux sym parm (caller | routine)
```

where ***description***

sym is a subprogram name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

Consider the following example. It describes the pragma required to call the C "printf" function.

```
*$pragma aux printf "*_" parm (value) caller []
      character cr/z0d/, nullchar/z00/
      call printf( 'value is %ld'//cr//nullchar,
     1             7143 )
      end
```

The first argument must be passed as a C string, a pointer to the actual character data terminated by a null character. By default, the address of a string descriptor is passed for arguments of type **CHARACTER**.

See the chapter entitled "Assembly Language Considerations" for more information on string descriptors. The second argument is of type **INTEGER** and is passed by value. Also note that "printf" is a function that takes a variable number of arguments, all pushed on the stack (an empty register set was specified).

14.2.6.6 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
*$pragma aux sym parm reverse
```

where	*description*
sym	is a subprogram name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for subprograms that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a subprogram.

```
*$pragma aux rtn parm reverse []
```

14.2.7 Describing Subprogram Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
*$pragma aux sym value {no8087 | reg_set | struct_info}
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where	*description*
sym	is a function name.
reg_set	is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

14.2.7.1 Returning Subprogram Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
*$pragma aux sym value reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

Note that the method described below for returning values of type **REAL** or **DOUBLE PRECISION** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set.*

1-byte	For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.
2-byte	For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.
4-byte	For 4-byte return values (including near pointers), only the following register are allowed: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used. This form of the auxiliary pragma is legal for functions of type **REAL** when using the "fpc" option only.
far pointer	For functions that return far pointers, the following register pairs are allowed: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. If no register set is specified, the registers DX:EAX will be used.
8-byte	For 8-byte return values (including functions of type **DOUBLE PRECISION**), only the following register pairs are allowed: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. If no register set is specified, the registers EDX:EAX will be used. This form of the auxiliary pragma is legal for functions of type **DOUBLE PRECISION** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

3. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

14.2.7.2 Returning Structures and Complex Numbers

Typically, structures and complex numbers are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register ESI to point to it. The called routine then places the return value at the location pointed to by register ESI.

Complex numbers are not scalars but rather an ordered pair of real numbers. One can also view complex numbers as a *structure* containing two real numbers.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
*$pragma aux sym value struct (caller | routine) reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register EAX by the called routine.

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: EAX, EDX, EBX, ECX, ESI or EDI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following registers: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register ESI to it.

```
*$pragma aux sym value struct struct
```

where ***description***

sym is a subprogram name.

14.2.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **REAL** or **DOUBLE PRECISION** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register ESI to it.

```
*$pragma aux sym value struct float
```

where ***description***

sym is a function name.

In other words, floating-point values are to be returned in the same way complex numbers are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **REAL** will be returned in register EAX. Function return values whose type is **DOUBLE PRECISION** will be returned in registers EDX:EAX. This is the default method for the "fpc" option.

```
*$pragma aux sym value no8087
```

where ***description***

sym is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **REAL** or **DOUBLE PRECISION** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
*$pragma aux sym value [8087]
```

where *description*

sym is a function name.

14.2.8 A Subprogram that Never Returns

The following form of the auxiliary pragma can be used to describe a subprogram that does not return to the caller.

```
*$pragma aux sym aborts
```

where *description*

sym is a subprogram name.

Consider the following example.

```
*$pragma aux exitrtn aborts
      ...
      call exitrtn()
      end
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Open Watcom F77 generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn`.

14.2.9 Describing How Subprograms Use Variables in Common

The following form of the auxiliary pragma can be used to describe a subprogram that does not modify any variable that appears in a common block defined by the caller.

```
*$pragma aux sym modify nomemory
```

where *description*

sym is a subprogram name.

Consider the following example.

```
integer i
common /blk/ i
while( i .lt. 1000 )do
    i = i + 383
endwhile
call myrtn()
i = i + 13143
end

block data
common /blk/ i
integer i/1033/
end
```

To compile the above program, "rtn.for", we issue the following command.

```
C>wfc rtn -mm -d1
C>wfc386 rtn -d1
```

The "d1" compiler option is specified so that the object file produced by Open Watcom F77 contains source line information.

We can generate a file containing a disassembly of `rtn.obj` by issuing the following command.

```
C>wdis rtn -l -s -r
```

The "s" option is specified so that the listing file produced by the Open Watcom Disassembler contains source lines taken from `rtn.for`. The listing file `rtn.lst` appears as follows.

```
Module: rtn.for
Group: 'DGROUP' _DATA,LDATA,CDATA,BLK

Segment: 'FMAIN_TEXT' BYTE USE32  00000036 bytes

        integer i
        common /blk/ i
 0000  52                FMAIN          push    edx
 0001  8b 15 00 00 00 00                mov     edx,L3

        while( i .lt. 1000 )do
 0007  81 fa e8 03 00 00 L1             cmp     edx,000003e8H
 000d  7d 08                            jge     L2

            i = i + 383
        endwhile
 000f  81 c2 7f 01 00 00                add     edx,0000017fH
 0015  eb f0                            jmp     L1

        call myrtn()
 0017  89 15 00 00 00 00 L2             mov     L3,edx
 001d  e8 00 00 00 00                   call    MYRTN
 0022  8b 15 00 00 00 00                mov     edx,L3

        i = i + 13143
 0028  81 c2 57 33 00 00                add     edx,00003357H
 002e  89 15 00 00 00 00                mov     L3,edx

        end
```

```
        block data
        common /blk/ i
        integer i/1033/
        end
 0034  5a                                pop     edx
 0035  c3                                ret

No disassembly errors

List of external symbols

Symbol
-----------------
MYRTN            0000001e
------------------------------------------------------------

Segment: 'BLK' PARA USE32  00000004 bytes
 0000  09 04 00 00             L3              - ....

No disassembly errors

------------------------------------------------------------
List of public symbols

SYMBOL          GROUP            SEGMENT          ADDRESS
-------------------------------------------------------
FMAIN                            FMAIN_TEXT       00000000

------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
*$pragma aux myrtn modify nomemory
```

If we compile the source file with the above pragma and disassemble the object file using the Open Watcom Disassembler, we get the following listing file.

```
Module: rtn.for
Group: 'DGROUP' _DATA,LDATA,CDATA,BLK

Segment: 'FMAIN_TEXT' BYTE USE32  00000030 bytes

*$pragma aux myrtn modify nomemory
        integer i
        common /blk/ i
 0000  52                 FMAIN           push    edx
 0001  8b 15 00 00 00 00                  mov     edx,L3

        while( i .lt. 1000 )do
 0007  81 fa e8 03 00 00 L1               cmp     edx,000003e8H
 000d  7d 08                              jge     L2

            i = i + 383
        endwhile
 000f  81 c2 7f 01 00 00                  add     edx,0000017fH
 0015  eb f0                              jmp     L1

        call myrtn()
 0017  89 15 00 00 00 00 L2               mov     L3,edx
 001d  e8 00 00 00 00                     call    MYRTN

        i = i + 13143
 0022  81 c2 57 33 00 00                  add     edx,00003357H
 0028  89 15 00 00 00 00                  mov     L3,edx

        end
```

```
        block data
        common /blk/ i
        integer i/1033/
        end
 002e  5a                                pop     edx
 002f  c3                                ret

No disassembly errors

List of external symbols

Symbol
----------------
MYRTN            0000001e
-------------------------------------------------------------

Segment: 'BLK' PARA USE32  00000004 bytes
 0000  09 04 00 00             L3              - ....

No disassembly errors

-------------------------------------------------------------
List of public symbols

SYMBOL          GROUP            SEGMENT          ADDRESS
-------------------------------------------------------------
FMAIN                            FMAIN_TEXT       00000000

-------------------------------------------------------------
```

Notice that the value of `i` is in register EDX after completion of the "while" loop. After the call to `myrtn`, the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any variable that appears in a common block defined by `Rtn` and hence register EDX contains the correct value of `i`.

The preceding auxiliary pragma deals with routines that modify variables in common. Let us consider the case where routines reference variables in common. The following form of the auxiliary pragma can be used to describe a subprogram that does not reference any variable that appears in a common block defined by the caller.

```
*$pragma aux sym parm nomemory modify nomemory
```

where ***description***

sym is a subprogram name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
*$pragma aux myrtn parm nomemory modify nomemory
```

If you now compile our source file and disassemble the object file using WDIS, the result is the following listing file.

```
Module: rtn.for
Group: 'DGROUP' _DATA,LDATA,CDATA,BLK

Segment: 'FMAIN_TEXT' BYTE USE32  0000002a bytes

*$pragma aux myrtn parm nomemory modify nomemory
        integer i
        common /blk/ i
 0000  52                FMAIN           push    edx
 0001  8b 15 00 00 00 00                 mov     edx,L3

        while( i .lt. 1000 )do
 0007  81 fa e8 03 00 00 L1              cmp     edx,000003e8H
 000d  7d 08                             jge     L2

            i = i + 383
        endwhile
 000f  81 c2 7f 01 00 00                 add     edx,0000017fH
 0015  eb f0                             jmp     L1

        call myrtn()
 0017  e8 00 00 00 00    L2              call    MYRTN

        i = i + 13143
 001c  81 c2 57 33 00 00                 add     edx,00003357H
 0022  89 15 00 00 00 00                 mov     L3,edx

        end

        block data
        common /blk/ i
        integer i/1033/
        end
 0028  5a                                pop     edx
 0029  c3                                ret

No disassembly errors

List of external symbols

Symbol
----------------
MYRTN            00000018
------------------------------------------------------------

Segment: 'BLK' PARA USE32  00000004 bytes
 0000  09 04 00 00             L3              - ....

No disassembly errors

------------------------------------------------------------
List of public symbols

SYMBOL          GROUP           SEGMENT          ADDRESS
-------------------------------------------------------
FMAIN                           FMAIN_TEXT       00000000

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register EDX before calling `myrtn`. The auxiliary pragma informs the compiler that `myrtn` does not reference any variable that appears in a common block defined by `myrtn` so updating `i` was not necessary before calling `myrtn`.

14.2.10 Describing the Registers Modified by a Subprogram

The following form of the auxiliary pragma can be used to describe the registers that a subprogram will use without saving.

```
*$pragma aux sym modify [exact] reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

Specifying a register set informs Open Watcom F77 that the registers belonging to the register set are modified by the subprogram. That is, the value in a register before calling the subprogram is different from its value after execution of the subprogram.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called subprogram. Also, since the EAX register is frequently used to return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called subprogram does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called subprogram.

```
*$pragma aux sym modify exact reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set.

The above form of the auxiliary pragma tells Open Watcom F77 not to assume that the registers used to pass arguments will be modified by the called subprogram. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:

```
*$ifdef __386__
*$pragma aux GetSP = value [esp] modify exact []
*$else
*$pragma aux GetSP = value [sp] modify exact []
*$endif

      program main
      integer GetSP
      print *, 'Current SP =', GetSP()
      end
```

14.2.11 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile subprograms. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

14.2.11.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a subprogram is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to subprograms.

```
*$pragma aux sym parm {reg_set}
```

where ***description***

sym is a subprogram name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack.

ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).

```
              +----------------+
        0     | 4th from top   |  ST(4)
              +----------------+
        1     | 5th from top   |  ST(5)
              +----------------+
        2     | 6th from top   |  ST(6)
              +----------------+
        3     | 7th from top   |  ST(7)
              +----------------+
ST -> 4       | top of stack   |  ST(0)
              +----------------+
        5     | 1st from top   |  ST(1)
              +----------------+
        6     | 2nd from top   |  ST(2)
              +----------------+
        7     | 3rd from top   |  ST(3)
              +----------------+
```

Starting with version 9.5, the Open Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
*$pragma aux myrtn parm (value) [8087]

      real x
      double precision y
      integer i
      integer j
      x = 7.7
      i = 7
      y = 77.77
      j = 77
      call myrtn( x, i, y, j )
      end
```

`myrtn` is an assembly language subprogram that requires four arguments. The first argument of type **REAL** (4 bytes), the second argument is of type **INTEGER** (4 bytes), the third argument is of type **DOUBLE PRECISION** (8 bytes) and the fourth argument is of type **INTEGER*4** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **REAL**, will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
*$pragma aux myrtn parm [eax 8087]
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **REAL** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register EAX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **DOUBLE PRECISION**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

14.2.11.2 Using the 80x87 to Return Subprogram Values

The following form of the auxiliary pragma can be used to describe a subprogram that returns a floating-point value in ST(0).

```
*$pragma aux sym value reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set containing the string "8087", i.e. [8087].

