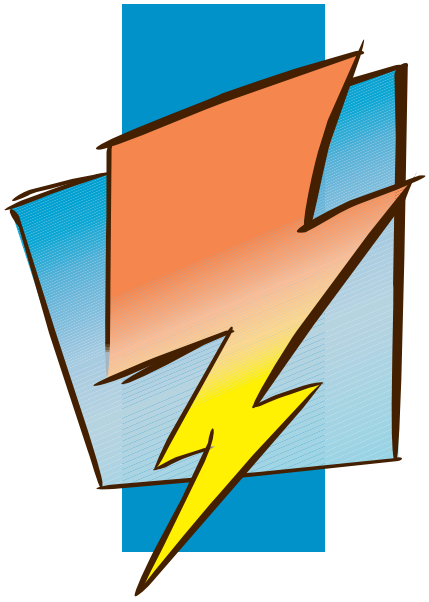
Open Watcom C/C++ Tools

User's Guide



Version 2.0

Open Watcom

Notice of Copyright

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

Preface

The *Open Watcom C/C++ Tools User's Guide* describes how to use Open Watcom's software development tools on Intel 80x86-based personal computers with DOS, Windows, or OS/2. The *Open Watcom C/C++ Tools User's Guide* describes the following tools:

- compile and link utility
- assembler
- object file library manager
- object file disassembler
- exe2bin utility
- far call optimization utility
- patch utility
- executable file strip utility
- make utility
- touch utility
- ide2make utility

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.

July, 1997.

Trademarks Used in this Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

The Open Watcom Compile and Link Utility

1 The Open Watcom C/C++ Compile and Link Utility

The Open Watcom C/C++ Compile and Link Utility is designed for generating applications, simply and quickly, using a single command line. On the command line, you can list source file names as well as object file names. Source files are either compiled or assembled based on file extension; object files and libraries are simply included in the link phase. Options can be passed on to both the compiler and linker.

1.1 WCL/WCL386 Command Line Format

The format of the command line is:

WCL [files] [options]
WCL386 [files] [options]

The square brackets [ ] denote items which are optional.

WCL is the name of the Open Watcom Compile and Link utility that invokes the 16-bit compiler.

WCL386 is the name of the Open Watcom Compile and Link utility that invokes the 32-bit compiler.

The files and options may be specified in any order. The Open Watcom Compile and Link utility uses the extension of the file name to determine if it is a source file, an object file, or a library file. Files with extensions of "OBJ" and "LIB" are assumed to be object files and library files respectively. Files with extensions of "ASM" are assumed to be assembler source files and will be assembled by the Open Watcom Assembler. Files with any other extension, including none at all, are assumed to be C/C++ source files and will be compiled. Pattern matching characters ("*" and "?") may be used in the file specifications.

If no file extension is specified for a file name then the Open Watcom Compile and Link utility will check for a file with one of the following extensions.

```
Order   Name.Ext        Assumed to be
-----   --------        ---------------
  1.    file.ASM        Assembler source code
  2.    file.CXX        C++ source code
  3.    file.CPP        C++ source code
  4.    file.CC         C++ source code
  5.    file.C          C   source code
```

It checks for each file in the order listed. By default, the Open Watcom Assembler will be selected to compile files with the extension "ASM". By default, the Open Watcom C++ compiler will be selected to compile files with any of the extensions "CXX", "CPP" or "CC". By default, the Open Watcom C compiler will be selected to compile a file with a "C" extension. The default selection of compiler can be overridden by the "cc" and "cc++" options, described below.

Options are prefixed with a slash (/) or a dash (–) and may be specified in any order. Options can include any of the Open Watcom C/C++ compiler options plus some additional options specific to the Open Watcom Compile and Link utility. A summary of options is displayed on the screen by simply entering the "WCL" or "WCL386" command with no arguments.

1.2 Open Watcom Compile and Link Options Summary

General options: Description:

c	compile the files only, do not link them
cc	treat source files as C code
cc++	treat source files as C++ code
y	ignore the WCL/WCL386 environment variable

Compiler options: Description:

0	(16-bit only) 8088 and 8086 instructions (default for 16-bit)
1	(16-bit only) 188 and 186 instructions
2	(16-bit only) 286 instructions
3	(16-bit only) 386 instructions
4	(16-bit only) 486 instructions
5	(16-bit only) Pentium instructions
6	(16-bit only) Pentium Pro instructions
3r	(32-bit only) generate 386 instructions based on 386 instruction timings and use register-based argument passing conventions
3s	(32-bit only) generate 386 instructions based on 386 instruction timings and use stack-based argument passing conventions
4r	(32-bit only) generate 386 instructions based on 486 instruction timings and use register-based argument passing conventions
4s	(32-bit only) generate 386 instructions based on 486 instruction timings and use stack-based argument passing conventions
5r	(32-bit only) generate 386 instructions based on Intel Pentium instruction timings and use register-based argument passing conventions (default for 32-bit)
5s	(32-bit only) generate 386 instructions based on Intel Pentium instruction timings and use stack-based argument passing conventions
6r	(32-bit only) generate 386 instructions based on Intel Pentium Pro instruction timings and use register-based argument passing conventions
6s	(32-bit only) generate 386 instructions based on Intel Pentium Pro instruction timings and use stack-based argument passing conventions

ad[=<file_name>] generate make style automatic dependency file
adbs force path separators generated in auto-dependency files to backslashes
add[=<file_name>] specify source dependency name generated in make style auto-dependency file
adhp[=<file_name>] specify path to use for headers with no path given
adfs force path separators generated in auto-dependency files to forward slashes
adt[=<target_name>] specify target name generated in make style auto-dependency file

bc	build target is a console application
bd	build target is a Dynamic Link Library (DLL)
bg	build target is a GUI application
bm	build target is a multi-thread environment
br	build target uses DLL version of C/C++ run-time libraries

bt[=<os>]	build target for operating system <os>
bw	build target uses default windowing support
d0	(C++ only) no debugging information
d1	line number debugging information
d1+	(C only) line number debugging information plus typing information for global symbols and local structs and arrays
d2	full symbolic debugging information
d2i	(C++ only) d2 and debug inlines; emit inlines as external out-of-line functions
d2s	(C++ only) d2 and debug inlines; emit inlines as static out-of-line functions
d2t	(C++ only) full symbolic debugging information, without type names
d3	full symbolic debugging with unreferenced type names ,*
d3i	(C++ only) d3 plus debug inlines; emit inlines as external out-of-line functions
d3s	(C++ only) d3 plus debug inlines; emit inlines as static out-of-line functions
d<name>[=text]	preprocessor #define name [text]
d+	allow extended -d macro definitions
db	generate browsing information
e<number>	set error limit number (default is 20)
ecc	set default calling convention to __cdecl
ecd	set default calling convention to __stdcall
ecf	set default calling convention to __fastcall
ecp	set default calling convention to __pascal
ecr	set default calling convention to __fortran
ecs	set default calling convention to __syscall
ecw	set default calling convention to __watcall (default)
ee	call epilogue hook routine
ef	use full path names in error messages
ei	force enum base type to use at least an int
em	force enum base type to use minimum
en	emit routine name before prologue
ep[<number>]	call prologue hook routine with number of stack bytes available
eq	do not display error messages (they are still written to a file)
er	(C++ only) do not recover from undefined symbol errors
et	Pentium profiling
ew	(C++ only) generate less verbose messages
ez	(32-bit only) generate Phar Lap Easy OMF-386 object file
fc=<file_name>	(C++ only) specify file of command lines to be batch processed
fh[q][=<file_name>]	use precompiled headers
fhd	store debug info for pre-compiled header once (DWARF only)
fhr	(C++ only) force compiler to read pre-compiled header
fhw	(C++ only) force compiler to write pre-compiled header
fhwe	(C++ only) don’t include pre-compiled header warnings when "we" is used
fi=<file_name>	force file_name to be included
fo=<file_name>	set object or preprocessor output file specification
fpc	generate calls to floating-point library
fpi	(16-bit only) generate in-line 80x87 instructions with emulation (default)
	(32-bit only) generate in-line 387 instructions with emulation (default)
fpi87	(16-bit only) generate in-line 80x87 instructions
	(32-bit only) generate in-line 387 instructions
fp2	generate in-line 80x87 instructions
fp3	generate in-line 387 instructions

fp5	generate in-line 80x87 instructions optimized for Pentium processor
fp6	generate in-line 80x87 instructions optimized for Pentium Pro processor
fpd	enable generation of Pentium FDIV bug check code
fpr	generate 8087 code compatible with older versions of compiler
fr=<file_name>	set error file specification
ft	try truncated (8.3) header file specification
fti	(C only) track include file opens
fx	do not try truncated (8.3) header file specification
fzh	(C++ only) do not automatically append extensions for include files
fzs	(C++ only) do not automatically append extensions for source files
g=<codegroup>	set code group name
h{w,d,c}	set debug output format (Open Watcom, Dwarf, Codeview)
i=<directory>	add directory to list of include directories
j	change char default from unsigned to signed
k	(C++ only) continue processing files (ignore errors)
m{f,s,m,c,l,h}	memory model — mf=flat ms=small mm=medium mc=compact ml=large mh=huge (default is "ms" for 16-bit and Netware, "mf" for 32-bit)
nc=<name>	set name of the code class
nd=<name>	set name of the "data" segment
nm=<name>	set module name different from filename
nt=<name>	set name of the "text" segment
o{a,b,c,d,e,f,f+,h,i,i+,k,l,l+,m,n,o,p,r,s,t,u,x,z}	control optimization
pil	preprocessor ignores #line directives
p{e,l,c,w=<num>}	preprocess file only, sending output to standard output; "c" include comments; "e" encrypt identifiers (C++ only); "l" include #line directives; w=<num> wrap output lines at <num> columns (zero means no wrap)
q	operate quietly
r	save/restore segment registers
ri	return chars and shorts as ints
s	remove stack overflow checks
sg	generate calls to grow the stack
st	touch stack through SS first
t=<num>	(C++ only) set tab stop multiplier
u<name>	preprocessor #undef name
v	output function declarations to .def file (with typedef names)
vc...	(C++ only) VC++ compatibility options
w<number>	set warning level number (default is w1)
wcd=<num>	warning control: disable warning message <num>
wce=<num>	warning control: enable warning message <num>
we	treat all warnings as errors
wo	(C only) (16-bit only) warn about problems with overlaid code
wx	set warning level to maximum setting
x	preprocessor ignores environment variables
xd	(C++ only) disable exception handling (default)
xdt	(C++ only) disable exception handling (same as "xd")
xds	(C++ only) disable exception handling (table-driven destructors)
xr	(C++ only) enable RTTI
xs	(C++ only) enable exception handling
xst	(C++ only) enable exception handling (direct calls for destruction)
xss	(C++ only) enable exception handling (table-driven destructors)
xx	ignore default directories for file search (.,../h,../c,...)
z{a,e}	disable/enable language extensions (default is ze)

zam	disable all predefined old extension macros (keyword macros, non-ISO names)
zat	(C++ only) disable alternative tokens
zc	place literal strings in code segment
zd{f,p}	allow DS register to "float" or "peg" it to DGROUP (default is zdp)
zdl	(32-bit only) load DS register directly from DGROUP
zev	(C only, Unix extension) enable arithmetic on void derived types
zf	(C++ only) let scope of for loop initialization extend beyond loop
zf{f,p}	allow FS register to be used (default for all but flat memory model) or not be used (default for flat memory model)
zfw	generate FWAIT instructions on 386 and later
zg	output function declarations to .def (without typedef names)
zg{f,p}	allow GS register to be used or not used
zk0	double-byte char support for Kanji
zk0u	translate Kanji double-byte characters to UNICODE
zk1	double-byte char support for Chinese/Taiwanese
zk2	double-byte char support for Korean
zkl	double-byte char support if current code page has lead bytes
zku=<codepage>	load UNICODE translate table for specified code page
zl	suppress generation of library file names and references in object file
zld	suppress generation of file dependency information in object file
zlf	add default library information to object files
zls	remove automatically inserted symbols (such as runtime library references)
zm	place each function in separate segment (near functions not allowed)
zmf	place each function in separate segment (near functions allowed)
zp[{1,2,4,8,16}]	set minimal structure packing (member alignment)
zpw	output warning when padding is added in a struct/class
zq	operate quietly
zri	inline floating point rounding code
zro	omit floating point rounding code
zs	syntax check only
zt<number>	set data threshold (default is zt32767)
zu	do not assume that SS contains segment of DGROUP
zv	(C++ only) enable virtual function removal optimization
zw	Microsoft Windows prologue/epilogue code sequences
zW	(16-bit only) Microsoft Windows optimized prologue/epilogue code sequences
zWs	(16-bit only) Microsoft Windows smart callback sequences
zz	remove "@size" from __stdcall function names (10.0 compatible)

See the ***Open Watcom C/C++ User's Guide*** for a full description of compiler options.

Linker options: ***Description:***

bcl=<system name> Compile and link for the specified system name. See the section for link option 'l=' below and the linker user guide for available system names. This is equivalent to specifying -bt=<system name> and -l=<system name>.
k<stack_size> set stack size
fd[=<directive_file>] keep directive file and, optionally, rename it (default name is "__WCL__.LNK").
fe=<executable> name executable file
fm[=<map_file>] generate map file and name it (optional)
lp (16-bit only) create an OS/2 protected-mode program
lr (16-bit only) create a DOS real-mode program
l=<system_name> link a program for the specified system. Among the supported systems are:

286	16-bit DOS executables (synonym for "DOS") under DOS and NT hosted platforms; 16-bit OS/2 executables (synonym for "OS2") under 32-bit OS/2 hosted OS/2 session.
386	32-bit DOS executables (synonym for "DOS4G") under DOS; 32-bit NT character-mode executables (synonym for "NT") under Windows NT; 32-bit OS/2 executables (synonym for "OS2V2") under 32-bit OS/2 hosted OS/2 session.
COM	16-bit DOS "COM" files
DOS	16-bit DOS executables
DOS4G	32-bit Tenberry Software DOS Extender executables
DOS4GNZ	32-bit Tenberry Software DOS Extender non-zero base executables
NETWARE	32-bit Novell NetWare 386 NLMs
NOVELL	32-bit Novell NetWare 386 NLMs (synonym for NETWARE)
NT	32-bit Windows NT character-mode executables
NT_DLL	32-bit Windows NT DLLs
NT_WIN	32-bit Windows NT windowed executables
OS2	16-bit OS/2 V1.x executables
OS2_DLL	16-bit OS/2 DLLs
OS2_PM	16-bit OS/2 PM executables
OS2V2	32-bit OS/2 executables
OS2V2_DLL	32-bit OS/2 DLLs
OS2V2_PM	32-bit OS/2 PM executables
PHARLAP	32-bit PharLap DOS Extender executables
QNX	16-bit QNX executables
QNX386	32-bit QNX executables
TNT	32-bit Phar Lap TNT DOS-style executable
WIN386	32-bit extended Windows 3.x executables/DLLs
WIN95	32-bit Windows 9x executables/DLLs
WINDOWS	16-bit Windows executables
WINDOWS_DLL	16-bit Windows Dynamic Link Libraries
X32R	32-bit FlashTek (register calling convention) executables
X32RV	32-bit FlashTek Virtual Memory (register calling convention) executables
X32S	32-bit FlashTek (stack calling convention) executables
X32SV	32-bit FlashTek Virtual Memory (stack calling convention) executables

These names are among the systems identified in the Open Watcom Linker initialization file, "WLSYSTEM.LNK". The Open Watcom Linker "SYSTEM" directives, found in this file, are used to specify default link options for particular (operating) systems. Users can augment the Open Watcom Linker initialization file with their own system definitions and these may be specified as an argument to the "l=" option. The "system_name" specified in the "l=" option is used to create a "SYSTEM system_name" Open Watcom Linker directive when linking the application.