14.2.11.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a subprogram unless the "fpr" option is used to generate backward compatible code (older Open Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified subprogram.

```
*$pragma aux sym modify reg_set
```

where	*description*
sym	is a subprogram name.
reg_set	is a register set containing the string "8087", i.e. [8087].

This instructs Open Watcom F77 to save any local variables that are located in the 80x87 cache before calling the specified routine.

Appendices

A. Use of Environment Variables

In the Open Watcom FORTRAN 77 software development package, a number of environment variables are used. This appendix summarizes their use with a particular component of the package.

A.1 FINCLUDE

The **FINCLUDE** environment variable describes the location of the Open Watcom FORTRAN 77 include files. This variable is used by Open Watcom FORTRAN 77.

```
SET FINCLUDE=[d:][path];[d:][path]...
```

The **FINCLUDE** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

A.2 LFN

The **LFN** environment variable is checked by the Open Watcom run-time C libraries and it is used to control DOS LFN (DOS Long File Name) support. Normally, these libraries will use DOS LFN support if it is available on host OS. If you don't wish to use DOS LFN support, you can define the **LFN** environment variable and setup it's value to 'N'. Using the "SET" command, define the environment variable as follows:

```
SET LFN=N
```

Now, when you run your application, the DOS LFN support will be ignored. To undefine the environment variable, enter the command:

```
SET LFN=
```

A.3 LIB

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

The **LIB** environment variable is used to select the libraries that will be used when the application is linked. This variable is used by the Open Watcom Linker (WLINK.EXE). The **LIB** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

If you have the 286 development system, 16-bit applications can be linked for DOS, Microsoft Windows, OS/2, and QNX depending on which libraries are selected. If you have the 386 development system, 32-bit applications can be linked for DOS Extender systems, Microsoft Windows and QNX.

A.4 LIBDOS

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a DOS application, the **LIBDOS** environment variable must include the location of the 16-bit Open Watcom F77 DOS library files (files with the ".lib" filename extension). This variable is used by the Open Watcom Linker (WLINK.EXE). The default installation directory for the 16-bit Open Watcom F77 DOS libraries is `\WATCOM\LIB286\DOS`.

Example:

```
C>set libdos=c:\watcom\lib286\dos
```

A.5 LIBWIN

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a 16-bit Microsoft Windows application, the **LIBWIN** environment variable must include the location of the 16-bit Open Watcom F77 Windows library files (files with the ".lib" filename extension). This variable is used by the Open Watcom Linker (WLINK.EXE). If you are developing a 32-bit Microsoft Windows application, see the description of the **LIBPHAR** environment variable. The default installation directory for the 16-bit Open Watcom F77 Windows libraries is `\WATCOM\LIB286\WIN`.

Example:

```
C>set libwin=c:\watcom\lib286\win
```

A.6 LIBOS2

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing an OS/2 application, the **LIBOS2** environment variable must include the location of the 16-bit Open Watcom F77 OS/2 library files (files with the ".lib" filename extension). This variable is used by the Open Watcom Linker (WLINK.EXE). The default installation directory for the 16-bit Open Watcom F77 OS/2 libraries is `\WATCOM\LIB286\OS2`. The **LIBOS2** environment variable must also include the directory of the OS/2 `DOSCALLS.LIB` file which is usually `\OS2`.

Example:

```
C>set libos2=c:\watcom\lib286\os2;c:\os2
```

A.7 LIBPHAR

The use of the **WATCOM** environment variable and the Open Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a 32-bit Windows or DOS Extender application, the **LIBPHAR** environment variable must include the location of the 32-bit Open Watcom F77 DOS Extender library files or the 32-bit Open Watcom F77 Windows library files (files with the ".lib" filename extension). This variable is used by the Open Watcom Linker (WLINK.EXE). The default installation directory for the 32-bit Open Watcom F77 DOS Extender libraries is `\WATCOM\LIB386\DOS`. The default installation directory for the 32-bit Open Watcom F77 Windows libraries is `\WATCOM\LIB386\WIN`.

Example:

```
C>set libphar=c:\watcom\lib386\dos
    or
C>set libphar=c:\watcom\lib386\win
```

A.8 NO87

The **NO87** environment variable is checked by the Open Watcom FORTRAN 77 run-time libraries that include floating-point emulation support. Normally, these libraries will detect the presence of a numeric data processor (80x87) and use it. If you have a numeric data processor in your system but you wish to test a version of your application that will use floating-point emulation, you can define the **NO87** environment variable. Using the "SET" command, define the environment variable as follows:

```
SET NO87=1
```

Now, when you run your application, the 80x87 will be ignored. To undefine the environment variable, enter the command:

```
SET NO87=
```

A.9 PATH

The **PATH** environment variable is used by DOS "COMMAND.COM" or OS/2 "CMD.EXE" to locate programs.

```
PATH [d:][path];[d:][path]...
```

The **PATH** environment variable should include the disk and directory of the Open Watcom FORTRAN 77 binary program files when using Open Watcom FORTRAN 77 and its related tools.

If your host system is DOS:

The default installation directory for 16-bit Open Watcom F77 and 32-bit Open Watcom F77 DOS binaries is called `\WATCOM\BINW`.

Example:

```
C>path c:\watcom\binw;c:\dos;c:\windows
```

If your host system is OS/2:

The default installation directories for 16-bit Open Watcom F77 and 32-bit Open Watcom F77 OS/2 binaries are called `\WATCOM\BINP` and `\WATCOM\BINW`.

Example:

```
[C:\]path c:\watcom\binp;c:\watcom\binw
```

If your host system is Windows NT:

The default installation directories for 16-bit Open Watcom F77 and 32-bit Open Watcom F77 Windows NT binaries are called `\WATCOM\BINNT` and `\WATCOM\BINW`.

Example:

```
C>path c:\watcom\binnt;c:\watcom\binw
```

The **PATH** environment variable is also used by the following programs in the described manner.

1. Open Watcom Compile and Link to locate the 16-bit Open Watcom F77 and 32-bit Open Watcom F77 compilers and the Open Watcom Linker.
2. "WD.EXE" to locate programs and debugger command files.

A.10 TMP

The **TMP** environment variable describes the location (disk and path) for temporary files created by the 16-bit Open Watcom F77 and 32-bit Open Watcom F77 compilers and the Open Watcom Linker.

```
SET TMP=[d:][path]
```

Normally, Open Watcom FORTRAN 77 will create temporary spill files in the current directory. However, by defining the **TMP** environment variable to be a certain disk and directory, you can tell Open Watcom FORTRAN 77 where to place its temporary files. The same is true of the Open Watcom Linker temporary file.

Consider the following definition of the **TMP** environment variable.

Example:

```
C>set tmp=d:\watcom\tmp
```

The Open Watcom FORTRAN 77 compiler and Open Watcom Linker will create its temporary files in `d:\watcom\tmp`.

A.11 WATCOM

In order for the Open Watcom Linker to locate the 16-bit Open Watcom F77 and 32-bit Open Watcom F77 library files, the **WATCOM** environment variable should be defined. The **WATCOM** environment variable is used to locate the libraries that will be used when the application is linked. The default directory for 16-bit Open Watcom F77 and 32-bit Open Watcom F77 files is "\WATCOM".

Example:

```
C>set watcom=c:\watcom
```

A.12 WCL

The **WCL** environment variable can be used to specify commonly-used WFL options.

```
SET WCL=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "mm" (compile code for medium memory model), "d1" (include line number debug information in the object file), and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcl=-mm -d1 -ox
```

Once the **WCL** environment variable has been defined, those options listed become the default each time the WFL command is used.

A.13 WCL386

The **WCL386** environment variable can be used to specify commonly-used WFL386 options.

```
SET WCL386=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "3s" (compile code for stack-based argument passing convention), "d1" (include line number debug information in the object file), and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcl386=-3s -d1 -ox
```

Once the **WCL386** environment variable has been defined, those options listed become the default each time the WFL386 command is used.

A.14 WCGMEMORY

The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:

```
C>set WCGMEMORY=?
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:

```
C>set WCGMEMORY=128
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure that the **WCGMEMORY** environment variable is set identically on both machines.

The second reason where this feature is useful is on virtual memory paging systems (e.g., OS/2) where an unlimited amount of memory can be used by the code generator. If a very large module is being compiled, it may take a very long time to compile it. The code generator will continue to allocate more and more memory and cause an excessive amount of paging. By restricting the amount of memory that the code generator can use, you can reduce the amount of time required to compile a routine.

A.15 WD

The **WD** environment variable can be used to specify commonly-used Open Watcom Debugger options. This environment variable is not used by the Windows version of the debugger, WDW.

```
SET WD=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "noinvoke" (do not execute the `profile.dbg` file) and "reg=10" (retain up to 10 register sets while tracing).

Example:

```
C>set wd=-noinvoke -reg#10
```

Once the **WD** environment variable has been defined, those options listed become the default each time the WD command is used.

A.16 WDW

The **WDW** environment variable can be used to specify commonly-used Open Watcom Debugger options. This environment variable is used by the Windows version of the debugger, WDW.

```
SET WDW=-option1 -option2 ...
```

These options are processed before options specified in the WDW prompt dialogue box. The following example defines the default options to be "noinvoke" (do not execute the `profile.dbg` file) and "reg=10" (retain up to 10 register sets while tracing).

Example:

```
C>set wdw=-noinvoke -reg#10
```

Once the **WDW** environment variable has been defined, those options listed become the default each time the WDW command is used.

A.17 WFC

The **WFC** environment variable can be used to specify commonly-used Open Watcom F77 options.

```
SET WFC=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "om" (compile with math optimizations).

Example:

```
C>set wfc=-d1 -om
```

Once the **WFC** environment variable has been defined, those options listed become the default each time the WFC command is used.

A.18 WFC386

The **WFC386** environment variable can be used to specify commonly-used Open Watcom F77 options.

```
SET WFC386=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "om" (compile with math optimizations).

Example:

```
C>set wfc386=-d1 -om
```

Once the **WFC386** environment variable has been defined, those options listed become the default each time the WFC386 command is used.

A.19 WFL

The **WFL** environment variable can be used to specify commonly-used WFL options.

```
SET WFL=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "mm" (compile code for medium memory model), "d1" (include line number debug information in the object file), and "ox" (default optimizations).

Example:

```
C>set wfl=-mm -d1 -ox
```

Once the **WFL** environment variable has been defined, those options listed become the default each time the WFL command is used.

A.20 WFL386

The **WFL386** environment variable can be used to specify commonly-used WFL386 options.

```
SET WFL386=-option1 -option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "mf" (flat memory model), "d1" (include line number debug information in the object file), and "ox" (default optimizations).

Example:

```
C>set wfl386=-mf -d1 -ox
```

Once the **WFL386** environment variable has been defined, those options listed become the default each time the WFL386 command is used.

A.21 WLANG

The **WLANG** environment variable can be used to control which language is used to display diagnostic and program usage messages by various Open Watcom software tools. The two currently-supported values for this variable are "English" or "Japanese".

```
SET WLANG=English
SET WLANG=Japanese
```

Alternatively, a numeric value of 0 (for English) or 1 (for Japanese) can be specified.

Example:

```
C>set wlang=0
```

By default, Japanese messages are displayed when the current codepage is 932 and English messages are displayed otherwise. Normally, use of the **WLANG** environment variable should not be required.

B. Open Watcom F77 Diagnostic Messages

The Open Watcom FORTRAN 77 compiler checks for errors both at compile time and execution time.

Compile time errors may result from incorrect program syntax, violations of the rules of the language, underflow and overflow as a result of evaluation of expressions, etc. Three types of messages are issued:

EXTENSION *EXT* - This indicates that the programmer has used a feature which is strictly an extension of the FORTRAN 77 language definition. Such extensions may not be accepted by other FORTRAN 77 compilers.

WARNING *WRN* - This indicates that a possible problem has been detected by the compiler. For example, an unlabelled executable statement which follows an unconditional transfer of control can never be executed and so the compiler will issue a message about this condition.

ERROR *ERR* - This indicates that some error was detected which must be corrected by the programmer.

An object file will be created as long as no ERROR type messages are issued.

Execution or run time errors may result from arithmetic underflow or overflow, input/output errors, etc. An execution time error causes the program to cease execution.

Consider the following program, named "DEMO1.FOR", which contains errors.

Example:

```
* This program demonstrates the following features of
* Open Watcom's FORTRAN 77 compiler:
*
*    1. Extensions to the FORTRAN 77 standard are flagged.
*
*    2. Compile time error diagnostics are extensive.  As many
*       errors as possible are diagnosed.
*
*    3. Warning messages are displayed where potential problems
*       can arise.
*
        PROGRAM MAIN
        DIMENSION A(10)
        DO I=1,10
            A(I) = I
            I = I + 1
        ENDLOOP
        GO TO 30
        J = J + 1
30      END
```

If we compile this program with the "extensions" option, the following output appears on the screen.