@<directive_file> include additional directive file
"linker directives" allows use of any linker directive

1.3 WCL/WCL386 Environment Variables

The **WCL** environment variable can be used to specify commonly used ***WCL*** options. The **WCL386** environment variable can be used to specify commonly used ***WCL386*** options. These options are processed before options specified on the command line.

Example:

```
C>set wcl=-d1 -ot

C>set wcl386=-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 the appropriate environment variable has been defined, those options listed become the default each time the ***WCL*** or ***WCL386*** command is used.

The **WCL** environment variable is used by ***WCL*** only. The **WCL386** environment variable is used by ***WCL386*** only. Both ***WCL*** and ***WCL386*** pass the relevant options to the Open Watcom C/C++ compiler and linker. This environment variable is not examined by the Open Watcom C/C++ compiler or the linker when invoked directly.

> ***Hint:*** If you are running DOS and you use the same ***WCL*** or ***WCL386*** options all the time, you may find it handy to place the "SET WCL" or "SET WCL386" command in your DOS system initialization file, `AUTOEXEC.BAT`. If you are running OS/2 and you use the same ***WCL*** or ***WCL386*** options all the time, you may find it handy to place the "SET WCL" or "SET WCL386" command in your OS/2 system initialization file, `CONFIG.SYS`.

1.4 WCL/WCL386 Command Line Examples

For most small applications, the ***WCL*** or ***WCL386*** command will suffice. We have only scratched the surface in describing the capabilities of the ***WCL*** and ***WCL386*** commands. The following examples describe the ***WCL*** and ***WCL386*** commands in more detail.

Suppose that your application is contained in three files called `apdemo.c`, `aputils.c`, and `apdata.c`. We can compile and link all three files with one command.

Example 1:

```
C>wcl -d2 apdemo.c aputils.c apdata.c
C>wcl386 -d2 apdemo.c aputils.c apdata.c
```

The executable program will be stored in `apdemo.exe` since `apdemo` appeared first in the list. Each of the three files is compiled with the "d2" debug option. Debugging information is included in the executable file.

We can issue a simpler command if the current directory contains only our three C/C++ source files.

Example 2:

```
C>wcl -d2 *.c
C>wcl386 -d2 *.c
```

WCL or ***WCL386*** will locate all files with the ".c" filename extension and compile each of them. The name of the executable file will depend on which of the C/C++ source files is found first. Since this is a

somewhat haphazard approach to naming the executable file, ***WCL*** and ***WCL386*** have an option, "fe", which will allow you to specify the name to be used.

Example 3:

```
C>wcl -d2 -fe=apdemo *.c
C>wcl386 -d2 -fe=apdemo *.c
```

By using the "fe" option, the executable file will always be called `apdemo.exe` regardless of the order of the C/C++ source files in the directory.

If the directory contains other C/C++ source files which are not part of the application then other tricks may be used to identify a subset of the files to be compiled and linked.

Example 4:

```
C>wcl -d2 -fe=apdemo ap*.c
C>wcl386 -d2 -fe=apdemo ap*.c
```

Here we compile only those C/C++ source files that begin with the letters "ap".

In our examples, we have recompiled all the source files each time. In general, we will only compile one of them and include the object code for the others.

Example 5:

```
C>wcl -d2 -fe=apdemo aputils.c ap*.obj
C>wcl386 -d2 -fe=apdemo aputils.c ap*.obj
```

The source file `aputils.c` is recompiled and `apdemo.obj` and `apdata.obj` are included when linking the application. The ".obj" filename extension indicates that this file need not be compiled.

Example 6:

```
C>wcl -fe=demo *.c utility.obj
C>wcl386 -fe=demo *.c utility.obj
```

All of the C/C++ source files in the current directory are compiled and then linked with `utility.obj` to generate `demo.exe`.

Example 7:

```
C>set wcl=-mm -d1 -ox -k4096
C>wcl -fe=grdemo gr*.c graph.lib -fd=grdemo

C>set wcl386=-d1 -ox -k4096
C>wcl386 -fe=grdemo gr*.c graph.lib -fd=grdemo
```

All C/C++ source files beginning with the letters "gr" are compiled and then linked with `graph.lib` to generate `grdemo.exe` which uses a 4K stack. The temporary linker directive file that is created by ***WCL*** or ***WCL386*** will be kept and renamed to `grdemo.lnk`.

Example 8:

```
C>set libos2=c:\watcom\lib286\os2;c:\os2
C>set lib=c:\watcom\lib286\dos;c:\watcom\lib286
C>set wcl=-mm -lp
C>wcl grdemo1 \watcom\lib286\os2\graphp.obj phapi.lib
```

The file `grdemo1` is compiled for the medium memory model and then linked with `graphp.obj` and `phapi.lib` to generate `grdemo1.exe` which is to be used with Phar Lap's 286 DOS Extender. The "lp" option indicates that an OS/2 format executable is to be created. The file `graphp.obj` in the directory "\WATCOM\LIB286\OS2" contains special initialization code for Phar Lap's 286 DOS Extender. The file `phapi.lib` is part of the Phar Lap 286 DOS Extender package. The **LIBOS2** environment variable must include the location of the OS/2 libraries and the **LIB** environment variable must include the location of the DOS libraries (in order to locate `graph.lib`). The **LIBOS2** environment variable must also include the location of the OS/2 file `doscalls.lib` which is usually "C:\OS2".

For more complex applications, you should use the "Make" utility.

2 The Open Watcom C/C++ POSIX-like Compiler Driver

The Open Watcom C/C++ POSIX-like Compiler Driver is designed for generating applications, simply and quickly, using a single command line. On the command line, you can list source file names as well as object file names. Source files are either compiled or assembled based on file extension; object files and libraries are simply included in the link phase. Options can be passed on to both the compiler and linker.

2.1 owcc Command Line Format

The format of the command line is:

owcc [options] [files]

The square brackets [ ] denote items which are optional.

The files and options may be specified in any order. The owcc utility uses the extension of the file name to determine if it is a source file, an object file, or a library file. Files with extensions of "o" and "lib" are assumed to be object files and library files respectively. Files with extensions of "asm" are assumed to be assembler source files and will be assembled by the Open Watcom Assembler. Files with any other extension, including none at all, are assumed to be C/C++ source files and will be compiled. Pattern matching characters ("*" and "?") may be used in the file specifications.

If no file extension is specified for a file name then the owcc utility will check for a file with one of the following extensions.

Order	Name.Ext	Assumed to be
1.	file.asm	Assembler source code
2.	file.cxx	C++ source code
3.	file.cpp	C++ source code
4.	file.cc	C++ source code
5.	file.c	C source code

It checks for each file in the order listed. By default, the Open Watcom Assembler will be selected to compile files with the extension "asm". By default, the Open Watcom C++ compiler will be selected to compile files with any of the extensions "cxx", "cpp" or "cc". By default, the Open Watcom C compiler will be selected to compile a file with a "c" extension. The default selection of compiler can be overridden by the "-x" option, described below.

Options are prefixed with a dash (–) and may be specified in any order. Option names were chosen to resemble those of the GNU Compiler Collection (a.k.a. GCC). They are translated into Open Watcom C/C++ options, or to directives for the Open Watcom C/C++ wlink utility, accordingly. A summary of options is displayed on the screen by running the compiler driver like this: "owcc -?". If run without any arguments the compiler driver just displays its name and hints towards the "-?" option.

2.2 owcc Options Summary

General options: Description:

c	compile the files only, do not link them
S	compile the source file(s), then run the Open Watcom C/C++ disassembler on the generated object file(s) instead of linking them. Please note that this leaves you with both an object file and an assembly source file. Unix compilers traditionally compile by generating asm source and pass that to the assembler, so there, the "-S" option is done by stopping short of assembling the file. Open Watcom C/C++ compiles directly to object files, so we need the disassembler to achieve a similar effect.
x {c,c++}	treat all source files as written in the specified programming language, regardless of filename suffix.
o <filename>	Change the name of the generated file. If only the preprocessor is run, this sends the preprocessed output to a file instead of the standard output stream. If only compilation is done, this allows to change the name of the object file. If compilation and disassembly is done, this changes the name of the assembly source file. If owcc runs the linker, this changes the name of the generated executable or DLL.
v	operate verbosely, displaying the actual command lines used to invoke the compiler and linker, and passing flags to them to operate verbosely, too.
zq	operate quietly (default). This is the opposite of the "-v" option.

Compiler options: Description:

march=i{,1,2,3}86,axp,mips,ppc which CPU architecture instruction set is used
mtune=i{3,4,5,6}86 which x86 CPU type to optimize for

mregparm=1	use register-based argument passing conventions (default)
mregparm=0	use stack-based argument passing conventions
MMD	generate auto depend makefile fragment
MF <file>	change name of makefile style auto depend file. Without this option, the filename is the same as the the base name of the source file, with a suffix of ".d".
MT <target>	specify target name generated in makefile style auto depend different than that of the object file name
mconsole	build target is a console application
shared	build target is a Dynamic Link Library (DLL)
mwindows	build target is a GUI application
mthreads	build target is a multi-thread environment
mrtdll	build target uses DLL version of C/C++ run-time libraries

mdefault-windowing build target uses default windowing support

g0	(C++ only) no debugging information
g1	line number debugging information
g1+	(C only) line number debugging information plus typing information for global symbols and local structs and arrays
g2	full symbolic debugging information
g2i	(C++ only) d2 and debug inlines; emit inlines as external out-of-line functions
g2s	(C++ only) d2 and debug inlines; emit inlines as static out-of-line functions
g2t	(C++ only) full symbolic debugging information, without type names
g3	full symbolic debugging with unreferenced type names ,*
g3i	(C++ only) d3 plus debug inlines; emit inlines as external out-of-line functions
g3s	(C++ only) d3 plus debug inlines; emit inlines as static out-of-line functions

g{watcom,dwarf,codeview} set debug output format (Open Watcom, Dwarf, Codeview)

D<name>[=text] preprocessor #define name [text]
D+ allow extended -D macro definitions
fbrowser generate browsing information
Wstop-after-errors=<number> set error limit number (default is 20)
mabi={cdecl,stdcall,fastcall,pascal,fortran,syscall,watcall} set default calling convention
fhook-epilogue call epilogue hook routine
fmessage-full-path use full path names in error messages
fno-short-enum force enum base type to use at least an int
fshort-enum force enum base type to use minimum
femit-names emit routine name before prologue
fhook-prologue[=<number>] call prologue hook routine with number of stack bytes available
include <file_name> force file_name to be included in front of the source file text
fo=<file_name> set object or preprocessor output file specification
msoft-float generate calls to floating-point library
fpmath=287 generate in-line 80x87 instructions
fpmath=387 generate in-line 387 instructions
fptune=586 generate in-line 80x87 instructions optimized for Pentium processor
fptune=686 generate in-line 80x87 instructions optimized for Pentium Pro processor
fr=<file_name> enable error file creation and specify its name
H (C only) track include file opens
I add directory to the list of include directories
fsigned-char change char default from unsigned to signed
k (C++ only) continue processing files (ignore errors)
mcmodel={f,s,m,c,l,h} select a memory model from these choices:

f	flat
s	small
m	medium
c	compact
l	large
h	huge
t	compile code for the small memory model and then use the Open Watcom Linker to generate a "COM" file

The default is small for 16-bit and Netware, flat for 32-bit targets.
O0 turn off all optimization
O1 enable some optimazion
O2 enable most of the usual optimizations
O3 enable even more optimizations
fno-strict-aliasing relax alias checking
fguess-branch-probability branch prediction
fno-optimize-sibling-calls disable call/ret optimization
finline-functions expand functions inline
finline-limit=num which functions to expand inline
fno-omit-frame-pointer generate traceable stack frames
fno-omit-leaf-frame-pointer generate more stack frames
frerun-optimizer enable repeated optimizations
finline-intrinsics[-max] inline intrinsic functions [-max: more aggressively]
fschedule-prologue control flow entry/exit seq.
floop-optimize perform loop optimizations
funroll-loops perform loop unrolling
finline-math generate inline math functions

funsafe-math-optimizations numerically unstable floating-point
ffloat-store improve floating-point consistency
fschedule-insns re-order instructions to avoid stalls
fkeep-duplicates ensure unique addresses for functions
fignore-line-directives preprocessor ignores #line directives
E preprocess sources, sending output to standard output or filename selected via -o
C include original comments in -E output
P don't include #line directives in -E output
fcpp-wrap=<num> wrap output lines at <num> columns (zero means no wrap)
ftabstop=<num> (C++ only) set tab stop multiplier
fno-stack-check remove stack overflow checks
fgrow-stack generate calls to grow the stack
fstack-probe touch stack through SS first
U <name> preprocessor #undef name
fwrite-def output function declarations to .def file (with typedef names)
w turn off all warnings (same as Wlevel0)
Wall turn on most warnings, but not all (same as Wlevel4)
Wlevel<number> set warning level number (default is w1)
Wextra set warning level to maximum setting
Wno-n<num> warning control: disable warning message <num>
Wn<num> warning control: enable warning message <num>
Werror treat all warnings as errors
Woverlay (C only) warn about problems with overlaid code
frtti (C++ only) enable RTTI
fno-eh (C++ only) disable exception handling (default)
feh (C++ only) enable exception handling
feh-direct (C++ only) enable exception handling (direct calls for destruction)
feh-table (C++ only) enable exception handling (table-driven destructors)
std={c89,c99,ow} select language dialect; c89 is (almost) strictly ANSI/ISO standard C89 only, c99 enables C99 support (may be incomplete), ow enables all Open Watcom C/C++ extensions.
fno-writable-strings place literal strings in code segment
fvoid-ptr-arithmetics (C only, Unix extension) enable arithmetic on void derived types
fwrite-def-without-typedefs output function declarations to .def (without typedef names)
fnostdlib suppress generation of library file names and references in object file
ffunction-sections place each function in separate segment (near functions not allowed)
fpack-struct=[{1,2,4,8,16}] set minimal structure packing (member alignment)
Wpadded output warning when padding is added in a struct/class
finline-fp-rounding inline floating point rounding code
fomit-fp-rounding omit floating point rounding code
fnonconst-initializers allow non-constant initializers
fsyntax-only syntax check only

See the ***Open Watcom C/C++ User's Guide*** for a full description of compiler options.

Linker options: ***Description:***

b <target name> Compile and link for the specified target system name. See the section linker user guide for available system names. The linker will effectively receive a -l=<target name> option. owcc looks up <system name> in a specification table "specs.owc" to find out which of the Open Watcom C utilities to run. One those options will be -bt=<os>, where <os> is the generic target platform name, and usually less specific than the linker <system name>. Among the supported systems are:

286	16-bit DOS executables (synonym for "DOS") under DOS and NT hosted platforms; 16-bit OS/2 executables (synonym for "OS2") under 32-bit OS/2 hosted OS/2 session.
386	32-bit DOS executables (synonym for "DOS4G") under DOS; 32-bit NT character-mode executables (synonym for "NT") under Windows NT; 32-bit OS/2 executables (synonym for "OS2V2") under 32-bit OS/2 hosted OS/2 session.
COM	16-bit DOS "COM" files
DOS	16-bit DOS executables
DOS4G	32-bit Tenberry Software DOS/4G DOS Extender executables
DOS4GNZ	32-bit Tenberry Software DOS/4G DOS Extender non-zero base executables
NETWARE	32-bit Novell NetWare 386 NLMs
NOVELL	32-bit Novell NetWare 386 NLMs (synonym for NETWARE)
NT	32-bit Windows NT character-mode executables
NT_DLL	32-bit Windows NT DLLs
NT_WIN	32-bit Windows NT windowed executables
OS2	16-bit OS/2 V1.x executables
OS2_DLL	16-bit OS/2 DLLs
OS2_PM	16-bit OS/2 PM executables
OS2V2	32-bit OS/2 executables
OS2V2_DLL	32-bit OS/2 DLLs
OS2V2_PM	32-bit OS/2 PM executables
PHARLAP	32-bit PharLap DOS Extender executables
QNX	16-bit QNX executables
QNX386	32-bit QNX executables
TNT	32-bit Phar Lap TNT DOS-style executable
WIN386	32-bit extended Windows 3.x executables/DLLs
WIN95	32-bit Windows 9x executables/DLLs
WINDOWS	16-bit Windows executables
WINDOWS_DLL	16-bit Windows Dynamic Link Libraries
X32R	32-bit FlashTek (register calling convention) executables
X32RV	32-bit FlashTek Virtual Memory (register calling convention) executables
X32S	32-bit FlashTek (stack calling convention) executables
X32SV	32-bit FlashTek Virtual Memory (stack calling convention) executables

These names are among the systems identified in the Open Watcom Linker initialization file, "wlsystem.lnk". The Open Watcom Linker "SYSTEM" directives, found in this file, are used to specify default link options for particular (operating) systems. Users can augment the Open Watcom Linker initialization file with their own system definitions and these may be specified as an argument to the "l=" option. The "system_name" specified in the "l=" option is used to create a "SYSTEM system_name" Open Watcom Linker directive when linking the application.

mstack-size=<size> set stack size

fd[=<directive_file>] keep linker directive file generated by this tool and, optionally, rename it (default name is "__owcc__.lnk").

fm[=<map_file>] generate map file, optionally specify its name.

s strip symbolic information not strictly required to run from executable.

Wl,"directives" send any supplementary directives directly to the linker

Wl,@<file> include additional linker directives from <file>. This is actually just a special case of -Wl used to pass the linker's @ directive to pull in directives from <file>

2.3 owcc Command Line Examples

For most small applications, the ***owcc*** command will suffice. We have only scratched the surface in describing the capabilities of the ***owcc*** command. The following examples describe the ***owcc*** commands in more detail.

Suppose that your application is contained in three files called `apdemo.c`, `aputils.c`, and `apdata.c`. We can compile and link all three files with one command.

Example 1:

```
C>owcc -g apdemo.c aputils.c apdata.c
```

The executable program will be stored in `a.out`. Each of the three files is compiled with the "g" debug option. Debugging information is included in the executable file.

We can issue a simpler command if the current directory contains only our three C/C++ source files.

Example 2:

```
C>owcc -g *.c
```

owcc will locate all files with the ".c" filename extension and compile each of them. The default name of the executable file will be `a.out`. Since it is only possible to have one executable with the name `a.out` in a directory, ***owcc*** has an option, "o", which will allow you to specify the name to be used.

Example 3:

```
C>owcc -g -o apdemo *.c
```

By using the "o" option, the executable file will always be called `apdemo`.

If the directory contains other C/C++ source files which are not part of the application then other tricks may be used to identify a subset of the files to be compiled and linked.

Example 4:

```
C>owcc -g -o apdemo ap*.c
```

Here we compile only those C/C++ source files that begin with the letters "ap".

In our examples, we have recompiled all the source files each time. In general, we will only compile one of them and include the object code for the others.

Example 5:

```
C>owcc -g -o apdemo aputils.c ap*.obj
```

The source file `aputils.c` is recompiled and `apdemo.obj` and `apdata.obj` are included when linking the application. The ".obj" filename extension indicates that this file need not be compiled.

Example 6:

```
C>owcc -o demo *.c utility.obj
```

All of the C/C++ source files in the current directory are compiled and then linked with `utility.obj` to generate `demo`. The temporary linker directive file that is created by ***owcc*** will be kept and renamed to `grdemo.lnk`.

For more complex applications, you should use a "Make" utility.

The Open Watcom Assembler

3 The Open Watcom Assembler

3.1 Introduction

This chapter describes the Open Watcom Assembler. It takes as input an assembler source file (a file with extension ".asm") and produces, as output, an object file.

The Open Watcom Assembler command line syntax is the following.

```
WASM [options] [d:][path]filename[.ext] [options] [@env_var]
```

The square brackets [ ] denote items which are optional.

WASM is the name of the Open Watcom Assembler.

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\ASM\". If not specified, the current directory is assumed.

filename is the file name of the assembler source file to be assembled.

ext is the file extension of the assembler source file to be assembled. If omitted, a file extension of ".asm" 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 options, each preceded by a slash ("/") or a dash ("−"). Options may be specified in any order.

The options supported by the Open Watcom Assembler are:

{0,1,2,3,4,5,6}{p}{r,s}

0	same as ".8086"
1	same as ".186"
2{p}	same as ".286" or ".286p"
3{p}	same as ".386" or ".386p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
4{p}	same as ".486" or ".486p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
5{p}	same as ".586" or ".586p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")
6{p}	same as ".686" or ".686p" (also defines "__386__" and changes the default USE attribute of segments from "USE16" to "USE32")

p	protect mode
add r	defines "__REGISTER__"
add s	defines "__STACK__"

Example:

```
-2      -3p      -4pr     -5p
```

bt=<os> defines "__<os>__" and checks the "<os>_INCLUDE" environment variable for include files

c do not output OMF COMENT records that allow WDISASM to figure out when data bytes have been placed in a code segment

d<name>[=text] define text macro

d1 line number debugging support

e stop reading assembler source file at END directive. Normally, anything following the END directive will cause an error.

e<number> set error limit number

fe=<file_name> set error file name

fo=<file_name> set object file name

fi=<file_name> force <file_name> to be included

fpc same as ".no87"

fpi inline 80x87 instructions with emulation

fpi87 inline 80x87 instructions

fp0 same as ".8087"

fp2 same as ".287" or ".287p"

fp3 same as ".387" or ".387p"

fp5 same as ".587" or ".587p"

fp6 same as ".687" or ".687p"

i=<directory> add directory to list of include directories

j or s force signed types to be used for signed values

m{t,s,m,c,l,h,f} memory model: (Tiny, Small, Medium, Compact, Large, Huge, Flat)

-mt	Same as ".model tiny"
-ms	Same as ".model small"
-mm	Same as ".model medium"
-mc	Same as ".model compact"
-ml	Same as ".model large"
-mh	Same as ".model huge"
-mf	Same as ".model flat"

Each of the model directives also defines "__<model>__" (e.g., ".model small" defines "__SMALL__"). They also affect whether something like "foo proc" is considered a "far" or "near" procedure.

nd=<name> set data segment name

nm=<name> set module name

nt=<name> set name of text segment

o allow C form of octal constants

zcm=<mode> set compatibility mode - watcom, masm or tasm, if <mode> is not specified then masm is used, default mode is watcom

zld remove file dependency information

zq or q operate quietly

zz remove "@size" from STDCALL function names

zzo don't mangle STDCALL symbols (WASM backward compatible)

? or h print this message

w<number>	set warning level number
we	treat all warnings as errors
wx	set warning level to maximum setting

3.2 Assembler Directives, Operators and Assembly Opcodes

It is not the intention of this chapter to describe assembly-language programming in any detail. You should consult a book that deals with this topic. However, we present an alphabetically ordered list of the directives, opcodes and register names that are recognized by the assembler.

```
.186            .286            .286c           .286p
.287            .386            .386p           .387
.486            .486p           .586            .586p
.686            .686p           .8086           .8087
addr            alias           align           .alpha
and             assume          .break          byte
casemap         catstr          .code           comm
comment         .const          .continue       .cref
.data           .data?          db              dd
df              .dosseg         dosseg          dp
dq              dt              dup             dw
dword           echo            .else           else
elseif          end             .endif          endif
endm            endp            ends            .endw
eq              equ             equ2            .err
.errb           .errdef         .errdif         .errdifi
.erre           .erridn         .erridni        .errnb
.errndef        .errnz          even            .exit
exitm           extern          externdef       extrn
far             .fardata        .fardata?       for
forc            fword           ge              global
group           gt              high            highword
.if             if              if1             if2
ifb             ifdef           ifdif           ifdifi
ife             ifidn           ifidni          ifnb
ifndef          include         includelib      invoke
irp             irpc            .k3d            label
le              length          lengthof        .lfcond
.list           .listall        .listif         .listmacro
.listmacroall   local           low             lowword
lroffset        lt              macro           mask
.mmx            mod             .model          name
ne              near            .no87           .nocref
.nolist         offset          opattr          option
org             oword           page            popcontext
proc            proto           ptr             public
purge           pushcontext     pword           qword
.radix          record          .repeat         repeat
.sall           sbyte           sdword          seg
segment         .seq            .sfcond         size
sizeof          .stack          .startup        struc
struct          subtitle        subttl          sword
tbyte           textequ         .tfcond         this
title           typedef         union           .until
uses            .while          width           word
```

```
.xcref          .xlist          .xmm            .xmm2
.xmm3

aaa             aad             aam             aas
adc             add             addpd           addps
addsd           addss           addsubpd        addsubps
and             andnpd          andnps          andpd
andps           arpl            bound           bp
bsf             bsr             bswap           bt
btc             btr             bts             call
callf           cbw             cdq             clc
cld             clflush         cli             clts
cmc             cmova           cmovae          cmovb
cmovbe          cmovc           cmove           cmovg
cmovge          cmovl           cmovle          cmovna
cmovnae         cmovnb          cmovnbe         cmovnc
cmovne          cmovng          cmovnge         cmovnl
cmovnle         cmovno          cmovnp          cmovns
cmovnz          cmovo           cmovp           cmovpe
cmovpo          cmovs           cmovz           cmp
cmpeqpd         cmpeqps         cmpeqsd         cmpeqss
cmplepd         cmpleps         cmplesd         cmpless
cmpltpd         cmpltps         cmpltsd         cmpltss
cmpneqpd        cmpneqps        cmpneqsd        cmpneqss
cmpnlepd        cmpnleps        cmpnlesd        cmpnless
cmpnltpd        cmpnltps        cmpnltsd        cmpnltss
cmpordpd        cmpordps        cmpordsd        cmpordss
cmppd           cmpps           cmps            cmpsb
cmpsd           cmpss           cmpsw           cmpunordpd
cmpunordps      cmpunordsd      cmpunordss      cmpxchg
cmpxchg8b       comisd          comiss          cpuid
cvtdq2pd        cvtdq2ps        cvtpd2dq        cvtpd2pi
cvtpd2ps        cvtpi2pd        cvtpi2ps        cvtps2dq
cvtps2pd        cvtps2pi        cvtsd2si        cvtsd2ss
cvtsi2sd        cvtsi2ss        cvtss2sd        cvtss2si
cvttpd2dq       cvttpd2pi       cvttps2dq       cvttps2pi
cvttsd2si       cvttss2si       cwd             cwde
daa             das             dec             div
divpd           divps           divsd           divss
emms            enter           f2xm1           fabs
fadd            faddp           fbld            fbstp
fchs            fclex           fcmovb          fcmovbe
fcmove          fcmovnb         fcmovnbe        fcmovne
fcmovnu         fcmovu          fcom            fcomi
fcomip          fcomp           fcompp          fcos
fdecstp         fdisi           fdiv            fdivp
fdivr           fdivrp          femms           feni
ffree           fiadd           ficom           ficomp
fidiv           fidivr          fild            fimul
fincstp         finit           fist            fistp
fisttp          fisub           fisubr          flat
fld             fld1            fldcw           fldenv
fldenvd         fldenvw         fldl2e          fldl2t
fldlg2          fldln2          fldpi           fldz
fmul            fmulp           fnclex          fndisi
fneni           fninit          fnop            fnrstor
fnrstord        fnrstorw        fnsave          fnsaved
fnsavew         fnstcw          fnstenv         fnstenvd
fnstenvw        fnstsw          fpatan          fprem
```

```
fprem1          fptan           frndint         frstor
frstord         frstorw         fsave           fsaved
fsavew          fscale          fsetpm          fsin
fsincos         fsqrt           fst             fstcw
fstenv          fstenvd         fstenvw         fstp
fstsw           fsub            fsubp           fsubr
fsubrp          ftst            fucom           fucomi
fucomip         fucomp          fucompp         fwait
fxam            fxch            fxrstor         fxsave
fxtract         fyl2x           fyl2xp1         haddpd
haddps          hlt             hsubpd          hsubps
idiv            imul            in              inc
ins             insb            insd            insw
int             into            invd            invlpg
iret            iretd           iretdf          iretf
ja              jae             jb              jbe
jc              jcxz            je              jecxz
jg              jge             jl              jle
jmp             jmpf            jna             jnae
jnb             jnbe            jnc             jne
jng             jnge            jnl             jnle
jno             jnp             jns             jnz
jo              jp              jpe             jpo
js              jz              lahf            lar
lddqu           ldmxcsr         lds             lea
leave           les             lfence          lfs
lgdt            lgs             lidt            lldt
lmsw            lock            lods            lodsb
lodsd           lodsw           loop            loopd
loope           looped          loopew          loopne
loopned         loopnew         loopnz          loopnzd
loopnzw         loopw           loopz           loopzd
loopzw          lsl             lss             ltr
maskmovdqu      maskmovq        maxpd           maxps
maxsd           maxss           mfence          minpd
minps           minsd           minss           monitor
mov             movapd          movaps          movd
movddup         movdq2q         movdqa          movdqu
movhlps         movhpd          movhps          movlhps
movlpd          movlps          movmskpd        movmskps
movntdq         movnti          movntpd         movntps
movntq          movq            movq2dq         movs
movsb           movsd           movshdup        movsldup
movss           movsw           movsx           movupd
movups          movzx           mul             mulpd
mulps           mulsd           mulss           mwait
near            neg             nop             not
or              orpd            orps            out
outs            outsb           outsd           outsw
packssdw        packsswb        packuswb        paddb
paddd           paddq           paddsb          paddsw
paddusb         paddusw         paddw           pand
pandn           pause           pavgb           pavgusb
pavgw           pcmpeqb         pcmpeqd         pcmpeqw
pcmpgtb         pcmpgtd         pcmpgtw         pextrw
pf2id           pf2iw           pfacc           pfadd
pfcmpeq         pfcmpge         pfcmpgt         pfmax
pfmin           pfmul           pfnacc          pfpnacc
pfrcp           pfrcpit1        pfrcpit2        pfrsqit1
```