```
C>wfc demo1 /exten
WATCOM FORTRAN 77/16 Optimizing Compiler Version 2.0 1997/07/16 09:22:47
Copyright (c) 2002-2022 the Open Watcom Contributors. All Rights Reserved.
Portions Copyright (c) 1984-2002 Sybase, Inc. All Rights Reserved.
Source code is available under the Sybase Open Watcom Public License.
See http://www.openwatcom.org/ for details.
demo1.for(14): *EXT* DO-05 this DO loop form is not FORTRAN 77 standard
demo1.for(16): *ERR* DO-07 column 13, DO variable cannot be redefined
    while DO loop is active
demo1.for(17): *ERR* SP-19 ENDLOOP statement does not match with DO
    statement
demo1.for(19): *WRN* ST-08 this statement will never be executed due
    to the preceding branch
demo1.for: 9 statements, 0 bytes, 1 extensions, 1 warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number of the line containing the error (in parentheses),
3. a message type of either extension (*EXT*), error (*ERR*) or warning (*WRN*),
4. a message class and number (e.g., ST-08), and
5. text explaining the nature of the error.

In the above example, the first error occurred on line 16 of the file "DEMO1.FOR". Error number DO-07 was diagnosed. The second error occurred on line 17 of the file "DEMO1.FOR". Error number SP-20 was diagnosed. The other errors are informational messages that do not prevent the successful compilation of the source file.

The following is a list of all messages produced by Open Watcom F77 followed by a brief description. Run-time messages (messages displayed during execution) are also presented. The messages contain references to `%s` and `%d`. They represent strings that are substituted by Open Watcom F77 to make the error message more exact. `%d` represents a string of digits; `%s` any string, usually a symbolic name such as `AMOUNT`, a type such as `INTEGER`, or a symbol class such as `SUBROUTINE`. An error message may contain more than one reference to %d. In such a case, the description will reference them as %dn where n is the occurrence of %d in the error message. The same is true for references to %s.

B.1 Subprogram Arguments

AR-01 *invalid number of arguments to intrinsic function %s1*

The number of actual arguments specified in the argument list for the intrinsic function %s1 does not agree with the dummy argument list. Consult the Language Reference for information on intrinsic functions and their arguments.

AR-02 *dummy argument %s1 appears more than once*

The same dummy argument %s1 is named more than once in the dummy argument list.

AR-03 *dummy argument %s1 must not appear before definition of ENTRY %s2*

The dummy argument %s1 has appeared in an executable statement before its appearance in the definition of %s2 in an ENTRY statement. This is illegal.

B.2 Block Data Subprograms

BD-01 *%s1 was initialized in a block data subprogram but is not in COMMON*

The variable or array element, %s1, was initialized in a BLOCK DATA subprogram but was not specified in a named COMMON block.

BD-02 *%s1 statement is not permitted in a BLOCK DATA subprogram*

The statement, %s1, is not allowed in a BLOCK DATA subprogram. The only statements which are allowed to appear are: IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, and type statements.

B.3 Source Format and Contents

CC-01 *invalid character encountered in source input*

The indicated statement contains an invalid character. Valid characters are: letters, digits, $, *, ., +, –, /, :, =, (,), !, %, ', and ,(comma). Any character may be used inside a character or hollerith string.

CC-02 *invalid character in statement number columns*

A column in columns 1 to 5 of the indicated statement contains a non-digit character. Columns 1 to 5 contain the statement number label. It is made up of digits from 0 to 9 and is greater than 0 and less than or equal to 99999.

CC-03 *character in continuation column, but no statement to continue*

The character in column 6 indicates that this line is a continuation of the previous statement but there is no previous statement to continue.

CC-04 *character encountered is not FORTRAN 77 standard*

A non-standard character was encountered in the source input stream. This is most likely caused by the use of lower case letters.

CC-05 *columns 1-5 in a continuation line must be blank*

When column 6 is marked as a continuation statement to the previous line, columns 1 to 5 must be left blank.

CC-06 *more than 19 continuation lines is not FORTRAN 77 standard*

More than 19 continuation lines is an extension to the FORTRAN 77 language.

CC-07 *end-of-line comment is not FORTRAN 77 standard*

End-of-line comments are an extension to the FORTRAN 77 language. End-of-line comments start with the exclamation mark (!) character.

CC-08 *D in column 1 is not FORTRAN 77 standard*

A "D" in column 1 signifies a debug statement that is compiled when the "__debug__" macro symbol is defined. If the "__debug__" macro symbol is not defined, the statement is ignored. The "c$define" compiler directive or the "define" compiler option can be used to define the "__debug__" macro symbol.

CC-09 *too many continuation lines*

The limit on the number of continuation lines has been reached. This limit depends on the size of each continuation line. A minimum of 61 continuation lines is permitted. If the "xline" option is used, a minimum of 31 continuation lines is permitted.

B.4 COMMON Blocks

CM-01 *%s1 already in COMMON*

The variable or array name, %s1, has already been specified in this or another COMMON block.

CM-02 *initializing %s1 in COMMON outside of block data subprogram is not FORTRAN 77 standard*

The symbol %s1, in a named COMMON block, has been initialized outside of a block data subprogram. This is an extension to the FORTRAN 77 language.

CM-03 *character and non-character data in COMMON is not FORTRAN 77 standard*

The FORTRAN 77 standard specifies that a COMMON block cannot contain both numeric and character data. Allowing COMMON blocks to contain both numeric and character data is an extension to the FORTRAN 77 standard.

CM-04 *COMMON block %s1 has been defined with a different size*

The COMMON block %s1 has been defined with a different size in another subprogram. A named COMMON block must define the same amount of storage units where ever named.

CM-05 *named COMMON block %s1 appears in more than one BLOCK DATA subprogram*

The named COMMON block, %s1, may not appear in more than one BLOCK DATA subprogram.

CM-06 *blank COMMON block has been defined with a different size*

The blank COMMON block has been defined with a different size in another subprogram. This is legal but a warning message is issued.

B.5 Constants

CN-01 *DOUBLE PRECISION COMPLEX constants are not FORTRAN 77 standard*

Double precision complex numbers are an extension to the FORTRAN 77 language. The indicated number is a complex number and at least one of the parts, real or imaginary, is a double precision constant. Both real and imaginary parts will be double precision.

CN-02 *invalid floating-point constant %s1*

The floating-point constant %s1 is invalid. Refer to the chapter entitled "Names, Data Types and Constants" in the Language Reference.

CN-03 *zero length character constants are not allowed*

FORTRAN 77 does not allow character constants of length 0 (i.e., an empty string).

CN-04 *invalid hexadecimal/octal constant*

An invalid hexadecimal or octal constant was specified. Hexadecimal constants can only contain digits or the letters ’a’ through ’f’ and ’A’ through ’F’. Octal constants can only contain the digits ’0’ through ’7’.

CN-05 *hexadecimal/octal constant is not FORTRAN 77 standard*

Hexadecimal and octal constants are extensions to the FORTRAN 77 standard.

B.6 Compiler Options

CO-01 *%s1 is already being included*

An attempt has been made to include a file that is currently being included in the program.

CO-02 *'%s1' option cannot take a NO prefix*

The compiler option %s1, cannot have the NO prefix specified. The NO prefix is used to negate an option. Certain options, including all options that require a value cannot have a NO prefix.

CO-03 *expecting an equals sign following the %s1 option*

The compiler option %s1, requires an equal sign to be between the option keyword and its associated value.

CO-04 *the '%s1' option requires a number*

The compiler option %s1 and an equal sign has been detected but the required associated value is missing.

CO-05 *option '%s1' not recognized - ignored*

The option %s1 is not a recognized compiler option and has been ignored. Consult the User's Guide for a complete list of compiler options.

CO-06 *'%s1' option not allowed in source input stream*

The option %s1 can only be specified on the command line. Consult the User's Guide for a description of which options are allowed in the source input stream.

CO-07 *nesting level exceeded for compiler directives*

Use of the C$IFDEF or C$IFNDEF compiler directives has caused the maximum nesting level to be exceeded. The maximum nesting level is 16.

CO-08 *mismatching compiler directives*

This error message is issued if, for example, a C$ENDIF directive is used and no matching C$IFDEF or C$IFNDEF precedes it. Incorrect nesting of C$IFDEF, C$IFNDEF, C$ELSE and C$ENDIF directives will also cause this message to be issued.

CO-09 *DATA option not allowed*

A source file has been included into the current program through the use of the INCLUDE compiler option. This included source file cannot contain the DATA compiler option.

CO-10 *maximum limit exceeded in the '%s1' option - option ignored*

The user has specified a value on an option which exceeds the maximum allowed value.

CO-11 *DATA option not allowed with OBJECT option*

The DATA compiler option can not appear a file that is compiled with the OBJECT option.

B.7 Compiler Errors

CP-01 *program abnormally terminated*

This message is issued during the execution of the program. If you are running FORTRAN 77, this message indicates that an internal error has occurred in the compiler. Please report this error and any other helpful information about the program being compiled to Watcom so that the problem can be fixed. .pc If you are running an application compiled by the Watcom FORTRAN 77 optimizing compiler, this message may indicate a problem with the compiler or a problem with your program. Try compiling your application with the "debug" option. This causes the generation of run-time checking code to validate, for example, array subscripts and will help ensure that your program is not in error.

CP-02 *argument %d1 incompatible with register*

The register specified in an auxiliary pragma for argument number %d1 is invalid.

CP-03 *subprogram %s1 has invalid return register*

The register specified in an auxiliary pragma for the return value of function %s1 is invalid. This error is issued when, for example, an auxiliary pragma is used to specify EAX as the return register for a double precision function.

CP-04 *low on memory - unable to fully optimize %s1*

There is not enough memory for the code generator to fully optimize subprogram %s1.

CP-05 *internal compiler error %d1*

This error is an internal code generation error. Please report the specified internal compiler error number and any other helpful information about the program being compiled to Watcom so that the problem can be fixed.

CP-06 *illegal register modified by %s1*

An illegal register was said to be modified by %s1 in the auxiliary pragma for %s1. In a 32-bit flat memory model, the base pointer register EBP and segment registers CS, DS, ES, and SS cannot be modified. In small data models, the base pointer register (32-bit EBP or 16-bit BP) and segment registers CS, DS, and SS cannot be modified. In large data models, the base pointer register (32-bit EBP or 16-bit BP) and segment registers CS, and SS cannot be modified.

CP-07 *%s1*

The message specified by %s1 indicates an error during the code generation phase. The most probable cause is an invalid instruction in the in-line assembly code specified in an auxiliary pragma.

CP-08 *fatal: %s1*

The specified error indicates that the code generator has been abnormally terminated. This message will be issued if any internal limit is reached or a keyboard interrupt sequence is pressed during the code generation phase.

CP-09 *dynamic memory not freed*

This message indicates an internal compiler error. Please report this error and any other helpful information about the program being compiled to Watcom so that the problem can be fixed.

CP-10 *freeing unowned dynamic memory*

This message indicates an internal compiler error. Please report this error and any other helpful information about the program being compiled to Watcom so that the problem can be fixed.

CP-11 *The automatic equivalence containing %s1 exceeds 32K limit*

In 16-bit environments, the size of an equivalence on the stack must not exceed 32767 bytes.

CP-12 *The return value of %s1 exceeds 32K limit*

In 16-bit environments, the size of the return value of a function must not exceed 32767 bytes.

CP-13 *The automatic variable %s1 exceeds 32K limit*

In 16-bit environments, the size of any variable on the stack must not exceed 32767 bytes.

B.8 Character Variables

CV-01 *CHARACTER variable %s1 with length (*) not allowed in this expression*

The length of the result of evaluating the expression is indeterminate. One of the operands has an indeterminate length and the result is being assigned to a temporary.

CV-02 *character variable %s1 with length (*) must be a subprogram argument*

The character variable %s1 with a length specification (*) can only be used to declare dummy arguments in the subprogram. The length of a dummy argument assumes the length of the corresponding actual argument.

CV-03 *left and right hand sides overlap in a character assignment statement*

The expression on the right hand side defines a substring of a character variable and tries to assign it to an overlapping part of the same character variable.