```
pfrsqrt         pfsub           pfsubr          pi2fd
pi2fw           pinsrw          pmaddwd         pmaxsw
pmaxub          pminsw          pminub          pmovmskb
pmulhrw         pmulhuw         pmulhw          pmullw
pmuludq         pop             popa            popad
popf            popfd           por             prefetch
prefetchnta     prefetcht0      prefetcht1      prefetcht2
prefetchw       psadbw          pshufd          pshufhw
pshuflw         pshufw          pslld           pslldq
psllq           psllw           psrad           psraw
psrld           psrldq          psrlq           psrlw
psubb           psubd           psubq           psubsb
psubsw          psubusb         psubusw         psubw
pswapd          punpckhbw       punpckhdq       punpckhqdq
punpckhwd       punpcklbw       punpckldq       punpcklqdq
punpcklwd       push            pusha           pushad
pushd           pushf           pushfd          pushw
pxor            rcl             rcpps           rcpss
rcr             rdmsr           rdpmc           rdtsc
rep             repe            repne           repnz
rept            repz            ret             retd
retf            retfd           retn            rol
ror             rsm             rsqrtps         rsqrtss
sahf            sal             sar             sbb
scas            scasb           scasd           scasw
seta            setae           setb            setbe
setc            sete            setg            setge
setl            setle           setna           setnae
setnb           setnbe          setnc           setne
setng           setnge          setnl           setnle
setno           setnp           setns           setnz
seto            setp            setpe           setpo
sets            setz            sfence          sgdt
shl             shld            short           shr
shrd            shufpd          shufps          sidt
sldt            smsw            sp              sqrtpd
sqrtps          sqrtsd          sqrtss          stc
std             sti             stmxcsr         stos
stosb           stosd           stosw           str
sub             subpd           subps           subsd
subss           sysenter        sysexit         test
ucomisd         ucomiss         unpckhpd        unpckhps
unpcklpd        unpcklps        verr            verw
wait            wbinvd          wrmsr           xadd
xchg            xlat            xlatb           xor
xorpd           xorps

ah              al              ax              bh
bl              bx              ch              cl
cr0             cr2             cr3             cr4
cs              cx              dh              di
dl              dr0             dr1             dr2
dr3             dr6             dr7             ds
dx              eax             ebp             ebx
ecx             edi             edx             es
esi             esp             fs              gs
mm0             mm1             mm2             mm3
mm4             mm5             mm6             mm7
si              ss              st              st0
```

```
st1             st2             st3             st4
st5             st6             st7             tr3
tr4             tr5             tr6             tr7
xmm0            xmm1            xmm2            xmm3
xmm4            xmm5            xmm6            xmm7
```

3.3 Unsupported Directives

Other assemblers support directives that this assembler does not. The following is a list of directives that are ignored by the Open Watcom Assembler (use of these directives results in a warning message).

```
.alpha          .cref           .lfcond         .list
.listall        .listif         .listmacro      .listmacroall
.nocref         .nolist         page            .sall
.seq            .sfcond         subtitle        subttl
.tfcond         title           .xcref          .xlist
```

The following is a list of directives that are flagged by the Open Watcom Assembler (use of these directives results in an error message).

```
addr            .break          casemap         catstr
.continue       echo            .else           endmacro
.endif          .endw           .exit           high
highword        .if             invoke          low
lowword         lroffset        mask            opattr
option          popcontext      proto           purge
pushcontext     .radix          record          .repeat
.startup        this            typedef         union
.until          .while          width
```

3.4 Open Watcom Assembler Specific

There are a few specific features in Open Watcom Assembler

3.4.1 Naming convention

```
                  Procedure   Variable
Convention          Name        Name
---------------   ----------  ---------
C                   '_*'        '_*'
WATCOM_C          see section Open Watcom "C" name mangler
SYSCALL             '*'         '*'
STDCALL            '_*@nn'     '_*'
STDCALL             '_*'        '_*'     see note 1
STDCALL             '*'         '*'    see note 2
BASIC               '^'         '^'
FORTRAN             '^'         '^'
PASCAL              '^'         '^'
```

Notes:

1. In STDCALL procedures name 'nn' is overall parametrs size in bytes. '@nn' is suppressed when -zz command line option is used (WATCOM 10.0 compatibility).

2. STDCALL symbols mangling is suppressed by -zzo command line option (WASM backward compatible).

3.4.2 Open Watcom "C" name mangler

Command line option	Procedure Name	Variable Name
0,1,2	'*_'	'_*'
3,4,5,6 with r	'*_'	'_*'
3,4,5,6 with s	'*'	'*'

3.4.3 Calling convention

Convention	Vararg	Parameters passed by	Parameters order	Cleanup caller stack
C	yes	stack	right to left	no
WATCOM_C	yes	registers	right to left	see note 1
	yes	stack	right to left	no
SYSCALL	yes	stack	right to left	no
STDCALL	yes	stack	right to left	yes see note 2
BASIC	no	stack	left to right	yes
FORTRAN	no	stack	left to right	yes
PASCAL	no	stack	left to right	yes

Notes:

1. If any parameter is passed on the stack then WASM automaticaly cleanup caller stack.

2. For STDCALL procedures WASM automaticaly cleanup caller stack, except case when vararg parameter is used.

3.5 Open Watcom Assembler Diagnostic Messages

1 Size doesn't match with previous definition

2 Invalid instruction with current CPU setting

3 LOCK prefix is not allowed on this instruction

4 REP prefix is not allowed on this instruction

5 Invalid memory pointer

6 Cannot use 386 addressing mode with current CPU setting

7 Too many base registers

8 Invalid index register

9 Scale factor must be 1, 2, 4 or 8

10 invalid addressing mode with current CPU setting

11 ESP cannot be used as index

12 Too many base/index registers

13 Memory offset cannot reference to more than one label

14 Offset must be relocatable

15 Memory offset expected

16 Invalid indirect memory operand

17 Cannot mix 16 and 32-bit registers

18 CPU type already set

19 Unknown directive

20 Expecting comma

21 Expecting number

22 Invalid label definition

23 Invalid use of SHORT, NEAR, FAR operator

24 No memory

25 Cannot use 386 segment register with current CPU setting

26 POP CS is not allowed

27 Cannot use 386 register with current CPU setting

28 Only MOV can use special register

29 Cannot use TR3, TR4, TR5 in current CPU setting

30 Cannot use SHORT with CALL

31 Only SHORT displacement is allowed

32 Syntax error

33 Prefix must be followed by an instruction

34 No size given before 'PTR' operator

35 Invalid IMUL format

36 Invalid SHLD/SHRD format

37 Too many commas

38 Syntax error: Unexpected colon

39 Operands must be the same size

40 Invalid instruction operands

41 Immediate constant too large

42 Can not use short or near modifiers with this instruction

43 Jump out of range

44 Displacement cannot be larger than 32k

45 Initializer value too large

46 Symbol already defined

47 Immediate data too large

48 Immediate data out of range

49 Can not transfer control to stack symbol

50 Offset cannot be smaller than WORD size

51 Can not take offset of stack symbol

52 Can not take segment of stack symbol

53 Segment too large

54 Offset cannot be larger than 32k

55 Operand 2 too big

56 Operand 1 too small

57 Too many arithmetic operators

58 Too many open square brackets

59 Too many close square brackets

60 Too many open brackets

61 Too many close brackets

62 Invalid number digit

63 Assembler Code is too long

64 Brackets are not balanced

65 Operator is expected

66 Operand is expected

67 Too many tokens in a line

68 Bracket is expected

69 Illegal use of register

70 Illegal use of label

71 Invalid operand in addition

72 Invalid operand in subtraction

73 One operand must be constant

74 Constant operand is expected

75 A constant operand is expected in addition

76 A constant operand is expected in subtraction

77 A constant operand is expected in multiplication

78 A constant operand is expected in division

79 A constant operand is expected after a positive sign

80 A constant operand is expected after a negative sign

81 Label is not defined

82 More than one override

83 Label is expected

84 Only segment or group label is allowed

85 Only register or label is expected in override

86 Unexpected end of file

87 Label is too long

88 This feature has not been implemented yet

89 Internal Error #1

90 Can not take offset of group

91 Can not take offset of segment

92 Invalid character found

93 Invalid operand size for instruction

94 This instruction is not supported

95 size not specified -- BYTE PTR is assumed

96 size not specified -- WORD PTR is assumed

97 size not specified -- DWORD PTR is assumed

500 Segment parameter is defined already

501 Model parameter is defined already

502 Syntax error in segment definition

503 'AT' is not supported in segment definition

504 Segment definition is changed

505 Lname is too long

506 Block nesting error

507 Ends a segment which is not opened

508 Segment option is undefined

509 Model option is undefined

510 No segment is currently opened

511 Lname is used already

512 Segment is not defined

513 Public is not defined

514 Colon is expected

515 A token is expected after colon

516 Invalid qualified type

517 Qualified type is expected

518 External definition different from previous one

519 Memory model is not found in .MODEL

520 Cannot open include file

521 Name is used already

522 Library name is missing

523 Segment name is missing

524 Group name is missing

525 Data emitted with no segment

526 Seglocation is expected

527 Invalid register

528 Cannot address with assumed register

529 Invalid start address

530 Label is already defined

531 Token is too long

532 The line is too long after expansion

533 A label is expected after colon

534 Must be associated with code

535 Procedure must have a name

536 Procedure is alreadly defined

537 Language type must be specified

538 End of procedure is not found

539 Local variable must immediately follow PROC or MACRO statement

540 Extra character found

541 Cannot nest procedures

542 No procedure is currently defined

543 Procedure name does not match

544 Vararg requires C calling convention

545 Model declared already

546 Model is not declared

547 Backquote expected

548 COMMENT delimiter expected

549 End directive required at end of file

550 Nesting level too deep

551 Symbol not defined

552 Insert Stupid warning #1 here

553 Insert Stupid warning #2 here

554 Spaces not allowed in command line options

555 Error:

556 Source File

557 No filename specified.

558 Out of Memory

559 Cannot Open File -

560 Cannot Close File -

561 Cannot Get Start of Source File -

562 Cannot Set to Start of Source File -

563 Command Line Contains More Than 1 File To Assemble

564 include path %s.

565 Unknown option %s. Use /? for list of options.

566 read more command line from %s.

567 Internal error in %s(%u)

568 OBJECT WRITE ERROR !!

569 NO LOR PHARLAP !!

570 Parameter Required

571 Expecting closing square bracket

572 Expecting file name

573 Floating point instruction not allowed with /fpc

574 Too many errors

575 Build target not recognised

576 Public constants should be numeric

577 Expecting symbol

578 Do not mix simplified and full segment definitions

579 Parms passed in multiple registers must be accessed separately, use %s

580 Ten byte variables not supported in register calling convention

581 Parameter type not recognised

582 forced error:

583 forced error: Value not equal to 0 : %d

584 forced error: Value equal to 0: %d

585 forced error: symbol defined: %s

586 forced error: symbol not defined: %s

587 forced error: string blank : <%s>

588 forced error: string not blank : <%s>

589 forced error: strings not equal : <%s> : <%s>

590 forced error: strings equal : <%s> : <%s>

591 included by file %s(%d)

592 macro called from file %s(%d)

593 Symbol %s not defined

594 Extending jump

595 Ignoring inapplicable directive

596 Unknown symbol class '%s'

597 Symbol class for '%s' already established

598 number must be a power of 2

599 alignment request greater than segment alignment

600 '%s' is already defined

601 %u unclosed conditional directive(s) detected

Object File Utilities

4 The Open Watcom Library Manager

4.1 Introduction

The Open Watcom Library Manager can be used to create and update object library files. It takes as input an object file or a library file and creates or updates a library file. For OS/2, Win16 and Win32 applications, it can also create import libraries from Dynamic Link Libraries.

An object library is essentially a collection of object files. These object files generally contain utility routines that can be used as input to the Open Watcom Linker to create an application. The following are some of the advantages of using library files.

1. Only those modules that are referenced will be included in the executable file. This eliminates the need to know which object files should be included and which ones should be left out when linking an application.

2. Libraries are a good way of organizing object files. When linking an application, you need only list one library file instead of several object files.

The Open Watcom Library Manager currently runs under the following operating systems.

- DOS
- OS/2
- QNX
- Windows

4.2 The Open Watcom Library Manager Command Line

The following describes the Open Watcom Library Manager command line.