B.9 Data Initialization

DA-01 *implied DO variable %s1 must be an integer variable*

The implied DO variable %s1 must be declared as a variable of type INTEGER or must have an implicit INTEGER type.

DA-02 *repeat specification must be a positive integer*

The repeat specification in the constant list of the DATA statement must be an unsigned positive integer.

DA-03 *%s1 appears in an expression but is not an implied DO variable*

The variable %s1 is used to express the array elements in the DATA statement but the variable is not used as an implied DO variable.

DA-04 *%s1 in blank COMMON block cannot be initialized*

A blank or unnamed COMMON block is a COMMON statement with the block name omitted. The entries in blank COMMON blocks cannot be initialized using DATA statements.

DA-05 *data initialization with hexadecimal constant is not FORTRAN 77 standard*

Data initialization with hexadecimal constants is an extension to the FORTRAN 77 language.

DA-06 *cannot initialize %s1 %s2*

Symbol %s2 was used as a %s1. It is illegal for such a symbol to be initialized in a DATA statement. The DATA statement can only be used to initialize variables, arrays, array elements, and substrings.

DA-07 *data initialization in %s1 statement is not FORTRAN 77 standard*

Data initialization in type specification statements is an extension to the FORTRAN 77 language. These include: CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, and REAL.

DA-08 *not enough constants for list of variables*

There are not enough constants specified to initialize all of the names listed in the DATA statement.

DA-09 *too many constants for list of variables*

There are too many constants specified to initialize the names listed in the DATA statement.

DA-10 *cannot initialize %s1 variable %s2 with %s3 constant*

The constant of type %s3 cannot be used to initialize the variable %s2 of type %s1.

DA-11 *entity can only be initialized once during data initialization*

An attempt has been made to initialize an entity more than once in DATA statements.

B.10 Dimensioned Variables

DM-01 *using %s1 incorrectly in dimension expression*

The name used as a dimension declarator has been previously declared as type %s1 and cannot be used as a dimension declarator. A dimension declarator must be an integer expression.

DM-02 *array or array element (possibly substring) associated with %s1 too small*

The dummy argument, array %s1, is defined to be larger than the size of the actual argument.

B.11 DO-loops

DO-01 *statement number %i1 already defined in line %d2 - DO loop is backwards*

The statement number to indicate the end of the DO control structure has been used previously in the program unit and cannot be used to terminate the DO loop. The terminal statement named in the DO statement must follow the DO statement.

DO-02 *%s1 statement not allowed at termination of DO range*

A non-executable statement cannot be used as the terminal statement of a DO loop. These statements include: all declarative statements, ADMIT, AT END, BLOCK DATA, CASE, DO, ELSE, ELSE IF, END, END AT END, END BLOCK, END GUESS, END IF, END LOOP, END SELECT, END WHILE, ENTRY, FORMAT, FUNCTION, assigned GO TO, unconditional GO TO, GUESS, arithmetic and block IF, LOOP, OTHERWISE, PROGRAM, RETURN, SAVE, SELECT, STOP, SUBROUTINE, UNTIL, and WHILE.

DO-03 *improper nesting of DO loop*

A nested DO loop has not been properly terminated before the termination of the outer DO loop.

DO-04 *ENDDO cannot terminate DO loop with statement label*

The ENDDO statement can only terminate a DO loop in which no statement label was specified in the defining DO statement.

DO-05 *this DO loop form is not FORTRAN 77 standard*

As an extension to FORTRAN 77, the following forms of the DO loop are also supported. .autonote .note A DO loop with no statement label specified in the defining DO statement. .note The DO WHILE form of the DO statement. .endnote

DO-06 *expecting comma or DO variable*

The item following the DO keyword and the terminal statement-label (if present) must be either a comma or a DO variable. A DO variable is an integer, real or double precision variable name. The DO statement syntax is as follows: .millust begin DO <tsl> <,> DO-var = ex, ex <, ex> .millust end

DO-07 *DO variable cannot be redefined while DO loop is active*

The DO variable named in the DO statement cannot have its value altered by a statement in the DO loop structure.

DO-08 *incrementation parameter for DO-loop cannot be zero*

The third expression in the DO statement cannot be zero. This expression indicates the increment to the DO variable each iteration of the DO loop. If the increment expression is not specified a value of 1 is assumed.

B.12 Equivalence and/or Common

EC-01 *equivalencing %s1 has caused extension of COMMON block %s2 to the left*

The name %s1 has been equivalenced to a name in the COMMON block %s2. This relationship has caused the storage of the COMMON block to be extended to the left. FORTRAN 77 does not allow a COMMON block to be extended in this way.

EC-02 *%s1 and %s2 in COMMON are equivalenced to each other*

The names %s1 and %s2 appear in different COMMON blocks and each occupies its own piece of storage and therefore cannot be equivalenced.

B.13 END Statement

EN-01 *missing END statement*

The END statement for a PROGRAM, SUBROUTINE, FUNCTION or BLOCK DATA subprogram was not found before the next subprogram or the end of the source input stream.

B.14 Equal Sign

EQ-01 *target of assignment is illegal*

The target of an assignment statement, an input/output status specifier in an input/output statement, or an inquiry specifier in an INQUIRE statement, is illegal. The target in any of the above cases must be a variable name, array element, or a substring name.

EQ-02 *cannot assign value to %s1*

An attempt has been made to assign a value to a symbol with class %s1. For example, an array name cannot be the target of an assignment statement. This error may also be issued when an illegal target is used for the input/output status specifier in an input/output statement or an inquiry specifier in an INQUIRE statement.

EQ-03 *illegal use of equal sign*

An equal sign has been found in the statement but the statement is not an assignment statement.

EQ-04 *multiple assignment is not FORTRAN 77 standard*

More than one equal sign has been found in the assignment statement.

EQ-05 *expecting equals sign*

The equal sign is missing or misplaced. The PARAMETER statement uses an equal sign to equate a symbolic name to the value of a constant expression. The I/O statements use an equal sign to equate the appropriate values to the various specifiers. The DO statement uses an equal sign to assign the initial value to the DO variable.

B.15 Equivalenced Variables

EV-01 *%s1 has been equivalenced to 2 different relative positions*

The storage unit referenced by %s1 has been equivalenced to two different storage units starting in two different places. One name cannot be associated to two different values at the same time.

EV-02 *EQUIVALENCE list must contain at least 2 names*

The list of names to make a storage unit equivalent to several names must contain at least two names.

EV-03 *%s1 incorrectly subscripted in %s2 statement*

The name %s1 has been incorrectly subscripted in a %s2 statement.

EV-04 *incorrect substring of %s1 in %s2 statement*

An attempt has been made to incorrectly substring %s1 in a %s2 statement. For example, if a CHARACTER variable was declared to be of length 4 then (2:5) would be an invalid substring expression.

EV-05 *equivalencing CHARACTER and non-CHARACTER data is not FORTRAN 77 standard*

Equivalencing numeric and character data is an extension to the FORTRAN 77 language.

EV-06 *attempt to substring %s1 in EQUIVALENCE statement but type is %s2*

An attempt has been made to substring the symbolic name %s1 in an EQUIVALENCE statement but the type of the name is %s2 and should be of type CHARACTER.

B.16 Exponentiation

EX-01 *zero**J where J <= 0 is not allowed*

Zero cannot be raised to a power less than or equal to zero.

EX-02 *X**Y where X < 0.0, Y is not of type INTEGER, is not allowed*

When X is less than zero, Y may only be of type INTEGER.

EX-03 *(0,0)**Y where Y is not real is not allowed*

In complex exponentiation, when the base is zero, the exponent may only be a real number or a complex number whose imaginary part is zero.

B.17 ENTRY Statement

EY-01 *type of entry %s1 does not match type of function %s2*

If the type of a function is CHARACTER or a user-defined STRUCTURE, then the type of all entry names must match the type of the function name.

EY-02 *ENTRY statement not allowed within structured control blocks*

FORTRAN 77 does not allow an ENTRY statement to appear between the start and end of a control structure.

EY-03 *size of entry %s1 does not match size of function %s2*

The name %s1 found in an ENTRY statement must be declared to be the same size as that of the function name. If the name of the function or the name of any entry point has a length specification of (*), then all such entries must have a length specification of (*) otherwise they must all have a length specification of the same integer value.

B.18 Format

FM-01 *missing delimiter in format string, comma assumed*

The omission of a comma between the descriptors listed in a format string is an extension to the FORTRAN 77 language. Care should be taken when omitting the comma since the assumed separation may not occur in the intended place.

FM-02 *missing or invalid constant*

An unsigned integer constant was expected with the indicated edit descriptor but was not correctly placed or was missing.

FM-03 *Ew.dDe format code is not FORTRAN 77 standard*

The edit descriptor Ew.dDe is an extension to the FORTRAN 77 language.

FM-04 *missing decimal point*

The indicated edit descriptor must have a decimal point and an integer to indicate the number of decimal positions. These edit descriptors include: F, E, D and G.

FM-05 *missing or invalid edit descriptor in format string*

In the format string, two delimiters were found in succession with no valid descriptor in between.

FM-06 *unrecognizable edit descriptor in format string*

An edit descriptor has been found in the format string that is not a valid code. Valid codes are: apostrophe (’), I, F, E, D, G, L, A, Z, H, T, TL, TR, X, /, :, S, SP, SS, P, BN, B, $, and \.

FM-07 *invalid repeat specification*

The indicated repeatable edit descriptor is invalid. 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, Ew.dDe, and Zw where w and e are positive unsigned integer constants, and d and m are unsigned integer constants.

FM-08 *$ or \ format code is not FORTRAN 77 standard*

The non-repeatable edit descriptors $ and \ are extensions to the FORTRAN 77 language.

FM-09 *invalid field modifier*

The indicated edit descriptor for a field is incorrect. Consult the Language Reference for the correct form of the edit descriptor.

FM-10 *expecting end of FORMAT statement but found more text*

The right parenthesis was encountered in the FORMAT statement to terminate the statement and more text was found on the line.

FM-11 *repeat specification not allowed for this format code*

A repeat specification was found in front of a format code that is a nonrepeatable edit descriptor. These include: apostrophe, H, T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, $,and \.

FM-12 *no statement number on FORMAT statement*

The FORMAT statement must have a statement label. This statement number is used by I/O statements to reference the FORMAT statement.

FM-13 *no closing quote on apostrophe edit descriptor*

The closing quote of an apostrophe edit descriptor was not found.

FM-14 *field count greater than 256 is invalid*

The repeat specification of the indicated edit descriptor is greater than the maximum allowed of 256.

FM-15 *invalid field width specification*

The width specifier on the indicated edit descriptor is invalid.

FM-16 *Z format code is not FORTRAN 77 standard*

The Z (hexadecimal format) repeatable edit descriptor is an extension to the FORTRAN 77 language.

FM-17 *FORMAT statement exceeds allotted storage size*

The maximum allowable size of a FORMAT statement has exceeded. The statement must be split into two or more FORMAT statements.

FM-18 *format specification not allowed on input*

A format specification, in the FORMAT statement, is not allowed to be used as an input specification. Valid specifications include: T, TL,TR, X, /, :, P, BN, BZ, I, F, E, D, G, L, A, and Z.

FM-19 *FORMAT missing repeatable edit descriptor*

An attempt has been made to read or write a piece of data without a valid repeatable edit descriptor. All data requires a repeatable edit descriptor in the format. 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, Ew.dDe, and Zw where w and e are positive unsigned integer constants, and d and m are unsigned integer constants.

FM-20 *missing constant before X edit descriptor, 1 assumed*

The omission of the constant before an X edit descriptor in a format specification is an extension to the FORTRAN 77 language.

FM-21 *Ew.dQe format code is not FORTRAN 77 standard*

The edit descriptor Ew.dQe is an extension to the FORTRAN 77 language.

FM-22 *Qw.d format code is not FORTRAN 77 standard*

The edit descriptor Qw.d is an extension to the FORTRAN 77 language.

B.19 GOTO and ASSIGN Statements

GO-01 *%s1 statement label may not appear in ASSIGN statement but did in line %d2*

The statement label in the ASSIGN statement in line %d2 references a %s1 statement. The statement label in the ASSIGN statement must appear in the same program unit and must be that of an executable statement or a FORMAT statement.

GO-02 *ASSIGN of statement number %i1 in line %d2 not allowed*

The statement label %d1 in the ASSIGN statement is used in the line %d2 which references a non-executable statement. A statement label must appear in the same program unit as the ASSIGN statement and must be that of an executable statement or a FORMAT statement.