```
WLIB [options_1] lib_file [options_2] [cmd_list]
```

The square brackets "[]" denote items which are optional.

lib_file is the file specification for the library file to be processed. If no file extension is specified, a file extension of "lib" is assumed.

options_1 is a list of valid options. Options may be specified in any order. If you are using a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager, options are preceded by a "/" or "—" character. If you are using a UNIX-hosted version of the Open Watcom Library Manager, options are preceded by a "—" character.

options_2 is a list of valid options. These options are only permitted if you are running a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager and must be preceded by a "/" character. The "—" character cannot be used as an option delimiter for options following the library file name since it will be interpreted as a delete command.

cmd_list is a list of commands to the Open Watcom Library Manager specifying what operations are to be performed. Each command in *cmd_list* is separated by a space.

The following is a summary of valid options. Items enclosed in square brackets "[]" are optional. Items separated by an or-bar "|" and enclosed in parentheses "()" indicate that one of the items must be specified. Items enclosed in angle brackets "<>" are to be replaced with a user-supplied name or value (the "<>" are not included in what you specify).

?	display the usage message
b	suppress creation of backup file
c	perform case sensitive comparison
d=<output_directory>	directory in which extracted object modules will be placed
fa	output AR format library (host default ar format)
fab	output AR format library (BSD ar format)
fac	output AR format library (COFF ar format)
fag	output AR format library (GNU ar format)
fm	output MLIB format library
fo	output OMF format library
h	display the usage message
ia	generate AXP import records
ii	generate X86 import records
ip	generate PPC import records
ie	generate ELF import records
ic	generate COFF import records
io	generate OMF import records
i(r|n)(n|o)	imports for the resident/non-resident names table are to be imported by name/ordinal.
l[=<list_file>]	create a listing file
m	display C++ mangled names
n	always create a new library
o=<output_file>	set output file name for library
p=<record_size>	set library page size (supported for "OMF" library format only)
pa	set optimal library page size automaticaly (supported for "OMF" library format only)
q	suppress identification banner
s	strip line number records from object files (supported for "OMF" library format only)
t	remove path information from module name specified in THEADR records (supported for "OMF" library format only)
v	do not suppress identification banner
x	extract all object modules from library
zld	strip file dependency info from object files (supported for "OMF" library format only)

The following sections describe the operations that can be performed on a library file. Note that before making a change to a library file, the Open Watcom Library Manager makes a backup copy of the original library file unless the "o" option is used to specify an output library file whose name is different than the original library file, or the "b" option is used to suppress the creation of the backup file. The backup copy has the same file name as the original library file but has a file extension of "bak". Hence, **lib_file** should not have a file extension of "bak".

4.3 Open Watcom Library Manager Module Commands

The following is a summary of basic Open Watcom Library Manager module manipulation commands:

+	add module to a library
-	remove module from a library
**** or :***	extract module from a library (**:** is used with a UNIX-hosted version of the Open Watcom Library Manager, otherwise ***** is used)
++	add import library entry

4.4 Adding Modules to a Library File

An object file can be added to a library file by specifying a **+obj_file** command where **obj_file** is the file specification for an object file. If you are using a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager, a file extension of "obj" is assumed if none is specified. If you are using a UNIX-hosted version of the Open Watcom Library Manager, a file extension of "o" is assumed if none is specified. If the library file does not exist, a warning message will be issued and the library file will be created.

Example:

```
wlib mylib +myobj
```

In the above example, the object file "myobj" is added to the library file "mylib.lib".

When a module is added to a library, the Open Watcom Library Manager will issue a warning if a symbol redefinition occurs. This will occur if a symbol in the module being added is already defined in another module that already exists in the library file. Note that the module will be added to the library in any case.

It is also possible to combine two library files together. The following example adds all modules in the library "newlib.lib" to the library "mylib.lib".

Example:

```
wlib mylib +newlib.lib
```

Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are adding an object file.

4.5 Deleting Modules from a Library File

A module can be deleted from a library file by specifying a **-mod_name** command where **mod_name** is the file name of the object file when it was added to the library with the directory and file extension removed.

Example:

```
wlib mylib -myobj
```

In the above example, the Open Watcom Library Manager is instructed to delete the module "myobj" from the library file "mylib.lib".

It is also possible to specify a library file instead of a module name.

Example:

```
wlib mylib -oldlib.lib
```

In the above example, all modules in the library file "oldlib.lib" are removed from the library file "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are removing an object module.

4.6 Replacing Modules in a Library File

A module can be replaced by specifying a **-+mod_name** or **+-mod_name** command. The module **mod_name** is deleted from the library. The object file "mod_name" is then added to the library.

Example:

```
wlib mylib -+myobj
```

In the above example, the module "myobj" is replaced by the object file "myobj".

It is also possible to merge two library files.

Example:

```
wlib mylib -+updlib.lib
```

In the above example, all modules in the library file "updlib.lib" replace the corresponding modules in the library file "mylib.lib". Any module in the library "updlib.lib" not in library "mylib.lib" is added to the library "mylib.lib". Note that you must specify the "lib" file extension. Otherwise, the Open Watcom Library Manager will assume you are replacing an object module.

4.7 Extracting a Module from a Library File

A module can be extracted from a library file by specifying a ***mod_name** [=file_name] command for a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager or a **:mod_name** [=file_name] command for a UNIX-hosted version of the Open Watcom Library Manager. The module **mod_name** is not deleted but is copied to a disk file. If **mod_name** is preceded by a path specification, the output file will be placed in the directory identified by the path specification. If **mod_name** is followed by a file extension, the output file will contain the specified file extension.

Example:

```
wlib mylib *myobj        DOS, OS/2 or Windows-hosted
     or
wlib mylib :myobj        UNIX-hosted
```

In the above example, the module "myobj" is copied to a disk file. The disk file will be an object file with file name "myobj". If you are running a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager, a file extension of "obj" will be used. If you are running a UNIX-hosted version of the Open Watcom Library Manager, a file extension of "o" will be used.

Example:

```
wlib mylib *myobj.out    DOS, OS/2 or Windows-hosted
     or
wlib mylib :myobj.out    UNIX-hosted
```

In the above example, the module "myobj" will be extracted from the library file "mylib.lib" and placed in the file "myobj.out"

The following form of the extract command can be used if the module name is not the same as the output file name.

Example:

```
wlib mylib *myobj=newmyobj.out    DOS, OS/2 or Windows-hosted
     or
wlib mylib :myobj=newmyobj.out    UNIX-hosted
```

You can extract a module from a file and have that module deleted from the library file by specifying a ***-mod_name** command for a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager or a **:-mod_name** command for a UNIX-hosted version of the Open Watcom Library Manager. The following example performs the same operations as in the previous example but, in addition, the module is deleted from the library file.

Example:

```
wlib mylib *-myobj.out   DOS, OS/2 or Windows-hosted
     or
wlib mylib :-myobj.out   UNIX-hosted
```

Note that the same result is achieved if the delete operator precedes the extract operator.

4.8 Creating Import Libraries

The Open Watcom Library Manager can also be used to create import libraries from Dynamic Link Libraries. Import libraries are used when linking OS/2, Win16 or Win32 applications.

Example:

```
wlib implib +dynamic.dll
```

In the above example, the following actions are performed. For each external symbol in the specified Dynamic Link Library, a special object module is created that identifies the external symbol and the actual name of the Dynamic Link Library it is defined in. This object module is then added to the specified library. The resulting library is called an import library.

Note that you must specify the "dll" file extension. Otherwise, the Open Watcom Library Manager will assume you are adding an object file.

4.9 Creating Import Library Entries

An import library entry can be created and added to a library by specifying a command of the following form.

```
++sym.dll_ name[.[altsym].export_ name][.ordinal]
```

where	***description:***
sym	is the name of a symbol in a Dynamic Link Library.
dll_name	is the name of the Dynamic Link Library that defines `sym`.
altsym	is the name of a symbol in a Dynamic Link Library. When omitted, the default symbol name is `sym`.
export_name	is the name that an application that is linking to the Dynamic Link Library uses to reference `sym`. When omitted, the default export name is `sym`.
ordinal	is the ordinal value that can be used to identify `sym` instead of using the name `export_ name`.

Example:

```
wlib math ++_ _ sin.trig.sin.1
```

In the above example, an import library entry will be created for symbol `sin` and added to the library "math.lib". The symbol `sin` is defined in the Dynamic Link Library called "trig.dll" as `_ _ sin`. When an application is linked with the library "math.lib", the resulting executable file will contain an import by ordinal value 1. If the ordinal value was omitted, the resulting executable file would contain an import by name `sin`.

4.10 Commands from a File or Environment Variable

The Open Watcom Library Manager can be instructed to process all commands in a disk file or environment variable by specifying the **@name** command where **name** is a file specification for the command file or the name of an environment variable. A file extension of "lbc" is assumed for files if none is specified. The commands must be one of those previously described.

Example:

```
wlib mylib @mycmd
```

In the above example, all commands in the environment variable "mycmd" or file "mycmd.lbc" are processed by the Open Watcom Library Manager.

4.11 Open Watcom Library Manager Options

The following sections describe the list of options allowed when invoking the Open Watcom Library Manager.

4.11.1 Suppress Creation of Backup File - "b" Option

The "b" option tells the Open Watcom Library Manager to not create a backup library file. In the following example, the object file identified by "new" will be added to the library file "mylib.lib".

Example:

```
wlib -b mylib +new
```

If the library file "mylib.lib" already exits, no backup library file ("mylib.bak") will be created.

4.11.2 Case Sensitive Symbol Names - "c" Option

The "c" option tells the Open Watcom Library Manager to use a case sensitive compare when comparing a symbol to be added to the library to a symbol already in the library file. This will cause the names "myrtn" and "MYRTN" to be treated as different symbols. By default, comparisons are case insensitive. That is the symbol "myrtn" is the same as the symbol "MYRTN".

4.11.3 Specify Output Directory - "d" Option

The "d" option tells the Open Watcom Library Manager the directory in which all extracted modules are to be placed. The default is to place all extracted modules in the current directory.

In the following example, the module "mymod" is extracted from the library "mylib.lib". If you are running a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager, the module will be placed in the file "\obj\mymod.obj". If you are running a UNIX-hosted version of the Open Watcom Library Manager, the module will be placed in the file "/o/mymod.o".

Example:

```
wlib -d=\obj mymod       DOS, OS/2 or Windows-hosted
    or
wlib -d=/o mymod         UNIX-hosted
```

4.11.4 Specify Output Format - "f" Option

The "f" option tells the Open Watcom Library Manager the format of the output library. The default output format is determined by the type of object files that are added to the library when it is created. The possible output format options are:

fa output AR format library, host OS default ar format

fab output AR format library, BSD ar format

fac output AR format library, COFF ar format

fag output AR format library, GNU ar format

fm output MLIB format library

fo output OMF format library

4.11.5 Generating Imports - "i" Option

The "i" option can be used to describe type of import library to create.

ia generate AXP import records

ii generate X86 import records

ip generate PPC import records

ie generate ELF import records

ic generate COFF import records

io generate OMF import records

When creating import libraries from Dynamic Link Libraries, import entries for the names in the resident and non-resident names tables are created. The "i" option can be used to describe the method used to import these names.

iro Specifying "iro" causes imports for names in the resident names table to be imported by ordinal.

irn Specifying "irn" causes imports for names in the resident names table to be imported by name. This is the default.

ino Specifying "ino" causes imports for names in the non-resident names table to be imported by ordinal. This is the default.

inn Specifying "inn" causes imports for names in the non-resident names table to be imported by name.

Example:

```
wlib -iro -inn implib +dynamic.dll
```

Note that you must specify the "dll" file extension for the Dynamic Link Library. Otherwise an object file will be assumed.

4.11.6 Creating a Listing File - "l" Option

The "l" (lower case "L") option instructs the Open Watcom Library Manager to produce a list of the names of all symbols that can be found in the library file to a listing file. The file name of the listing file is the same as the file name of the library file. The file extension of the listing file is "lst".

Example:

```
wlib -l mylib
```

In the above example, the Open Watcom Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.lst".

An alternate form of this option is `-l=list_ file.` With this form, you can specify the name of the listing file. When specifying a listing file name, a file extension of "lst" is assumed if none is specified.

Example:

```
wlib -l=mylib.out mylib
```

In the above example, the Open Watcom Library Manager is instructed to list the contents of the library file "mylib.lib" and produce the output to a listing file called "mylib.out".

You can get a listing of the contents of a library file to the terminal by specifying only the library name on the command line as demonstrated by the following example.

Example:

```
wlib mylib
```

4.11.7 Display C++ Mangled Names - "m" Option

The "m" option instructs the Open Watcom Library Manager to display C++ mangled names rather than displaying their demangled form. The default is to interpret mangled C++ names and display them in a somewhat more intelligible form.

4.11.8 Always Create a New Library - "n" Option

The "n" option tells the Open Watcom Library Manager to always create a new library file. If the library file already exists, a backup copy is made (unless the "b" option was specified). The original contents of the library are discarded and a new library is created. If the "n" option was not specified, the existing library would be updated.

Example:

```
wlib -n mylib +myobj
```

In the above example, a library file called "mylib.lib" is created. It will contain a single object module, namely "myobj", regardless of the contents of "mylib.lib" prior to issuing the above command. If "mylib.lib" already exists, it will be renamed to "mylib.bak".

4.11.9 Specifying an Output File Name - "o" Option

The "o" option can be used to specify the output library file name if you want the original library to remain unchanged and a new library created.

Example:

```
wlib -o=newlib lib1 +lib2.lib
```

In the above example, the modules from "lib1.lib" and "lib2.lib" are added to the library "newlib.lib". Note that since the original library remains unchanged, no backup copy is created. Also, if the "l" option is used to specify a listing file, the listing file will assume the file name of the output library.

4.11.10 Specifying a Library Record Size - "p" and "pa" Options

The "p" option specifies the record size in bytes for each record in the library file. The record size must be a power of 2 and in the range 16 to 32768. If the record size is less than 16, it will be rounded up to 16. If the record size is greater than 16 and not a power of 2, it will be rounded up to the nearest power of 2. The default record size is 256 bytes.

Each entry in the dictionary of a library file contains an offset from the start of the file which points to a module. The offset is 16 bits and is a multiple of the record size. Since the default record size is 256, the maximum size of a library file for a record size of 256 is 256*64K. If the size of the library file increases beyond this size, you must increase the record size.

Example:

```
wlib -p=512 lib1 +lib2.lib
```

In the above example, the Open Watcom Library Manager is instructed to create/update the library file "lib1.lib" by adding the modules from the library file "lib2.lib". The record size of the resulting library file is 512 bytes.

The "pa" option specifies the record size is determined automaticaly to be minimal in size.

Example:

```
wlib -pa lib1 +lib2.lib
```

In the above example, the Open Watcom Library Manager is instructed to create/update the library file "lib1.lib" by adding the modules from the library file "lib2.lib". The record size of the resulting library file is optimal (minimal) regardles of what each library page size is.

4.11.11 Operate Quietly - "q" Option

The "q" option suppressing the banner and copyright notice that is normally displayed when the Open Watcom Library Manager is invoked.

Example:

```
wlib -q -l mylib
```

4.11.12 Strip Line Number Records - "s" Option

The "s" option tells the Open Watcom Library Manager to remove line number records from object files that are being added to a library. Line number records are generated in the object file if the "d1" option is specified when compiling the source code.

Example:
```
wlib -s mylib +myobj
```

4.11.13 Trim Module Name - "t" Option

The "t" option tells the Open Watcom Library Manager to remove path information from the module name specified in THEADR records in object files that are being added to a library. The module name is created from the file name by the compiler and placed in the THEADR record of the object file. The module name will contain path information if the file name given to the compiler contains path information.

Example:
```
wlib -t mylib +myobj
```

4.11.14 Operate Verbosely - "v" Option

The "v" option enables the display of the banner and copyright notice when the Open Watcom Library Manager is invoked.

Example:
```
wlib -v -l mylib
```

4.11.15 Explode Library File - "x" Option

The "x" option tells the Open Watcom Library Manager to extract all modules from the library. Note that the modules are not deleted from the library. Object modules will be placed in the current directory unless the "d" option is used to specify an alternate directory.

In the following example all modules will be extracted from the library "mylib.lib" and placed in the current directory.

Example:
```
wlib -x mylib
```

In the following example, all modules will be extracted from the library "mylib.lib". If you are running a DOS, OS/2 or Windows-hosted version of the Open Watcom Library Manager, the module will be placed in the "\obj" directory. If you are running a UNIX-hosted version of the Open Watcom Library Manager, the module will be placed in the file "/o" directory.

Example:
```
wlib -x -d=\obj mylib    DOS, OS/2 or Windows-hosted
    or
wlib -x -d=/o mylib      UNIX-hosted
```

4.12 Librarian Error Messages

The following messages may be issued by the Open Watcom Library Manager.

Error! Could not open object file '%s'.
Object file '%s' could not be found. This message is usually issued when an attempt is made to add a non-existent object file to the library.

Error! Could not open library file '%s'.
The specified library file could not be found. This is usually issued for input library files. For example, if you are combining two library files, the library file you are adding is an input library file and the library file you are adding to or creating is an output library file.

Error! Invalid object module in file '%s' not added.
The specified file contains an invalid object module.

Error! Dictionary too large. Recommend split library into two libraries.
The size of the dictionary in a library file cannot exceed 64K. You must split the library file into two separate library files.

Error! Redefinition of module '%s' in file '%s'.
This message is usually issued when an attempt is made to add a module to a library that already contains a module by that name.

Warning! Redefinition of symbol '%s' in file '%s' ignored.
This message is issued if a symbol defined by a module already in the library is also defined by a module being added to the library.

Error! Library too large. Recommend split library into two libraries or try a larger page_bound than %xH.
The record size of the library file does not allow the library file to increase beyond its current size. The record size of the library file must be increased using the "p" option.

Error! Expected '%s' in '%s' but found '%s'.
An error occurred while scanning command input.

Warning! Could not find module '%s' for deletion.
This message is issued if an attempt is made to delete a module that does not exist in the library.

Error! Could not find module '%s' for extraction.
This message is issued if an attempt is made to extract a module that does not exist in the library.

Error! Could not rename old library for backup.
The Open Watcom Library Manager creates a backup copy before making any changes (unless the "b" option is specified). This message is issued if an error occurred while trying to rename the original library file to the backup file name.

Warning! Could not open library '%s' : will be created.
The specified library does not exist. It is usually issued when you are adding to a non-existent library. The Open Watcom Library Manager will create the library.

Warning! Output library name specification ignored.
This message is issued if the library file specified by the "o" option could not be opened.

Warning! Could not open library '%s' and no operations specified: will not be created.
This message is issued if the library file specified on the command line does not exist and

no operations were specified. For example, asking for a listing file of a non-existent library will cause this message to be issued.

Warning! Could not open listing file '%s'.
The listing file could not be opened. For example, this message will be issued when a "disk full" condition is present.

Error! Could not open output library.
The output library could not be opened.

Error! Unable to write to output library.
An error occurred while writing to the output library.

Error! Unable to write to extraction file '%s'.
This message is issued when extracting an object module from a library file and an error occurs while writing to the output file.

Error! Out of Memory.
There was not enough memory to process the library file.

Error! Could not open file '%s'.
This message is issued if the output file for a module that is being extracted from a library could not be opened.

Error! Library '%s' is invalid. Contents ignored.
The library file does not contain the correct header information.

Error! Library '%s' has an invalid page size. Contents ignored.
The library file has an invalid record size. The record size is contained in the library header and must be a power of 2.

Error! Invalid object record found in file '%s'.
The specified file contains an invalid object record.

Error! No library specified on command line.
This message is issued if a library file name is not specified on the command line.

Error! Expecting library name.
This message is issued if the location of the library file name on the command line is incorrect.

Warning! Invalid file name '%s'.
This message is issued if an invalid file name is specified. For example, a file name longer that 127 characters is not allowed.

Error! Could not open command file '%s'.
The specified command file could not be opened.

Error! Could not read from file '%s'. Contents ignored as command input.
An error occurred while reading a command file.

5 The Object File Disassembler

5.1 Introduction

This chapter describes the Open Watcom Disassembler. It takes as input an object file (a file with extension ".obj") and produces, as output, the Intel assembly language equivalent. The Open Watcom compilers do not produce an assembly language listing directly from a source program. Instead, the Open Watcom Disassembler can be used to generate an assembly language listing from the object file generated by the compiler.

The Open Watcom Disassembler command line syntax is the following.

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

The square brackets [ ] denote items which are optional.

WDIS is the name of the Open Watcom Disassembler.

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\OBJ\". If not specified, the current directory is assumed.

filename is the file name of the object file to disassemble.

ext is the file extension of the object file to disassemble. If omitted, a file extension of ".obj" 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 options, each preceded by a slash ("/") or a dash ("−"). Options may be specified in any order.

The options supported by the Open Watcom Disassembler are:

a	write assembly instructions only to the listing file
e	include list of external names
ff	print FPU emulator fixups as comment line
fi	use alternate indexing format [80(x)86 only]
fp	do not use instruction name pseudonyms
fr	do not use register name pseudonyms [Alpha only]
fu	instructions/registers in upper case
i=<char>	redefine the initial character of internal labels (default: L)
l[=<list_file>]	create a listing file

m leave C++ names mangled
p include list of public names
s[=<source_file>]
using object file source line information, imbed original source lines into the output file

The following sections describe the list of options.

5.2 Changing the Internal Label Character - "i=<char>"

The "i" option permits you to specify the first character to be used for internal labels. Internal labels take the form "Ln" where "n" is one or more digits. The default character "L" can be changed using the "i" option. The replacement character must be a letter (a-z, A-Z). A lowercase letter is converted to uppercase.

Example:

```
C>wdis calendar -i=x
```

5.3 The Assembly Format Option - "a"

The "a" option controls the format of the output produced to the listing file. When specified, the Open Watcom Disassembler will produce a listing file that can be used as input to an assembler.

Example:

```
C>wdis calendar -a -l=calendar.asm
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.obj` and produce the output to the file `calendar.asm` so that it can be assembled by an assembler.

5.4 The External Symbols Option - "e"

The "e" option controls the amount of information produced in the listing file. When specified, a list of all externally defined symbols is produced in the listing file.

Example:

```
C>wdis calendar -e
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.obj` and produce the output, with a list of all external symbols, on the screen. A sample list of external symbols is shown below.

```
List of external symbols

Symbol
----------------
_ _ _ iob          0000032f 00000210 000001f4 00000158 00000139
_ _ CHK            00000381 00000343 000002eb 00000237 000000cb 00000006
Box_               000000f2
Calendar_           000000a7 00000079 00000049
ClearScreen_       00000016
fflush_            00000334 00000215 000001f9 0000015d 0000013e
int386_            000003af 00000372
Line_              000002db 000002b5 00000293 00000274 0000025a
localtime_          00000028
memset_            00000308
PosCursor_          0000031e 000001e1 00000148 00000123 000000b6
printf_            00000327 00000208 000001ec 00000150 00000131
strlen_            00000108
time_              0000001d
-------------------------------------------------------------
```

Each externally defined symbol is followed by a list of location counter values indicating where the symbol is referenced.

The "e" option is ignored when the "a" option is specified.

5.5 The FPU emulator fixups Option - "ff"

The "ff" option causes the FPU emulator fixups will be printed as comment line before Intel FPU instruction.

```
; FPU fixup FIDRQQ
fld         tbyte ptr [bx]
```

5.6 The Alternate Addressing Form Option - "fi"

The "fi" option causes an alternate syntactical form of the based or indexed addressing mode of the 80x86 to be used in an instruction. For example, the following form is used by default for Intel instructions.

```
mov ax,-2[bp]
```

If the "fi" option is specified, the following form is used.

```
mov ax,[bp-2]
```

5.7 The No Instruction Name Pseudonyms Option - "fp"

By default, AXP instruction name pseudonyms are emitted in place of actual instruction names. The Open Watcom AXP Assembler accepts instruction name pseudonyms. The "fp" option instructs the Open Watcom Disassembler to emit the actual instruction names instead.

5.8 The No Register Name Pseudonyms Option - "fr"

By default, AXP register names are emitted in pseudonym form. The Open Watcom AXP Assembler accepts register pseudonyms. The "fr" option instructs the Open Watcom Disassembler to display register names in their non-pseudonym form.

5.9 The Uppercase Instructions/Registers Option - "fu"

The "fu" option instructs the Open Watcom Disassembler to display instruction and register names in uppercase characters. The default is to display them in lowercase characters.

5.10 The Listing Option - "l[=<list_file>]"

By default, the Open Watcom Disassembler produces its output to the terminal. The "l" (lowercase L) option instructs the Open Watcom Disassembler to produce the output to a listing file. The default file name of the listing file is the same as the file name of the object file. The default file extension of the listing file is `.lst`.

Example:

```
C>wdis calendar -l
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.obj` and produce the output to a listing file called `calendar.lst`.

An alternate form of this option is "l=<list_file>". With this form, you can specify the name of the listing file. When specifying a listing file, a file extension of `.lst` is assumed if none is specified.

Example:

```
C>wdis calendar -l=calendar.lis
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.obj` and produce the output to a listing file called `calendar.lis`.

5.11 The Public Symbols Option - "p"

The "p" option controls the amount of information produced in the listing file. When specified, a list of all public symbols is produced in the listing file.

Example:

```
C>wdis calendar -p
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `calendar.obj` and produce the output, with a list of all exported symbols, to the screen. A sample list of public symbols is shown below.

The following is a list of public symbols in 80x86 code.

```
List of public symbols

SYMBOL                  SECTION                 OFFSET
-------------------------------------------------------
main_                   _ TEXT                  000002C0
void near Box( int, int, int, int )
                        _ TEXT                  00000093
void near Calendar( int, int, int, int, int, char near * )
                        _ TEXT                  0000014A
void near ClearScreen() _ TEXT                  00000000
void near Line( int, int, int, char, char, char )
                        _ TEXT                  00000036
void near PosCursor( int, int )
                        _ TEXT                  0000001A
```

The following is a list of public symbols in Alpha AXP code.

```
List of public symbols

SYMBOL                  SECTION                 OFFSET
-------------------------------------------------------
main                    .text                   000004F0
void near Box( int, int, int, int )
                        .text                   00000148
void near Calendar( int, int, int, int, int, char near * )
                        .text                   00000260
void near ClearScreen() .text                   00000000
void near Line( int, int, int, char, char, char )
                        .text                   00000060
void near PosCursor( int, int )
                        .text                   00000028
```

The "p" option is ignored when the "a" option is specified.

5.12 Retain C++ Mangled Names - "m"

The "m" option instructs the Open Watcom Disassembler to retain C++ mangled names rather than displaying their demangled form. The default is to interpret mangled C++ names and display them in a somewhat more intelligible form.

5.13 The Source Option - "s[=<source_file>]"

The "s" option causes the source lines corresponding to the assembly language instructions to be produced in the listing file. The object file must contain line numbering information. That is, the "d1" or "d2" option must have been specified when the source file was compiled. If no line numbering information is present in the object file, the "s" option is ignored.

The following defines the order in which the source file name is determined when the "s" option is specified.

1. If present, the source file name specified on the command line.
2. The name from the module header record.
3. The object file name.

In the following example, we have compiled the source file `mysrc.c` with "d1" debugging information. We then disassemble it as follows:

Example:

```
C>wdis mysrc -s -l
```

In the above example, the Open Watcom Disassembler is instructed to disassemble the contents of the file `mysrc.obj` and produce the output to the listing file `mysrc.lst`. The source lines are extracted from the file `mysrc.c`.

An alternate form of this option is "s=<source_file>". With this form, you can specify the name of the source file.

Example:

```
C>wdis mysrc -s=myprog.c -l
```

The above example produces the same result as in the previous example except the source lines are extracted from the file `myprog.c`.

5.14 An Example

Consider the following program contained in the file `hello.c`.

```
#include <stdio.h>

void main()
{
    printf( "Hello world\n" );
}
```

Compile it with the "d1" option. An object file called `hello.obj` will be produced. The "d1" option causes line numbering information to be generated in the object file. We can use the Open Watcom Disassembler to disassemble the contents of the object file by issuing the following command.

```
C>wdis hello -l -e -p -s -fu
```

The output will be written to a listing file called `hello.lst` (the "l" option was specified"). It will contain a list of external symbols (the "e" option was specified), a list of public symbols (the "p" option was specified) and the source lines corresponding to the assembly language instructions (the "s" option was specified). The source input file is called `hello.c`. The register names will be displayed in upper case (the "fu" option was specified). The output, shown below, is the result of using the Open Watcom C++ compiler.

The following is a disassembly of 80x86 code.

```
Module: HELLO.C
GROUP: 'DGROUP' CONST,CONST2,_ DATA,_ BSS

Segment: _ TEXT DWORD USE32 0000001A bytes

#include <stdio.h>

void main()
0000                    main_ :
0000    68 08 00 00 00      PUSH      0x00000008
0005    E8 00 00 00 00      CALL      _ _ CHK

{
    printf( "Hello world\n" );
000A    68 00 00 00 00      PUSH      offset L$1
000F    E8 00 00 00 00      CALL      printf_
0014    83 C4 04            ADD       ESP,0x00000004

}
0017    31 C0               XOR       EAX,EAX
0019    C3                  RET

Routine Size: 26 bytes,    Routine Base: _ TEXT + 0000

No disassembly errors

List of external references

SYMBOL
-------
_ _ CHK                 0006
printf_                 0010

Segment: CONST DWORD USE32 0000000D bytes
0000                    L$1:
0000    48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00          Hello world..



BSS Size: 0 bytes

List of public symbols

SYMBOL                  SECTION                 OFFSET
------------------------------------------------------
main_                   _ TEXT                  00000000
```

The following is a disassembly of Alpha AXP code.

```
                                .new_ section .text, "crx4"

#include <stdio.h>

void main()
0000                    main:
0000    23DEFFF0            LDA           SP,-0x10(SP)
0004    B75E0000            STQ           RA,(SP)

{
    printf( "Hello world\n" );
0008    261F0000            LDAH          A0,h^L$0(R31)
000C    22100000            LDA           A0,l^L$0(A0)
0010    43F00010            SEXTL         A0,A0
0014    D3400000            BSR           RA,j^printf

}
0018    201F0000            MOV           0x00000000,V0
001C    A75E0000            LDQ           RA,(SP)
0020    23DE0010            LDA           SP,0x10(SP)
0024    6BFA8001            RET           (RA)

Routine Size: 40 bytes,    Routine Base: .text + 0000

No disassembly errors

List of external references

SYMBOL
-------
printf                  0014

                                .new_ section .const, "drw4"
0000                    L$0:
0000    48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00 Hello world.....

                                .new_ section .const2, "drw4"

                                .new_ section .data, "drw4"

                                .new_ section .bss, "urw4"
0000                    .bss:

BSS Size: 0 bytes


                                .new_ section .pdata, "dr2"

0000                    //  Procedure descriptor for main
        main                                    //  BeginAddress      : 0
        main+0x28                               //  EndAddress        : 40
        00000000                                //  ExceptionHandler  : 0
        00000000                                //  HandlerData       : 0
        main+0x8                                //  PrologEnd         : 8

                                .new_ section .drectve, "iRr0"
0000    2D 64 65 66 61 75 6C 74 6C 69 62 3A 63 6C 69 62 -defaultlib:clib
0010    20 2D 64 65 66 61 75 6C 74 6C 69 62 3A 70 6C 69  -defaultlib:pli
0020    62 20 2D 64 65 66 61 75 6C 74 6C 69 62 3A 6D 61 b -defaultlib:ma
0030    74 68 20 00                                     th .

List of public symbols

SYMBOL                  SECTION                 OFFSET
------------------------------------------------------
main                    .text                   00000000
```

Let us create a form of the listing file that can be used as input to an assembler.

```
C>wdis hello -l=hello.asm -r -a
```

The output will be produced in the file `hello.asm`. The output, shown below, is the result of using the Open Watcom C++ compiler.