GO-03 *expecting TO*

The keyword TO is missing or misplaced in the ASSIGN statement.

B.20 Hollerith Constants

HO-01 *hollerith constant is not FORTRAN 77 standard*

Hollerith constants are an extension to the FORTRAN 77 language.

HO-02 *not enough characters for hollerith constant*

The number of characters following the H or h is not equal to the constant preceding the H or h. 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.

B.21 IF Statements

IF-01 *ELSE block must be the last block in block IF*

Another ELSE IF block has been found after the ELSE block. The ELSE block must be the last block in an IF block. The form of the block IF is as follows: .millust begin IF (logical expression) THEN [:block-label] {statement} { ELSE IF {statement} } [ELSE {statement}] ENDIF .millust end

IF-02 *expecting THEN*

The keyword THEN is missing or misplaced in the block IF statement. The form of the block IF is as follows: .millust begin IF (logical expression) THEN [:block-label] {statement} { ELSE IF {statement} } [ELSE {statement}] ENDIF .millust end

B.22 I/O Lists

IL-01 *missing or invalid format/FMT specification*

A valid format specification is required on all READ and WRITE statements. The format specification is specified by: .millust begin [FMT=] <format identifier> .millust end .pc .sy <format identifier> is one of the following: statement label, integer variable-name, character array-name, character expression, or *.

IL-02 *the UNIT may not be an internal file for this statement*

An internal file may only be referenced in a READ or WRITE statement. An internal file may not be referenced in a BACKSPACE, CLOSE, ENDFILE, INQUIRE, OPEN, or REWIND statement.

IL-03 *%s1 statement cannot have %s2 specification*

The I/O statement %s1 may not have the control information %s2 specified.

IL-04 *variable must have a size of 4*

The variable used as a specifier in an I/O statement must be of size 4 but another size was specified. These include the EXIST, OPENED, RECL, IOSTAT, NEXTREC, and NUMBER. The name used in the ASSIGN statement must also be of size 4 but a different size was specified.

IL-05 *missing or unrecognizable control list item %s1*

A control list item %s1 was encountered in an I/O statement and is not a valid control list item for that statement, or a control list item was expected and was not found.

IL-06 *attempt to specify control list item %s1 more than once*

The control list item %s1 in the indicated I/O statement, has been named more than once.

IL-07 *implied DO loop has no input/output list*

The implied DO loop specified in the I/O statement does not correspond with a variable or expression in the input/output list.

IL-08 *list-directed input/output with internal files is not FORTRAN 77 standard*

List-directed input/output with internal files is an extension to the FORTRAN 77 language.

IL-09 *FORTRAN 77 standard requires an asterisk for list-directed formatting*

An optional asterisk for list-directed formatting is an extension to the FORTRAN 77 language. The standard FORTRAN 77 language specifies that an asterisk is required.

IL-10 *missing or improper unit identification*

The control specifier, UNIT, in the I/O statement is either missing or identifies an improper unit. The unit specifier specifies an external unit or internal file. The external unit identifier is a non-negative integer expression or an asterisk. The internal file identifier is character variable, character array, character array element, or character substring.

IL-11 *missing unit identification or file specification*

An identifier to specifically identify the required file is missing. The UNIT specifier is used to identify the external unit or internal file. The FILE specifier in the INQUIRE and OPEN statements is used to identify the file name.

IL-12 *asterisk unit identifier not allowed in %s1 statement*

The BACKSPACE, CLOSE, ENDFILE, INQUIRE, OPEN, and REWIND statements require the external unit identifier be an unsigned positive integer from 0 to 999.

IL-13 *cannot have both UNIT and FILE specifier*

There are two valid forms of the INQUIRE statement; INQUIRE by FILE and INQUIRE by UNIT. Both of these specifiers cannot be specified in the same statement.

IL-14 *internal files require sequential access*

An attempt has been made to randomly access an internal file. Internal files may only be accessed sequentially.

IL-15 *END specifier with REC specifier is not FORTRAN 77 standard*

The FORTRAN 77 standard specifies that an end-of-file condition can only occur with a file connected for sequential access or an internal file. The REC specifier indicates that the file is connected for direct access. This extension allows the programmer to detect an end-of-file condition when reading the records sequentially from a file connected for direct access.

IL-16 *%s1 specifier in i/o list is not FORTRAN 77 standard*

The specified i/o list item is provided as an extension to the FORTRAN 77 language.

IL-17 *i/o list is not allowed with NAMELIST-directed format*

An i/o list is not allowed when the format specification is a NAMELIST.

IL-18 *non-character array as format specifier is not FORTRAN 77 standard*

A format specifier must be of type character unless it is an array name. Allowing a non-character array name is an extension to the FORTRAN 77 standard.

B.23 IMPLICIT Statements

IM-01 *illegal range of characters*

In the IMPLICIT statement, the first letter in the range of characters must be smaller in the collating sequence than the second letter in the range.

IM-02 *letter can only be implicitly declared once*

The indicated letter has been named more than once in this or a previous IMPLICIT statement. A letter may only be named once.

IM-03 *unrecognizable type*

The type declared in the IMPLICIT statement is not one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER.

IM-04 *(*) length specifier in an IMPLICIT statement is not FORTRAN 77 standard*

A character length specified of (*) in an IMPLICIT statement is an extension to the FORTRAN 77 language.

IM-05 *IMPLICIT NONE allowed once or not allowed with other IMPLICIT statements*

The IMPLICIT NONE statement must be the only IMPLICIT statement in the program unit in which it appears. Only one IMPLICIT NONE statement is allowed in a program unit.

B.24 Input/Output

IO-01 *BACKSPACE statement requires sequential access mode*

The file connected to the unit specified in the BACKSPACE statement has not been opened for sequential access.

IO-02 *input/output is already active*

An attempt has been made to read or write a record when there is an already active read or write in progress. The execution of a READ or WRITE statement has caused transfer to a function that contains a READ or WRITE statement.

IO-03 *ENDFILE statement requires sequential access mode*

The specified external unit identifier must be connected for sequential access but was connected for direct access.

IO-04 *formatted connection requires formatted input/output statements*

The FORM specifier in the OPEN statement specifies FORMATTED and the subsequent READ and/or WRITE statement does not use formatted I/O. If the FORM specifier has been omitted and access is SEQUENTIAL then FORMATTED is assumed. If the access is DIRECT then UNFORMATTED is assumed.

IO-05 *unformatted connection requires unformatted input/output statements*

The FORM specifier in the OPEN statement specifies UNFORMATTED and the subsequent READ and/or WRITE statement uses formatted I/O. If the FORM specifier has been omitted and access is SEQUENTIAL then FORMATTED is assumed. If the access is DIRECT then UNFORMATTED is assumed.

IO-06 *REWIND statement requires sequential access*

The external unit identifier is not connected to a sequential file. The REWIND statement positions to the first record in the file.

IO-07 *bad character in input field*

The data received from the record in a file does not match the type of the input list item.

IO-08 *BLANK specifier requires FORM specifier to be 'FORMATTED'*

In the OPEN statement, the BLANK specifier may only be used when the FORM specifier has the value of FORMATTED. The BLANK specifier indicates whether blanks are treated as zeroes or ignored.

IO-09 *system file error - %s1*

A system error has occurred while attempting to access a file. The I/O system error message is displayed.

IO-10 *format specification does not match data type*

A format specification in the FMT specifier or FORMAT statement specifies data of one type and the variable list specifies data of a different type.

IO-11 *input item does not match the data type of list variable*

In the READ statement, the data type of a variable listed is not of the same data type in the data file. For example, non-digit character data being read into an integer item.

IO-12 *internal file is full*

The internal file is full of data. If a file is a variable then the file may only contain one record. If the file is a character array then there can be one record for each array element.

IO-13 *RECL specifier is invalid*

In the OPEN statement, the record length specifier must be a positive integer expression.

IO-14 *invalid STATUS specifier in CLOSE statement*

The STATUS specifier can only have a value of KEEP or DELETE. If the STATUS in the OPEN statement is SCRATCH then the KEEP status on the CLOSE statement cannot be used.

IO-15 *unit specified is not connected*

The unit number specified in the I/O statement has not been previously connected.

IO-16 *attempt to perform data transfer past end of file*

An attempt has been made to read or write data after the end of file has been read or written.

IO-17 *invalid RECL specifier/ACCESS specifier combination*

In the OPEN statement, if the ACCESS specifier is DIRECT then the RECL specifier must be given.

IO-18 *REC specifier required in direct access input/output statements*

In the OPEN statement, the ACCESS specified was DIRECT. All subsequent input/output statements for that file must use the REC specifier to indicate which record to access.

IO-19 *REC specifier not allowed in sequential access input/output statements*

In the OPEN statement, the ACCESS specified was SEQUENTIAL. The REC specifier may not be used in subsequent I/O statements for that file. The REC specifier is used to indicate which record to access when access is DIRECT.

IO-20 *%s1 specifier may not change in a subsequent OPEN statement*

The %s1 specifier may not be changed on a subsequent OPEN statement for the same file, in the same program. Only the BLANK specifier may be changed.

IO-21 *invalid STATUS specifier for given file*

In the OPEN statement, the STATUS specifier does not match with the actual file status: OLD means the file must exist, NEW means the file must not exist. If the STATUS specifier is omitted, UNKNOWN is assumed.

IO-22 *invalid STATUS specifier/FILE specifier combination*

In the OPEN statement, if the STATUS is SCRATCH, the FILE specifier cannot be used. If the STATUS is NEW or OLD, the FILE specifier must be given.

IO-23 *record size exceeded during unformatted input/output*

This error is issued when the size of an i/o list item exceeds the maximum record size of the file. The record size can be specified using the RECL= specified in the OPEN statement.

IO-24 *unit specified does not exist*

The external unit identifier specified in the input/output statement has not yet been connected. Use preconnection or the OPEN statement to connect a file to the external unit identifier.

IO-25 *REC specifier is invalid*

The REC specifier must be an unsigned positive integer.

IO-26 *UNIT specifier is invalid*

The UNIT specifier must be an unsigned integer between 0 and 999 inclusive.

IO-27 *formatted record or format edit descriptor is too large for record size*

This error is issued when the amount of formatted data in a READ, WRITE or PRINT statement exceeds the maximum record size of the file. The record size can be specified using the RECL= specified in the OPEN statement.

IO-28 *illegal '%s1=' specifier*

In the OPEN or CLOSE statement the value associated with the %s1 specifier is not a valid value. In the OPEN statement, STATUS may only be one of OLD, NEW, SCRATCH or UNKNOWN; ACCESS may only be one of SEQUENTIAL, APPEND or DIRECT; FORM may only be one of FORMATTED or UNFORMATTED; CARRIAGECONTROL may only be one of YES or NO; RECORDTYPE may only be one of FIXED, TEXT or VARIABLE; ACTION may only be one of READ, WRITE or READ/WRITE; and BLANK may only be one of NULL, or ZERO. In the CLOSE statement the STATUS may only be one of KEEP or DELETE.

IO-29 *invalid CARRIAGECONTROL specifier/FORM specifier combination*

The CARRIAGECONTROL specifier is only allowed with formatted i/o statements.

IO-30 *i/o operation not consistent with file attributes*

An attempt was made to read from a file that was opened with ACTION=WRITE or write to a file that was opened with ACTION=READ. This message is also issued if you attempt to write to a read-only file or read from a write-only file.

IO-31 *symbol %s1 not found in NAMELIST*

During NAMELIST-directed input, a symbol was specified that does not belong to the NAMELIST group specified in the i/o statement.

IO-32 *syntax error during NAMELIST-directed input*

Bad input was encountered during NAMELIST-directed input. Data must be in a special form during NAMELIST-directed input.

IO-33 *subscripting error during NAMELIST-directed i/o*

An array was incorrectly subscripted during NAMELIST-directed input.

IO-34 *substring error during NAMELIST-directed i/o*

An character array element or variable was incorrectly substring during NAMELIST-directed input.

IO-35 *BLOCKSIZE specifier is invalid*

In the OPEN statement, the block size specifier must be a positive integer expression.

IO-36 *invalid operation for files with no record structure*

An attempt has been made to perform an i/o operation on a file that requires a record structure. For example, it is illegal to use a BACKSPACE statement for a file that has no record structure.

IO-37 *integer overflow converting character data to integer*

An overflow has occurred while converting the character data read to its internal representation as an integer.

IO-38 *range exceeded converting character data to floating-point*

An overflow or underflow has occurred while converting the character data read to its internal representation as a floating-point number.