The following is a disassembly of 80x86 code.

```
.387
.386p
                PUBLIC  main_
                EXTRN   _ _ CHK:BYTE
                EXTRN   printf_ :BYTE
                EXTRN   _ _ _ wcpp_ 3_ data_ init_ fs_ root_ :BYTE
                EXTRN   _ cstart_ :BYTE
DGROUP          GROUP   CONST,CONST2,_ DATA,_ BSS
_ TEXT          SEGMENT DWORD PUBLIC USE32 'CODE'
                ASSUME CS:_ TEXT, DS:DGROUP, SS:DGROUP
main_ :
    PUSH      0x00000008
    CALL      near ptr _ _ CHK
    PUSH      offset L$1
    CALL      near ptr printf_
    ADD       ESP,0x00000004
    XOR       EAX,EAX
    RET
_ TEXT          ENDS
CONST           SEGMENT DWORD PUBLIC USE32 'DATA'
L$1:
                DB      0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f
                DB      0x72, 0x6c, 0x64, 0x0a, 0x00

CONST           ENDS
CONST2          SEGMENT DWORD PUBLIC USE32 'DATA'
CONST2          ENDS
_ DATA          SEGMENT DWORD PUBLIC USE32 'DATA'
_ DATA          ENDS
_ BSS           SEGMENT DWORD PUBLIC USE32 'BSS'
_ BSS           ENDS

                END
```

The following is a disassembly of Alpha AXP code.

```
.globl          main
.extrn          printf
.extrn          _ cstart_
.new_ section .text, "crx4"
main:
    LDA           $SP,-0x10($SP)
    STQ           $RA,($SP)
    LDAH          $A0,h^`L$0`($ZERO)
    LDA           $A0,l^`L$0`($A0)
    SEXTL         $A0,$A0
    BSR           $RA,j^printf
    MOV           0x00000000,$V0
    LDQ           $RA,($SP)
    LDA           $SP,0x10($SP)
    RET           $ZERO,($RA),0x00000001
```

```
.new_ section .const, "drw4"
`L$0`:
                .asciiz "Hello world\n"
                .byte   0x00, 0x00, 0x00



.new_ section .pdata, "dr2"

                //  0000                Procedure descriptor for main
                .long   main                    //  BeginAddress      : 0
                .long   main+0x28               //  EndAddress        : 40
                .long   00000000                //  ExceptionHandler  : 0
                .long   00000000                //  HandlerData       : 0
                .long   main+0x8                //  PrologEnd         : 8

.new_ section .drectve, "iRr0"
                .asciiz "-defaultlib:clib -defaultlib:plib -defaultlib:math "
```

6 Optimization of Far Calls

Optimization of far calls can result in smaller executable files and improved performance. It is most useful when the automatic grouping of logical segments into physical segments takes place. Note that, by default, automatic grouping is performed by the Open Watcom Linker.

The Open Watcom C, C++ and FORTRAN 77 compilers automatically enable the far call optimization. The Open Watcom Linker will optimize far calls to procedures that reside in the same physical segment as the caller. For example, a large code model program will probably contain many far calls to procedures in the same physical segment. Since the segment address of the caller is the same as the segment address of the called procedure, only a near call is necessary. A near call does not require a relocation entry in the relocation table of the executable file whereas a far call does. Thus, the far call optimization will result in smaller executable files that will load faster. Furthermore, a near call will generally execute faster than a far call, particularly on 286 and 386-based machines where, for applications running in protected mode, segment switching is fairly expensive.

The following describes the far call optimization. The **call far label** instruction is converted to one of the following sequences of code.

```
push   cs                 seg    ss
call   near label         push   cs
nop                       call   near label
```

Notes:

1. The **nop** or **seg ss** instruction is present since a **call far label** instruction is five bytes. The **push cs** instruction is one byte and the **call near label** instruction is three bytes. The **seg ss** instruction is used because it is faster than the **nop** instruction.

2. The called procedure will still use a **retf** instruction but since the code segment and the near address are pushed on the stack, the far return will execute correctly.

3. The position of the padding instruction is chosen so that the return address is word aligned. A word aligned return address improves performance.

4. When two consecutive **call far label** instructions are optimized and the first **call far label** instruction is word aligned, the following sequence replaces both **call far label** instructions.

```
push    cs
call    near label1
seg     ss
push    cs
seg     cs
call    near label2
```

5. If your program contains only near calls, this optimization will have no effect.

A far jump optimization is also performed by the Open Watcom Linker. This has the same benefits as the far call optimization. A **jmp far label** instruction to a location in the same segment will be replaced by the following sequence of code.

```
jmp     near label
mov     ax,ax
```

Note that for 32-bit segments, this instruction becomes `mov eax,eax`.

6.1 Far Call Optimizations for Non-Open Watcom Object Modules

The far call optimization is automatically enabled when object modules created by the Open Watcom C, C++, or FORTRAN 77 compilers are linked. These compilers mark those segments in which this optimization can be performed. The following utility can be used to enable this optimization for object modules that have been created by other compilers or assemblers.

6.1.1 The Open Watcom Far Call Optimization Enabling Utility

Only DOS, OS/2 and Windows-hosted versions of the Open Watcom Far Call Optimization Enabling Utility are available. A QNX-hosted version is not necessary since QNX-hosted development tools that generate object files, generate the necessary information that enables the far call optimization.

The format of the Open Watcom Far Call Optimization Enabling Utility is as follows. Items enclosed in square brackets are optional; items enclosed in braces may be repeated zero or more times.

FCENABLE { [option] [file] }

where ***description:***

option is an option and must be preceded by a dash ('-') or slash ('/').

file is a file specification for an object file or library file. If no file extension is specified, a file extension of "obj" is assumed. Wild card specifiers may be used.

The following describes the command line options.

b Do not create a backup file. By default, a backup file will be created. The backup file name will have the same file name as the input file and a file extension of "bob" for object files and "bak" for library files.

c Specify a list of class names, each separated by a comma. This enables the far call optimization for all segments belonging to the specified classes.

s Specify a list of segment names, each separated by a comma. This enables the far call optimization for all specified segments.

x Specify a list of ranges, each separated by a comma, for which no far call optimizations are to be made. A range has the following format.

```
seg_name start-end
        or
seg_name start:length
```

seg_name is the name of a segment. *start* is an offset into the specified segment defining the start of the range. *end* is an offset into the specified segment defining the end of the range. *length* is the number of bytes from *start* to be included in the range. All values are assumed to be hexadecimal.

Notes:

1. If more than one class list or segment list is specified, only the last one is used. A class or segment list applies to all object and library files regardless of their position relative to the class or segment list.

2. A range list applies only to the first object file following the range specification. If the object file contains more than one module, the range list will only apply to the first module in the object file.

The following examples illustrate the use of the Open Watcom Far Call Optimization Enabling Utility.

Example:
```
fcenable -c code *.obj
```

In the above example, the far call optimization will be enabled for all segments belonging to the "code" class.

Example:
```
fcenable -s _text *.obj
```

In the above example, the far call optimization will be enabled for all segments with name "_text".

Example:
```
fcenable -x special 0:400 asmfile.obj
```

In the above example, the far call optimization will be disabled for the first 1k bytes of the segment named "special" in the object file "asmfile".

Example:
```
fcenable -x special 0-ffffffff asmfile.obj
```

In the above example, the far call optimization will be disabled for the entire segment named "special" in the object file "asmfile".

7 The Open Watcom Exe2bin Utility

The exe2bin utility strips off the header of a DOS executable file and applies any necessary fixups. In addition, it is able to display the header and relocations of an executable file in human readable format.

When DOS executes a program (supplied as an ".exe" file) it first reads the header of the executable file and ensures there is enough memory to load the program. If there is, DOS loads the file — excluding the header — to memory. Before jumping to the entry point, DOS has to adjust a number of certain locations that depend on the load address of the program. These adjustments consist of the addition of the load address to each entry in the above mentioned list of relocations. These relocations are part of the header of an executable file. The load address may vary from invocation to invocation, this creates the need for the existence of relocations.

As exe2bin strips the executable header, the relocations are lost (among other things). This would render the resulting output useless, if exe2bin were not to apply the relocations as part of the conversion process. Just like DOS, exe2bin therefore needs to know the load address. This is supplied via an argument to exe2bin.

Some programs do not rely on the address they are being loaded at, and consequently do not contain any relocations. In this case exe2bin merely copies the contents of the input file (apart from the header) to the output file.

The phrase "binary part" (also "binary data") is used as a technical term in the documentation of exe2bin. It denotes the data following the header. The length of the binary data is determined by the header entries "Size mod 512", "Number of pages" and "Size of header". It is not directly related to the actual size of the input file.

> ***Note:*** Although Open Watcom Exe2bin is capable of producing DOS ".COM" executables, this functionality is only provided for compatibility with other tools. The preferred way of generating ".COM" executables is to use the Open Watcom Linker with directive `"format dos com".` Refer to the Open Watcom Linker Guide for details.

7.1 The Open Watcom Exe2bin Utility Command Line

The format of the Open Watcom Exe2bin command line is as follows. Items enclosed in square brackets ("[ ]") are optional.

> ***EXE2BIN [options] exe_file [bin_file]***

where ***description:***

options is a list of options, each preceded by a dash ("−"). On non-UNIX platforms, a slash ("/") may be also used instead of a dash. Options may be specified in any order. Supported options are:

- ***h*** display the executable file header
- ***r*** display the relocations of the executable file
- ***l=<seg>*** specify the load address of the binary file
- ***x*** enable extended capabilities of Open Watcom Exe2bin

exe_file is a file specification for a 16-bit DOS executable file used as input. If no file extension is specified, a file extension of ".exe" is assumed. Wild card specifiers may not be used.

bin_file is an optional file specification for a binary output file. If no file name is given, the extension of the input file is replaced by "bin" and taken as the name for the binary output file.

Description:

1. If are any relocations in the input file, the -l option becomes mandatory (and is useless otherwise).
2. If exe2bin is called without the -x option, certain restrictions to the input file apply (apart from being a valid DOS executable file):
   - the size of the binary data must be <= 64 KByte
   - no stack must be defined, i.e. ss:sp = 0x0000:0x0000
   - the code segment must be always zero, i.e. cs = 0x0000
   - the initial instruction pointer must be either ip = 0x0000 or ip = 0x0100

   None of the above restrictions apply if the -x option is supplied.
3. If cs:ip = 0x0000:0x0100 and the -x option is not specified, no relocations are allowed in the input file. Furthermore, exe2bin skips another 0x100 bytes following the header (in addition to the latter).

   This behaviour allows the creation of DOS ".COM" executables and is implemented for backward compatibility. It is however strongly suggested to use the Open Watcom Linker instead (together with directive `"format dos com"`).

The examples below illustrate the use of Open Watcom Exe2bin.

Example:

```
exe2bin prog.exe
```

Strips off the executable header from `prog.exe` and writes the binary part to `prog.bin`. If there are any relocations in `prog.exe` or if the input file violates any of the restrictions listed above, the execution of exe2bin fails.

Example:

```
exe2bin -x prog.exe
```

Same as above but the `"-x"` option relaxes certain restrictions.

> ***Note:*** Even if exe2bin is sucessfully invoked with identical input files as in the preceding examples (i.e. with vs. without -x) the output files may differ. This happens when cs:ip = 0x0000:0x0100 causes exe2bin to skip additional 0x100 bytes from the input file, if the user did not specify -x.

Example:

```
exe2bin -h prog.exe test.bin
```

Displays the header of `prog.exe` , strips it off and copies the binary part to `test.bin`.

Example:

```
exe2bin -h -r -x -l=0xE000 bios.exe bios.rom
```

Displays the header and the relocations (if any) of `bios.exe` strips the header and applies any fixups to (i.e. relocates) `bios.exe` as if it were to be loaded at 0xE000:0x0000. The result will be written to `bios.rom`

The above command line may serve as an example of creating a 128 KByte BIOS image for the PC-AT architecture.

7.2 Exe2bin Messages

This is a list of the diagnostic messages exe2bin may display, accompanied by more verbose descriptions and some possible causes.

Error opening %s for reading.
The input executable file could not be opened for reading.

Check that the input file exists and exe2bin has read permissions.

Error opening %s for writing.
The output binary file could not be opened for writing.

Make sure the media is not write protected, has enough free space to hold the output file, and exe2bin has write permissions.

Error allocating file I/O buffer.
There is not enough free memory to allocate a file buffer.

Error reading while copying data.
An error occured while reading the binary part of the input file.

This is most likely due to a corrupted executable header. Run exe2bin with the -h option and check the size reported. The size of the input file must be at least ("Number of pages" - 1) * 512 + "Size mod 512". Omit decrementing the number of pages if "Size mod 512" happens to equal zero.

Error writing while copying data.
The output binary file can not be written to.

Make sure the media has enough free space to hold the output file and is not removed while writing to it.

Error. %s has no valid executable header.
The signature (the first two bytes of the input file) does not match "MZ".

exe2bin can only use valid DOS executable files as input.

Error allocating/reading reloc-table.
There is either not enough free memory to allocate a buffer for the relocations (each relocation takes about 4 bytes) or there was an error while reading from the input file.

Error. Option "-l=<seg>" mandatory (there are relocations).
The executable file contains relocations. Therefore, exe2bin needs to know the segment the binary output file is supposed to reside at.

Either provide a segment as an argument to the -l option or rewrite your executable file to not contain any relocations.

Error: Binary part exceeds 64 KBytes.
The binary part of the input file is larger than 64 KBytes.

The restriction applies because the -x option was not specified. Check if the extended behaviour is suitable or rewrite the program to shorten the binary part.

Error: Stack segment defined.
The header defines an initial stack, i.e. ss:sp != 0x0000:0x0000.

The restriction applies because the -x option was not specified. Check if the extended behaviour is suitable or rewrite the program to not have a segment of class "stack".

Error: CS:IP neither 0x0000:0x0000 nor 0x0000:0x0100.
The header defines an initial cs:ip not matching any of the two values.

The restriction applies because the -x option was not specified. Check if the extended behaviour is suitable or rewrite the program to have a different entry point (cf. Open Watcom Linker `"option start"`).

Error: com-file must not have relocations.
Although the binary part is <= 64 KByte in length, there is no stack defined and the cs:ip is 0x0000:0x0100, i.e. exe2bin assumes you try to generate a ".COM" executable, there are relocations in the input file.

".COM" files are not allowed to contain relocations. Either produce an ".EXE" file instead or rewrite the program to avoid the need for relocations. In order to do the latter, look for statements that refer to segments or groups such as `mov ax, _TEXT` or `mov ax, DGROUP`.

Executable Image Utilities

8 The Open Watcom Patch Utility

8.1 Introduction

The Open Watcom Patch Utility is a utility program which may be used to apply patches or bug fixes to Open Watcom's compilers and its associated tools. As problems are reported and fixed, patches are created and made available on Open Watcom's BBS, Open Watcom's FTP site, or CompuServe for users to download and apply to their copy of the tools.