B.25 Program Termination

KO-01 *floating-point divide by zero*

An attempt has been made to divide a number by zero in a floating-point expression.

KO-02 *floating-point overflow*

The floating-point expression result has exceeded the maximum floating-point number.

KO-03 *floating-point underflow*

The floating-point expression result has exceeded the minimum floating-point number.

KO-04 *integer divide by zero*

An attempt has been made to divide a number by zero in an integer expression.

KO-05 *program interrupted from keyboard*

The user has interrupted the compilation or execution of a program through use of the keyboard.

KO-06 *integer overflow*

The integer expression result has exceeded the maximum integer number.

KO-07 *maximum pages of output exceeded*

The specified maximum number of output pages has been exceeded. The maximum number of output pages can be increased by using the "pages=n" option in the command line or specifying C$PAGES=n in the source file.

KO-08 *statement count has been exceeded*

The maximum number of source statements has been executed. The maximum number of source statements that can be executed can be increased by using the "statements=n" option in the command line or specifying C$STATEMENTS=n in the source file.

KO-09 *time limit exceeded*

The maximum amount of time for program execution has been exceeded. The maximum amount of time can be increased by using the "time=t" option in the command line or specifying C$TIME=t in the source file.

B.26 Library Routines

LI-01 *argument must be greater than zero*

The argument to the intrinsic function must be greater than zero (i.e., a positive number).

LI-02 *absolute value of argument to arcsine, arccosine must not exceed one*

The absolute value of the argument to the intrinsic function ASIN or ACOS cannot be greater than or equal to the value 1.0.

LI-03 *argument must not be negative*

The argument to the intrinsic function must be greater than or equal to zero.

LI-04 *argument(s) must not be zero*

The argument(s) to the intrinsic function must not be zero.

LI-05 *argument of CHAR must be in the range zero to 255*

The argument to the intrinsic function CHAR must be in the range 0 to 255 inclusive. CHAR returns the character represented by an 8-bit pattern.

LI-06 *%s1 intrinsic function cannot be passed 2 complex arguments*

The second argument to the intrinsic function CMPLX and DCMPLX cannot be a complex number.

LI-07 *argument types must be the same for the %s1 intrinsic function*

The second argument to the intrinsic function CMPLX or DCMPLX must be of the same type as the first argument. The second argument may only be used when the first argument is of type INTEGER, REAL or DOUBLE PRECISION.

LI-08 *expecting numeric argument, but %s1 argument was found*

The argument to the intrinsic function, INT, REAL, DBLE, CMPLX, or DCMPLX was of type %s1 and a numeric argument was expected.

LI-09 *length of ICHAR argument greater than one*

The length of the argument to the intrinsic function ICHAR must be of type CHARACTER and length of 1. ICHAR converts a character to its integer representation.

LI-10 *cannot pass %s1 as argument to intrinsic function*

The item %s1 cannot be used as an argument to an intrinsic function. Only constants, simple variables, array elements, and substring array elements may be used as arguments.

LI-11 *intrinsic function requires argument(s)*

An attempt has been made to invoke an intrinsic function and no actual arguments were listed.

LI-12 *%s1 argument type is invalid for this generic function*

The type of the argument used in the generic intrinsic function is not correct.

LI-13 *this intrinsic function cannot be passed as an argument*

Only the specific name of the intrinsic function can be used as an actual argument. The generic name may not be used. When the generic and intrinsic names are the same, use the INTRINSIC statement.

LI-14 *expecting %s1 argument, but %s2 argument was found*

An argument of type %s2 was passed to a function but an argument of type %s1 was expected.

LI-15 *intrinsic function was assigned wrong type*

The declared type of an intrinsic function does not agree with the actual type.

LI-16 *intrinsic function %s1 is not FORTRAN 77 standard*

The specified intrinsic function is provided as an extension to the FORTRAN 77 language.

LI-17 *argument to ALLOCATED intrinsic function must be an allocatable array or character*(*) variable*

The argument to the intrinsic function ALLOCATED must be an allocatable array or character*(*) variable.

LI-18 *invalid argument to ISIZEOF intrinsic function*

The argument to the intrinsic function ISIZEOF must be a user-defined structure name, a symbol name, or a constant.

B.27 Mixed Mode

MD-01 *relational operator has a logical operand*

The operands of a relational expression must either be both arithmetic or both character expressions. The operand indicated is a logical expression.

MD-02 *mixing DOUBLE PRECISION and COMPLEX types is not FORTRAN 77 standard*

The mixing of items of type DOUBLE PRECISION and COMPLEX in an expression is an extension to the FORTRAN 77 language.

MD-03 *operator not expecting %s1 operands*

Operands of type %s1 cannot be used with the indicated operator. The operators **, /, *, +, and – may only have numeric type data. The operator // may only have character type data.

MD-04 *operator not expecting %s1 and %s2 operands*

Operands of conflicting type have been encountered. For example, in a relational expression, it is not possible to compare a character expression to an arithmetic expression. Also, the type of the left hand operand of the field selection operator must be a user-defined structure.

MD-05 *complex quantities can only be compared using .EQ. or .NE.*

Complex operands cannot be compared using less than (.LT.), less than or equal (.LE.), greater than (.GT.), or greater than or equal (.GE.) operators.

MD-06 *unary operator not expecting %s1 type*

The unary operators, + and –, may only be used with numeric types. The unary operator .NOT. may be used only with a logical or integer operand. The indicated operand was of type %s1 which is not one of the valid types.

MD-07 *logical operator with integer operands is not FORTRAN 77 standard*

Integer operands are permitted with the logical operators .AND., .OR., .EQV., .NEQV., .NOT. and .XOR. as an extension to the FORTRAN 77 language.

MD-08 *logical operator %s1 is not FORTRAN 77 standard*

The specified logical operator is an extension to the FORTRAN 77 standard.

B.28 Memory Overflow

MO-01 *%s1 exceeds compiler limit of %u2 bytes*

An internal compiler limit has been reached. %s1 describes the limit and %d2 specifies the limit.

MO-02 *out of memory*

All available memory has been used up. During the compilation phase, memory is primarily used for the symbol table. During execution, memory is used for file descriptors and buffers, and dynamically allocatable arrays and character*(*) variables.

MO-03 *dynamic memory exhausted due to length of this statement - statement ignored*

There was not enough memory to encode the specified statement. This message is usually issued when the compiler is low on memory or if the statement is a very large statement that spans many continuation lines. This error does not terminate the compiler since it may have been caused by a very large statement. The compiler attempts to compile the remaining statements.

MO-04 *attempt to deallocate an unallocated array or character*(*) variable*

An attempt has been made to deallocate an array that has not been previously allocated. An array or character*(*) variable must be allocated using an ALLOCATE statement.

MO-05 *attempt to allocate an already allocated array or character*(*) variable*

An attempt has been made to allocate an array or character*(*) variable that has been previously allocated in an ALLOCATE statement.

MO-06 *object memory exhausted*

The amount of object code generated for the program has exceeded the amount of memory allocated to store the object code. The "/codesize" option can be used to increase the amount of memory allocated for object code.

B.29 Parentheses

PC-01 *missing or misplaced closing parenthesis*

An opening parenthesis '(' was found but no matching closing parenthesis ')' was found before the end of the statement.

PC-02 *missing or misplaced opening parenthesis*

A closing parenthesis ')' was found before the matching opening parenthesis '('.

PC-03 *unexpected parenthesis*

A parenthesis was found in a statement where parentheses are not expected.

PC-04 *unmatched parentheses*

The parentheses in the expression are not balanced.

B.30 PRAGMA Compiler Directive

PR-01 *expecting symbolic name*

Every auxiliary pragma must refer to a symbol. This error is issued when the symbolic name is illegal or missing. Valid symbolic names are formed from the following characters: a dollar sign, an underscore, digits and any letter of the alphabet. The first character of a symbolic name must be alphabetic, a dollar sign, or an underscore.

PR-02 *illegal size specified for VALUE attribute*

The VALUE argument attribute of an auxiliary pragma contains in illegal length specification. Valid length specifications are 1, 2, 4 and 8.

PR-03 *illegal argument attribute*

An illegal argument attribute was specified. Valid argument attributes are VALUE, REFERENCE, or DATA_REFERENCE.

PR-04 *continuation line must contain a comment character in column 1*

When continuing a line of an auxiliary pragma directive, the continued line must end with a back-slash ('\') character and the continuation line must begin with a comment character ('c', 'C' or '*') in column 1.

PR-05 *expecting '%s1' near '%s2'*

A syntax error was found while processing a PRAGMA directive. %s1 identifies the expected information and %s2 identifies where in the pragma the error occurred.

PR-06 *in-line byte sequence limit exceeded*

The limit on the number of bytes of code that can be generated in-line using a an auxiliary pragma has been exceeded. The limit is 127 bytes.

PR-07 *illegal hexadecimal data in byte sequence*

An illegal hexadecimal constant was encountered while processing a in-line byte sequence of an auxiliary pragma. Valid hexadecimal constants in an in-line byte sequence must begin with the letter Z or z and followed by a string of hexadecimal digits.

PR-08 *symbol '%s1' in in-line assembly code cannot be resolved*

The symbol %s1, referenced in an assembly language instruction in an auxiliary pragma, could not be resolved.

B.31 RETURN Statement

RE-01 *alternate return specifier only allowed in subroutine*

An alternate return specifier, in the RETURN statement, may only be specified when returning from a subroutine.

RE-02 *RETURN statement in main program is not FORTRAN 77 standard*

A RETURN statement in the main program is allowed as an extension to the FORTRAN 77 standard.

B.32 SAVE Statement

SA-01 *COMMON block %s1 saved but not properly defined*

The named COMMON block %s1 was listed in a SAVE statement but there is no named COMMON block defined by that name.

SA-02 *COMMON block %s1 must be saved in every subprogram in which it appears*

The named COMMON block %s1 appears in a SAVE statement in another subprogram and is not in a SAVE statement in this subprogram. 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.

SA-03 *name already appeared in a previous SAVE statement*

The indicated name has already been referenced in another SAVE statement in this subprogram.

B.33 Statement Functions

SF-01 *statement function definition contains duplicate dummy arguments*

A dummy argument is repeated in the argument list of the statement function.

SF-02 *character length of statement function name must not be (*)*

If the type of a character function is character, its length specification must not be (*); it must be a constant integer expression.

SF-03 *statement function definition contains illegal dummy argument*

A dummy argument of type CHARACTER must have a length specification of an integer constant expression that is not (*).

SF-04 *cannot pass %s1 %s2 to statement function*

The actual arguments 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.

SF-05 *%s1 actual argument was passed to %s2 dummy argument*

The indicated actual argument is of type %s1 which is not the same type as that of the dummy argument of type %s2.

SF-06 *incorrect number of arguments passed to statement function %s1*

The number of arguments passed to statement function %s1 does not agree with the number of dummy arguments specified in its definition.

SF-07 *type of statement function name must not be a user-defined structure*

The type of a statement function cannot be a user-defined structure. Valid types for statement functions are: LOGICAL*1, LOGICAL, INTEGER*1, INTEGER*2, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, and CHARACTER. If the statement function is of type CHARACTER, its length specification must not be (*); it must be an integer constant expression.

B.34 Source Management

SM-01 *system file error reading %s1 - %s2*

An I/O error, described by %s2, has occurred while reading the FORTRAN source file %s1.

SM-02 *error opening file %s1 - %s2*

The FORTRAN source file %s1 could not be opened. The error is described by %s2.

SM-03 *system file error writing %s1 - %s2*

An I/O error, described by %s2, has occurred while writing to the file %s1.

SM-04 *error spawning %s1 - %s2*

An error, described by %s2, occurred while trying to spawn the external program named %s1.

SM-05 *error while linking*

An error occurred while trying to create the executable file. See the WLINK documentation for a description of the error.

SM-06 *error opening %s1 - too many temporary files exist*

The compiler was not able to open a temporary file for intermediate storage during code generation. Temporary files are created in the directory specified by the TMP environment variable. If the TMP environment variable is not set, the temporary file is created in the current directory. This error is issued if an non-existent directory is specified in the TMP environment variable, or more than 26 concurrent compiles are taking place in a multi-tasking environment and the directory in which the temporary files are created is the same for all compilation processes.

SM-07 *generation of browsing information failed*

An error occurred during the generation of browsing information. For example, a disk full condition encountered during the creation of the browser module file will cause this message to be issued. Browsing information is generated when the /db switch is specified.

B.35 Structured Programming Features

SP-01 *cannot have both ATEND and the END= specifier*

It is not valid to use the AT END control statement and the END= option on the READ statement. Only one method can be used to control the end-of-file condition.

SP-02 *ATEND must immediately follow a READ statement*

The indicated AT END control statement or block does not immediately follow a READ statement. The AT END control statement or block is executed when an end-of-file condition is encountered during the read.

SP-03 *block label must be a symbolic name*

The indicated block label must be a symbolic name. A symbolic name must start with a letter and contain no more than 32 letters and digits. A letter is an upper or lower case letter of the alphabet, a dollar sign ($), or an underscore (_). A digit is a character in the range ’0’ to ’9’.

SP-04 *could not find a structure to %s1 from*

This message is issued in the following cases. .autonote .note There is no control structure to QUIT from. The QUIT statement will transfer control to the statement following the currently active control structure or return from a REMOTE BLOCK if no other control structures are active within the REMOTE BLOCK. .note There is no control structure to EXIT from. The EXIT statement is used to exit a loop-processing structure such as DO, DO WHILE, WHILE and LOOP, to return from a REMOTE BLOCK regardless of the number of active control structures within the REMOTE BLOCK, or to transfer control from a GUESS or ADMIT block to the statement following the ENDGUESS statement. .note There is no active looping control structure from which a CYCLE statement can be used. A CYCLE statement can only be used within a DO, DO WHILE, WHILE and LOOP control structure. .endnote

SP-05 *REMOTE BLOCK is not allowed in the range of any control structure*

An attempt has been made to define a REMOTE BLOCK inside a control structure. Control structures include IF, LOOP, WHILE, DO, SELECT and GUESS. When a REMOTE BLOCK definition is encountered during execution, control is transferred to the statement following the corresponding END BLOCK statement.

SP-06 *the SELECT statement must be followed immediately by a CASE statement*

The statement immediately after the SELECT statement must be a CASE statement. The SELECT statement 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.

SP-07 *cases are overlapping*

The case lists specified in the CASE statements in the SELECT control structure are in conflict. Each case list must specify a unique integer constant expression or range.

SP-08 *select structure requires at least one CASE statement*

In the SELECT control structure, there must be at least one CASE statement.

SP-09 *cannot branch to %i1 from outside control structure in line %d2*

The statement in line %d2 passes control to the statement %d1 in a control structure. Control may only be passed out of a control structure or to another place in that control structure. Control structures include DO, GUESS, IF, LOOP, SELECT, and WHILE.

SP-10 *cannot branch to %i1 inside structure on line %d2*

The statement attempts to pass control to statement %d1 in line %d2 which is in a control structure. Control may only be passed out of a control structure or to another place in that control structure. Control structures include DO, GUESS, IF, LOOP, SELECT, and WHILE.

SP-11 *low end of range exceeds the high end*

The first number, the low end of the range, is greater than the second number, the high end of the range.

SP-12 *default case block must follow all case blocks*

The default case block in the SELECT control structure must be the last case block. A case block may not follow the default case block.

SP-13 *attempt to branch out of a REMOTE BLOCK*

An attempt has been made to transfer execution control out of a REMOTE BLOCK. A REMOTE BLOCK may only be terminated with the END BLOCK statement. Execution of a REMOTE BLOCK is similar in concept to execution of a subroutine.

SP-14 *attempt to EXECUTE undefined REMOTE BLOCK %s1*

The REMOTE BLOCK %s1 referenced in the EXECUTE statement does not exist in the current program unit. A REMOTE BLOCK is local to the program unit in which it is defined and may not be referenced from another program unit.

SP-15 *attempted to use REMOTE BLOCK recursively*

An attempt was made to execute a REMOTE BLOCK which was already active.

SP-16 *cannot RETURN from subprogram within a REMOTE BLOCK*

An illegal attempt has been made to execute a RETURN statement within a REMOTE BLOCK in a subprogram.

SP-17 *%s1 statement is not FORTRAN 77 standard*

The statement %s1 is an extension to the FORTRAN 77 language.

SP-18 *%s1 block is unfinished*

The block starting with the statement %s1 does not have the ending block statement. For example: ATENDDO-ENDATEND, DO-ENDDO, GUESS-ENDGUESS, IF-ENDIF, LOOP-ENDLOOP, SELECT-ENDSELECT, STRUCTURE-ENDSTRUCTURE and WHILE-ENDWHILE.

SP-19 *%s1 statement does not match with %s2 statement*

The statement %s1, which ends a control structure, cannot be used with statement %s2 to form a control structure. Valid control structures are: LOOP - ENDLOOP, LOOP - UNTIL, WHILE - ENDWHILE, and WHILE - UNTIL.

SP-20 *incomplete control structure found at %s1 statement*

The ending control structure statement %s1 was found and there was no preceding matching beginning statement. Valid control structures include: ATENDDO - ENDATEND, GUESS - ENDGUESS, IF - ENDIF, LOOP - ENDLOOP, REMOTE BLOCK - ENDBLOCK, and SELECT - ENDSELECT.

SP-21 *%s1 statement is not allowed in %s2 definition*

Statement %s1 is not allowed between a %s2 statement and the corresponding END %s2 statement. For example, an EXTERNAL statement is not allowed between a STRUCTURE and END STRUCTURE statement, a UNION and END UNION statement, or a MAP and END MAP statement.

SP-22 *no such field name found in structure %s1*

A structure reference contained a field name that does not belong to the specified structure.

SP-23 *multiple definition of field name %s1*

The field name %s1 has already been defined in a structure.

SP-24 *structure %s1 has not been defined*

An attempt has been made to declare a symbol of user-defined type %s1. No structure definition for %s1 has occurred.

SP-25 *structure %s1 has already been defined*

The specified structure has already been defined as a structure.

SP-26 *structure %s1 must contain at least one field*

Structures must contain at least one field definition.

SP-27 *recursion detected in definition of structure %s1*

Structure %s1 has been defined recursively. For example, it is illegal for structure X to contain a field that is itself a structure named X.

SP-28 *illegal use of structure %s1 containing union*

Structures containing unions cannot be used in formatted I/O statements or data initialized.

SP-29 *allocatable arrays cannot be fields within structures*

An allocatable array cannot appear as a field name within a structure definition.

SP-30 *an integer conditional expression is not FORTRAN 77 standard*

A conditional expression is the expression that is evaluated and checked to determine a path of execution. A conditional expression can be found in an IF or WHILE statement. FORTRAN 77 requires that the conditional expression be a logical expression. As an extension, an integer expression is also allowed. When an integer expression is used, it is converted to a logical expression by comparing the value of the integer expression to zero.

SP-31 *%s1 statement must be used within %s2 definition*

The statement identified by %s1 must appear within a definition identified by %s2.

B.36 Subprograms

SR-01 *name can only appear in an EXTERNAL statement once*

A function/subroutine name appears more than once in an EXTERNAL statement.

SR-02 *character function %s1 may not be called since size was declared as (*)*

In the declaration of the character function name, the length was defined to be (*). The (*) length specification is only allowed for external functions, dummy arguments or symbolic character constants.

SR-03 *%s1 can only be used an an argument to a subroutine*

The specified class of an argument must only be passed to a subroutine. For example, an alternate return specifier is illegal as a subscript or an argument to a function.

SR-04 *name cannot appear in both an INTRINSIC and EXTERNAL statement*

The same name appears in an INTRINSIC statement and in an EXTERNAL statement.

SR-05 *expecting a subroutine name*

The subroutine named in the CALL statement does not define a subroutine. A subroutine is declared in a SUBROUTINE statement.

SR-06 *%s1 statement not allowed in main program*

The main program can contain any statements except a FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY statement. A SAVE statement is allowed but has no effect in the main program. A RETURN statement in the main program is an extension to the FORTRAN 77 language.

SR-07 *not an intrinsic FUNCTION name*

A name in the INTRINSIC statement is not an intrinsic function name. Refer to the Language Reference for a complete list of the intrinsic functions.

SR-08 *name can only appear in an INTRINSIC statement once*

An intrinsic function name appears more than once in the intrinsic function list.

SR-09 *subprogram recursion detected*

An attempt has been made to recursively invoke a subprogram, that is, to invoke an already active subprogram.

SR-10 *two main program units in the same file*

There are two places in the program that signify the start of a main program. The PROGRAM statement or the first statement that is not enclosed by a PROGRAM, FUNCTION, SUBROUTINE or BLOCK DATA statement specifies the main program start.

SR-11 *only one unnamed %s1 is allowed in an executable program*

There may only be one unnamed BLOCK DATA subprogram or main program in an executable program.

SR-12 *function referenced as a subroutine*

An attempt has been made to invoke a function using the CALL statement.

SR-13 *attempt to invoke active function/subroutine*

An attempt has been made to invoke the current function/subroutine or a function/subroutine that was used to invoke current function/subroutine. The traceback produced when the error occurred lists all currently active functions/subroutines.

SR-14 *dummy argument %s1 is not in dummy argument list of entered subprogram*

The named dummy argument found in the ENTRY statement does not appear in the subroutine's dummy argument list in the subprogram statement.

SR-15 *function referenced as %s1 but defined to be %s2*

An attempt has been made to invoke a function of the type %s1 but the function was defined as %s2 in the FUNCTION or ENTRY statement. The function name's type must be correctly declared in the main program.

SR-16 *function referenced as CHARACTER*%u1 but defined to be CHARACTER*%u2*

The character length of the function in the calling subprogram is %d1 but the length used to define the function is %d2. These two lengths must match.

SR-17 *missing main program*

The program file is either empty or contains only subroutines and functions. Each program require a main program. A main program starts with an optional PROGRAM statement and ends with an END statement.

SR-18 *subroutine referenced as a function*

An attempt has been made to invoke a name as a function and has been defined as a subroutine in a SUBROUTINE or ENTRY statement.

SR-19 *attempt to invoke a block data subprogram*

An attempt has been made to invoke a block data subprogram. Block data subprograms are used to initialize variables before program execution commences.

SR-20 *structure type of function %s1 does not match expected structure type*

The function returns a structure that is not equivalent to the structure expected. Two structures are equivalent if the types and orders of each field are the same. Unions are considered equivalent if their sizes are the same. Field names, and the structure name itself, do not have to be the same.

B.37 Subscripts and Substrings

SS-01 *substringing of function or statement function return value is not FORTRAN 77 standard*

The character value returned from a CHARACTER function or statement function cannot be substring. Only character variable names and array element names may be substring.

SS-02 *substringing valid only for character variables and array elements*

An attempt has been made to substring a name that is not defined to be of type CHARACTER and is neither a variable nor an array element.

SS-03 *subscript expression out of range; %s1 does not exist*

An attempt has been made to reference an element in an array that is out of bounds of the declared array size. The array element %s1 does not exist.

SS-04 *substring expression (%i1:%i2) is out of range*

An expression in the substring is larger than the string length or less than the value 1. The substring expression must be one in which .millust begin 1 <= %d1 <= %d2 <= len .millust end

B.38 Statements and Statement Numbers

ST-01 *statement number %i1 has already been defined in line %d2*

The two statements, in line %d2 and the current line, in the current program unit have the same statement label number, namely %d1.

ST-02 *statement function definition appears after first executable statement*

There is a statement function definition after the first executable statement in the program unit. Statement function definitions must follow specification statements and precede executable statements. Check that the statement function name is not an undeclared array name.

ST-03 *%s1 statement must not be branched to but was in line %d2*

Line %d2 passed execution control down to the statement %s1. The specification statements, ADMIT, AT END, BLOCK DATA, CASE, ELSE, ELSE IF, END AT END, END BLOCK, END DO, END LOOP, END SELECT, END WHILE, ENTRY, FORMAT, FUNCTION, OTHERWISE, PROGRAM, QUIT, REMOTE BLOCK, SAVE, SUBROUTINE, and UNTIL statements may not have control of execution transferred to it.

ST-04 *branch to statement %i1 in line %d2 not allowed*

An attempt has been made to pass execution control up to the statement labelled %d1 in line %d2. The specification statements, ADMIT, AT END, BLOCK DATA, CASE, ELSE, ELSE IF, END AT END, END BLOCK, END DO, END LOOP, END SELECT, END WHILE, ENTRY, FORMAT, FUNCTION, OTHERWISE, PROGRAM, QUIT, REMOTE BLOCK, SAVE, SUBROUTINE, and UNTIL statements may not have control of execution transferred to it.

ST-05 *specification statement must appear before %s1 is initialized*

The variable %s1 has been initialized in a specification statement. A COMMON or EQUIVALENCE statement then references the variable. The COMMON or EQUIVALENCE statement must appear before the item can be initialized. Use the DATA statement to initialize data in this case.

ST-06 *statement %i1 was referenced as a FORMAT statement in line %d2*

The statement in line %d2 references statement label %d1 as a FORMAT statement. The statement at that label is not a FORMAT statement.

ST-07 *IMPLICIT statement appears too late*

The current IMPLICIT statement is out of order. The IMPLICIT statement may be interspersed with the PARAMETER statement but must appear before other specification statements.

ST-08 *this statement will never be executed due to the preceding branch*

Because execution control will always be passed around the indicated statement, the statement will never be executed.

ST-09 *expecting statement number*

The keyword GOTO or ASSIGN has been detected and the next part of the statement was not a statement number as was expected.

ST-10 *statement number %i1 was not used as a FORMAT statement in line %d2*

The statement at line %d2 with statement number %d1 is not a FORMAT statement but the current statement uses statement number %d1 as if it labelled a FORMAT statement.

ST-11 *specification statement appears too late*

The indicated specification statement appears after a statement function definition or an executable statement. All specification statements must appear before these types of statements.

ST-12 *%s1 statement not allowed after %s2 statement*

The statement %s1 cannot be the object of a %s2 statement. %s2 represents a logical IF or WHILE statement. These statements include: specification statements, ADMIT, AT END, CASE, DO, ELSE, ELSE IF END, END AT END, END BLOCK, END DO, END GUESS, ENDIF, END LOOP, END SELECT, END WHILE, ENTRY, FORMAT, FUNCTION, GUESS, logical IF, block IF, LOOP, OTHERWISE, PROGRAM, REMOTE BLOCK, SAVE, SELECT, SUBROUTINE, UNTIL, and WHILE.

ST-13 *statement number must be 99999 or less*

The statement label number specified in the indicated statement has more than 5 digits.

ST-14 *statement number cannot be zero*

The statement label number specified in the indicated statement is zero. Statement label numbers must be greater than 0 and less than or equal to 99999.

ST-15 *this statement could branch to itself*

The indicated statement refers to a statement label number which appears on the statement itself and therefore could branch to itself, creating an endless loop.

ST-16 *missing statement number %i1 - used in line %d2*

A statement with the statement label number %d1 does not exist in the current program unit. The statement label number is referenced in line %d2 of the program unit.

ST-17 *undecodeable statement or misspelled word %s1*

The statement cannot be identified as an assignment statement or any other type of FORTRAN statement. The first word of a FORTRAN statement must be a statement keyword or the statement must be an assignment statement.

ST-18 *statement %i1 will never be executed due to the preceding branch*

The statement with the statement label number of %d1 will never be executed because the preceding statement will always pass execution control around the statement and no other reference is made to the statement label.

ST-19 *expecting keyword or symbolic name*

The first character of a statement is not an alphabetic. The first word of a statement must be a statement keyword or a symbolic name. Symbolic names must start with a letter (upper case or lower case), a dollar sign ($) or an underscore (_).

ST-20 *number in %s1 statement is longer than 5 digits*

The number in the PAUSE or STOP statement is longer than 5 digits.

ST-21 *position of DATA statement is not FORTRAN 77 standard*

The FORTRAN 77 standard requires DATA statements to appear after all specification statements. As an extension to the standard, Watcom FORTRAN 77 allows DATA statements to appear before specification statements. Note that in the latter case, the type of the symbol must be established before data initialization occurs.

ST-22 *no FORMAT statement with given label*

The current statement refers to the label of a FORMAT statement but the label appears on some other statement that is not a FORMAT statement.

ST-23 *statement number not in list or not the label of an executable statement*

The specified statement number in the indicated statement is not in the list of statement numbers or it is not the statement label number of an executable statement.

ST-24 *attempt to branch into a control structure*

An attempt has been made to pass execution control into a control structure. A statement uses a computed statement label number to transfer control. This value references a statement inside a control structure.

B.39 Subscripted Variables

SV-01 *variable %s1 in array declarator must be in COMMON or a dummy argument*

The variable %s1 was used as an array declarator in a subroutine or function but the variable was not in a COMMON block nor was it a dummy argument in the FUNCTION, SUBROUTINE or ENTRY statement.

SV-02 *adjustable/assumed size array %s1 must be a dummy argument*

The array %s1 used in the current subroutine or function must be a dummy argument. When the array declarator is adjustable or assumed-size, the array name must be a dummy argument.

SV-03 *invalid subscript expression*

The indicated subscript expression is not a valid integer expression or the high bound of the array is less than the low bound of the array when declaring the size of the array.

SV-04 *invalid number of subscripts*

The number of subscripts used to describe an array element does not match the number of subscripts in the array declaration. The maximum number of subscripts allowed is 7.

SV-05 *using %s1 name incorrectly without list*

An attempt has been made to assign a value to the declared array %s1. Values may only be assigned to elements in the array. An array element is the array name followed by integer expressions enclosed in parentheses and separated by commas.

SV-06 *cannot substring array name %s1*

An attempt has been made to substring the array %s1. Only an array element may be substring.

SV-07 *%s1 treated as an assumed size array*

A dummy array argument has been declared with 1 in the last dimension. The array is treated as if an '*' had been specified in place of the 1. This is done to support a feature called "pseudo-variable dimensioning" which was supported by some FORTRAN IV compilers and is identical in concept to FORTRAN 77 assumed-size arrays.

SV-08 *assumed size array %s1 cannot be used as an i/o list item or a format/unit identifier*

Assumed size arrays (arrays whose last dimension is '*') must not appear as an i/o list item (i.e. in a PRINT statement), a format identifier or an internal file specifier.

SV-09 *limit of 65535 elements per dimension has been exceeded*

On the IBM PC, for 16-bit real mode applications, the number of elements in a dimension must not exceed 65535.

B.40 Syntax Errors

SX-01 *unexpected number or name %s1*

The number or name %s1 is in an unexpected place in the statement.

SX-02 *bad sequence of operators*

The indicated arithmetic operator is out of order. An arithmetic operator is one of the following: **, *, /, +, and −. All arithmetic operators must be followed by at least a primary. A primary is an array element, constant, (expression), function name, or variable name.

SX-03 *invalid operator*

The indicated operator between the two arithmetic primaries is not a valid operator. Valid arithmetic operators include: **, *, /, +, and −. A primary is an array element, constant, (expression), function name, or variable name.

SX-04 *expecting end of statement after right parenthesis*

The end of the statement is indicated by the closing right parenthesis but more characters were found on the line. Multiple statements per line are not allowed in FORTRAN 77.

SX-05 *expecting an asterisk*

The next character of the statement should be an asterisk but another character was found instead.

SX-06 *expecting colon*

A colon (:) was expecting but not found. For example, the colon separating the low and high bounds of a character substring was not found.

SX-07 *expecting colon or end of statement*

On a control statement, a word was found at the end of the statement that cannot be related to the statement. The last word on several of the control statements may be a block label. All block labels must be preceded by a colon (:).

SX-08 *missing comma*

A comma was expected and is missing. There must be a comma after the statement keyword AT END when a statement follows. A comma must occur between the two statement labels in the GO TO statement. A comma must occur between the expressions in the DO statement. A comma must occur between the names listed in the DATA statement and specification statements. A comma must occur between the specifiers in I/O statements.

SX-09 *expecting end of statement*

The end of the statement was expected but more words were found on the line and cannot be associated to the statement. FORTRAN 77 only allows for one statement per line.

SX-10 *expecting integer variable*

The name indicated in the statement must be of type INTEGER but is not.

SX-11 *expecting %s1 name*

A name with the characteristic %s1 was expected at the indicated place in the statement but is missing.

SX-12 *expecting an integer*

The length specifier, as in the IMPLICIT statement, must be an integer constant or an integer constant expression. The repeat specifier of the value to be assigned to the variables, as in the DATA statement, must be an integer constant or an integer constant expression.

SX-13 *expecting INTEGER, REAL, or DOUBLE PRECISION variable*

The indicated DO variable is not one of the types INTEGER, REAL, or DOUBLE PRECISION.

SX-14 *missing operator*

Two primaries were found in an expression and an operator was not found in between. A primary is an array element, constant, (expression), function name, or variable name.

SX-15 *expecting a slash*

A slash is expected in the indicated place in the statement. Slashes must be balanced as parentheses. Slashes are used to enclose the initial data values in specification statements or to enclose names of COMMON blocks.

SX-16 *expecting %s1 expression*

An expression of type %s1 is required.

SX-17 *expecting a constant expression*

A constant expression is required.

SX-18 *expecting INTEGER, REAL, or DOUBLE PRECISION expression*

The indicated expression is not one of type INTEGER, REAL, or DOUBLE PRECISION. Each expression following the DO variable must be an expression of one of these types.

SX-19 *expecting INTEGER or CHARACTER constant*

In the PAUSE and STOP statement, the name following the keyword must be a constant of type INTEGER or of type CHARACTER. This constant will be printed on the console when the statement is executed.

SX-20 *unexpected operator*

An operand was expected but none was found. For example, in an I/O statement, the comma is used to separate I/O list items. Two consecutive commas without an I/O list item between them would result in this error.

SX-21 *no closing quote on literal string*

The closing quote of a literal string was not found before the end of the statement.

SX-22 *missing or invalid constant*

In a DATA statement, the constant required to initialize a variable was not found or incorrectly specified.

SX-23 *expecting character constant*

A character constant is required.

B.41 Type Statements

TY-01 *length specification before array declarator is not FORTRAN 77 standard*

An array declarator specified immediately after the length specification of the array is an extension to the FORTRAN 77 language.

TY-02 *%i1 is an illegal length for %s2 type*

The length specifier %d1 is not valid for the type %s2. For type LOGICAL, valid lengths are 1 and 4. For the type INTEGER, valid lengths are 1, 2, and 4. For the type REAL, valid lengths are 4 and 8. For the type COMPLEX, valid lengths are 8 and 16. On the IBM PC, the length specifier for items of type CHARACTER must be greater than 0 and not exceed 65535.

TY-03 *length specifier in %s1 statement is not FORTRAN 77 standard*

A length specifier in certain type specification statements is an extension to the FORTRAN 77 language. These include: LOGICAL*1, LOGICAL*4, INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16.

TY-04 *length specification not allowed with type %s1*

A length specification is not allowed in a DOUBLE PRECISION or DOUBLE COMPLEX statement.

TY-05 *type of %s1 has already been established as %s2*

The indicated name %s1 has already been declared to have a different type, namely %s2. The name %s1 cannot be used in this specification statement.

TY-06 *type of %s1 has not been declared*

The indicated name %s1 has not been declared. This message is only issued when the IMPLICIT NONE specification statement is used.

TY-07 *%s1 of type %s2 is illegal in %s3 statement*

The symbol %s1 with type %s2 cannot be used in statement %s3. For example, a symbol of type STRUCTURE cannot be used in a PARAMETER statement.

B.42 Variable Names

VA-01 *illegal use of %s1 name %s2 in %s3 statement*

The name %s2 has been defined as %s1 and cannot be used as a name in the statement %s3.

VA-02 *symbolic name %s1 is longer than 6 characters*

Symbolic names greater than 6 characters is an extension to the FORTRAN 77 language. The maximum length is 32 characters.

VA-03 *%s1 has already been defined as a %s2*

The name %s1 has been previously defined as a %s2 in another statement and cannot be redefined as specified in the indicated statement.

VA-04 *%s1 %s2 has not been defined*

The name %s2 has been referenced to be a %s1 but has not been defined as such in the program unit.

VA-05 *%s1 is an unreferenced symbol*

The name %s1 has been defined but not referenced.

VA-06 *%s1 already belongs to this NAMELIST group*

The name %s1 can only appear in a NAMELIST group once. However, a name can belong to multiple NAMELIST groups.

VA-07 *%s1 has been used but not defined*

%s1 has not been defined before using it in a way that requires its definition. Note that symbols that are equivalenced, belong to a common block, are dummy arguments, or passed as an argument to a subprogram, will not be checked to ensure that they have been defined before requiring a value.

VA-08 *dynamically allocating %s1 is not FORTRAN 77 standard*

Allocatable storage are extensions to the FORTRAN 77 standard.

VA-09 *%s1 in NAMELIST %s2 is illegal*

Symbol %s1 appearing in NAMELIST %s2 is illegal. Symbols appearing in a NAMELIST cannot be dummy arguments, allocatable, or of a user-defined type.

3

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X