8.2 Applying a Patch

The format of the BPATCH command line is:

BPATCH [options] patch_file

The square brackets [ ] denote items which are optional.

where ***description:***

options is a list of valid Open Watcom Patch Utility options, each preceded by a dash ("−"). Options may be specified in any order. The possible options are:

- ***-p*** Do not prompt for confirmation
- ***-b*** Do not create a .BAK file
- ***-q*** Print current patch level of file

patch_file is the file specification for a patch file provided by Open Watcom.

Suppose a patch file called "wlink.a" is supplied by Open Watcom to fix a bug in the file "WLINK.EXE". The patch may be applied by typing the command:

```
bpatch wlink.a
```

The Open Watcom Patch Utility locates the file `C:\WATCOM\BINW\WLINK.EXE` using the **PATH** environment variable. The actual name of the executable file is extracted from the file `wlink.a`. It then verifies that the file to be patched is the correct one by comparing the size of the file to be patched to the expected size. If the file sizes match, the program responds with:

```
Ok to modify 'C:\WATCOM\BINW\WLINK.EXE'? [y|n]
```

If you respond with "yes", BPATCH will modify the indicated file. If you respond with "no", BPATCH aborts. Once the patch has been applied the resulting file is verified. First the file size is checked to make

sure it matches the expected file size. If the file size matches, a check-sum is computed and compared to the expected check-sum.

Notes:

1. If an error message is issued during the patch process, the file that you specified to be patched will remain unchanged.

2. If a sequence of patch files exist, such as "wlink.a", "wlink.b" and "wlink.c", the patches must be applied in order. That is, "wlink.a" must be applied first followed by "wlink.b" and finally "wlink.c".

8.3 Diagnostic Messages

If the patch cannot be successfully applied, one of the following error messages will be displayed.

Usage: BPATCH {-p} {-q} {-b} <file>
-p = Do not prompt for confirmation
-b = Do not create a .BAK file
-q = Print current patch level of file
The command line was entered with no arguments.

File '%s' has not been patched
This message is issued when the "-q" option is used and the file has not been patched.

File '%s' has been patched to level '%s'
This message is issued when the "-q" option is used and the file has been patched to the indicated level.

File '%s' has already been patched to level '%s' - skipping
This message is issued when the file has already been patched to the same level or higher.

Command line may only contain one file name
More than one file name is specified on the command line. Make sure that "/" is not used as an option delimiter.

Command line must specify a file name
No file name has been specified on the command line.

'%s' is not a Open Watcom patch file
The patch file is not of the required format. The required header information is not present.

'%s' is not a valid Open Watcom patch file
The patch file is not of the required format. The required header information is present but the remaining contents of the file have been corrupted.

'%s' is the wrong size (%lu1). Should be (%lu2)
The size of the file to be patched (%lu1) is not the same as the expected size (%lu2).

Cannot find '%s'
Cannot find the executable to be patched.

Cannot open '%s'

An error occurred while trying to open the patch file, the file to be patched or the resulting file.

Cannot read '%s'

An input error occurred while reading the old version of the file being patched.

Cannot rename '%s' to '%s'

The file to be patched could not be renamed to the backup file name or the resulting file could not be renamed to the name of the file that was patched.

Cannot write to '%s'

An output error occurred while writing to the new version of the file to be patched.

I/O error processing file '%s'

An error occurred while seeking in the specified file.

No memory for %s

An attempt to allocate memory dynamically failed.

Patch program aborted!

This message is issued if you answered no to the "OK to modify" prompt.

Resulting file has wrong checksum (%lu) - Should be (%lu2)

The check-sum of the resulting file (%lu) does not match the expected check-sum (%lu2). This message is issued if you have patched the wrong version.

Resulting file has wrong size (%lu1) - Should be (%lu2)

The size of the resulting file (%lu1) does not match the expected size (%lu2). This message is issued if you have patched the wrong version.

9 The Open Watcom Strip Utility

9.1 Introduction

The Open Watcom Strip Utility may be used to manipulate information that is appended to the end of an executable file. The information can be either one of two things:

1. Symbolic debugging information
2. Resource information

This information can be added or removed from the executable file. Symbolic debugging information is placed at the end of an executable file by the Open Watcom Linker or the Open Watcom Strip Utility. Resource information is placed at the end of an executable by a resource compiler or the Open Watcom Strip Utility.

Once a program has been debugged, the Open Watcom Strip Utility allows you to remove the debugging information from the executable file so that you do not have to remove the debugging directives from the linker directive file and link your program again. Removal of the debugging information reduces the size of the executable image.

All executable files generated by the Open Watcom Linker can be specified as input to the Open Watcom Strip Utility. Note that for executable files created for Novell's NetWare operating system, debugging information created using the "NOVELL" option in the "DEBUG" directive cannot be removed from the executable file. You must remove the "DEBUG" directive from the directive file and re-link your application.

The Open Watcom Strip Utility currently runs under the following operating systems.

- DOS
- OS/2
- QNX
- Windows NT/2000/XP
- Windows 95/98/Me

9.2 The Open Watcom Strip Utility Command Line

The Open Watcom Strip Utility command line syntax is:

WSTRIP [options] input_file [output_file] [info_file]

where:

[] The square brackets denote items which are optional.

options

-n (noerrors) Do not issue any diagnostic message.

-q (quiet) Do not print any informational messages.

-r (resources) Process resource information rather than debugging information.

-a (add) Add information rather than remove information.

input_file is a file specification for the name of an executable file. If no file extension is specified, the Open Watcom Strip Utility will assume one of the following extensions: "exe", "dll", "exp", "rex", "nlm", "dsk", "lan", "nam", "msl", "cdm", "ham", "qnx" or no file extension. Note that the order specified in the list of file extensions is the order in which the Open Watcom Strip Utility will select file extensions.

output_file is an optional file specification for the output file. If no file extension is specified, the file extension specified in the input file name will be used for the output file name. If "." is specified, the input file name will be used.

info_file is an optional file specification for the file in which the debugging or resource information is to be stored (when removing information) or read (when adding information). If no file extension is specified, a file extension of "sym" is assumed for debugging information and "res" for resource information. To specify the name of the information file but not the name of an output file, a "." may be specified in place of *output_file*.

Description:

1. If the "r" (resource) option is not specified then the default action is to add/remove symbolic debugging information.

2. If the "a" (add) option is not specified then the default action is to remove information.

3. If *output_file* is not specified, the debugging or resource information is added to or removed from *input_file*.

4. If *output_file* is specified, *input_file* is copied to *output_file* and the debugging or resource information is added to or removed from *output_file*. *input_file* remains unchanged.

5. If *info_file* is specified then the debugging or resource information that is added to or removed from the executable file is read from or written to this file. The debugging or resource information may be appended to the executable by specifying the "a" (add) option. Also, the debugging information may be appended to the executable by concatenating the debugging information file to the end of the executable file (the files must be treated as binary files).

6. During processing, the Open Watcom Strip Utility will create a temporary file, ensuring that a file by the chosen name does not already exist.

9.3 Strip Utility Messages

The following messages may be issued by the Open Watcom Strip Utility.

Usage: WSTRIP [options] input_file [output_file] [info_file]
options: (-option is also accepted)
/n don't print warning messages
/q don't print informational messages
/r process resource information rather than debugging information
/a add information rather than delete information
input_file: executable file
output_file: optional output executable or '.'
info_file: optional output debugging or resource information file or input debugging or resource informational file
The command line was entered with no arguments.

Too low on memory
There is not enough free memory to allocate file buffers.

Unable to find '%s'
The specified file could not be located.

Cannot create temporary file
All the temporary file names are in use.

Unable to open '%s' to read
The input executable file cannot be opened for reading.

'%s' is not a valid executable file
The input file has invalid executable file header information.

'%s' does not contain debugging information
There is nothing to strip from the specified executable file.

Seek error on '%s'
An error occurred during a seek operation on the specified file.

Unable to create output file '%s'
The output file could not be created. Check that the output disk is not write-protected or that the specified output file is not marked "read-only".

Unable to create symbol file '%s'
The symbol file could not be created.

Error reading '%s'
An error occurred while reading the input executable file.

Error writing to '%s'
An error occurred while writing the output executable file or the symbol file. Check the

amount of free space on the output disk. If the input and output files reside on the same disk, there might not be enough room for a second copy of the executable file during processing.

Cannot erase file '%s'

The input executable file is probably marked "read-only" and therefore could not be erased (the input file is erased whenever the output file has the same name).

Cannot rename file '%s'

The output executable file could not be renamed. Ordinarily, this should never occur.

The Make/Touch Utilities

10 The Open Watcom Make Utility

10.1 Introduction

The Open Watcom Make utility is useful in the development of programs and text processing but is general enough to be used in many different applications. Make uses the fact that each file has a time-stamp associated with it that indicates the last time the file was updated. Make uses this time-stamp to decide which files are out of date with respect to each other. For instance, if we have an input data file and an output report file we would like the output report file to accurately reflect the contents of the input data file. In terms of time-stamps, we would like the output report to have a more recent time-stamp than the input data file (we will say that the output report file should be "younger" than the input data file). If the input file had been modified then we would know from the younger time-stamp (in comparison to the report file) that the report file was out of date and should be updated. Make may be used in this and many other situations to ensure that files are kept up to date.

Some readers will be quite familiar with the concepts of the Make file maintenance tool. Open Watcom Make is patterned after the Make utility found on UNIX systems. The next major section is simply intended to summarize, for reference purposes only, the syntax and options of Make's command line and special macros. Subsequent sections go into the philosophy and capabilities of Open Watcom Make. If you are not familiar with the capabilities of the Make utility, we recommend that you skip to the next major section entitled "Dependency Declarations" and read on.

10.2 Open Watcom Make Reference

The following sub-sections serve as a reference guide to the Open Watcom Make utility.

10.2.1 Open Watcom Make Command Line Format

The formal Open Watcom Make command line syntax is shown below.

WMAKE [options] [macro_defs] [targets]

As indicated by the square brackets [ ], all items are optional.

options is a list of valid Open Watcom Make options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

macro_defs is a list of valid Open Watcom Make macro definitions. Macro definitions are of the form:

```
A=B
```

and are readily identified by the presence of the "=" (the "#" character may be used instead of the "=" character if necessary). Surround the definition with quotes (") if it contains

blanks (e.g., "debug_opt=debug all"). The macro definitions specified on the command line supersede any macro definitions defined in makefiles. Macro names are case-insensitive unless the "ms" option is used to select Microsoft NMAKE mode.

targets is one or more targets described in the makefile.

10.2.2 Open Watcom Make Options Summary

In this section, we present a terse summary of the Open Watcom Make options. This summary is displayed on the screen by simply entering "WMAKE ?" on the command line.

Example:

```
C>wmake ?
```

-a	make all targets by ignoring time-stamps
-b	block/ignore all implicit rules
-c	do not verify the existence of files made
-d	debug mode - echo all work as it progresses
-e	always erase target after error/interrupt (disables prompting)
-f	the next parameter is a name of dependency description file
-h	do not print out Make identification lines (no header)
-i	ignore return status of all commands executed
-k	on error/interrupt: continue on next target
-l	the next parameter is the name of a output log file
-m	do not search for MAKEINIT file
-ms	Microsoft NMAKE mode
-n	no execute mode - print commands without executing
-o	use circular implicit rule path
-p	print the dependency tree as understood from the file
-q	query mode - check targets without updating them
-r	do not use default definitions
-s	silent mode - do not print commands before execution
-sn	noisy mode - always print commands before execution
-t	touch files instead of executing commands
-u	UNIX compatibility mode
-v	verbose listing of inline files
-y	show why a target will be updated
-z	do not erase target after error/interrupt (disables prompting)

10.2.3 Command Line Options

Command line options, available with Open Watcom Make, allow you to control the processing of the makefile.

a

make all targets by ignoring time-stamps

The "a" option is a safe way to update every target. For program maintenance, it is the preferred method over deleting object files or touching source files.

b

block/ignore all implicit rules

The "b" option will indicate to Make that you do not want any implicit rule checking done. The "b" option is useful in makefiles containing double colon "::" explicit rules because an implicit rule search is conducted after a double colon "::" target is updated. Including the directive `.BLOCK` in a makefile also will disable implicit rule checking.

c

do not verify the existence of files made

Make will check to ensure that a target exists after the associated command list is executed. The target existence checking may be disabled with the "c" option. The "c" option is useful in processing makefiles that were developed with other Make utilities. The `.NOCHECK` directive is used to disable target existence checks in a makefile.

d

debug mode - echo all work as it progresses

The "d" option will print out information about the time-stamp of files and indicate how the makefile processing is proceeding.

e

always erase target after error/interrupt (disables prompting)

The "e" option will indicate to Make that, if an error or interrupt occurs during makefile processing, the current target being made may be deleted without prompting. The `.ERASE` directive may be used as an equivalent option in a makefile.

f

the next parameter is a name of dependency description file

The "f" option specifies that the next parameter on the command line is the name of a makefile which must be processed. If the "f" option is specified then the search for the default makefile named "MAKEFILE" is not done. Any number of makefiles may be processed with the "f" option.

Example:

```
wmake /f myfile
wmake /f myfile1 /f myfile2
```

h

do not print out Make identification lines (no header)

The "h" option is useful for less verbose output. Combined with the "q" option, this allows a batch file to silently query if an application is up to date. Combined with the "n" option, a batch file could be produced containing the commands necessary to update the application.

i

ignore return status of all commands executed

The "i" option is equivalent to the `.IGNORE` directive.

k

on error/interrupt: continue on next target

Make will stop updating targets when a non-zero status is returned by a command. The "k" option will continue processing targets that do not depend on the target that caused the error. The `.CONTINUE` directive in a makefile will enable this error handling capability.

l

the next parameter is the name of a output log file

Make will output an error message when a non-zero status is returned by a command. The "l" option specifies a file that will record all error messages output by Make during the processing of the makefile.

m

do not search for the MAKEINIT file

The default action for Make is to search for an initialization file called "MAKEINIT" or "TOOLS.INI" if the "ms" option is set. The "m" option will indicate to Make that processing of the MAKEINIT file is not desired.

ms

Microsoft NMAKE mode

The default action for Make is to process makefiles using Open Watcom syntax rules. The "ms" option will indicate to Make that it should process makefiles using Microsoft syntax rules. For example, the line continuation in NMAKE is a backslash ("\") at the end of the line.

n

no execute mode - print commands without executing

The "n" option will print out what commands should be executed to update the application without actually executing them. Combined with the "h" option, a batch file could be produced which would contain the commands necessary to update the application.

Example:

```
wmake /h /n >update.bat
update
```

This is useful for applications which require all available resources (memory and devices) for executing the updating commands.

o

use circular implicit rule path

When this option is specified, Make will use a circular path specification search which may save on disk activity for large makefiles. The "o" option is equivalent to the `.OPTIMIZE` directive.

p

print out makefile information

The "p" option will cause Make to print out information about all the explicit rules, implicit rules, and macro definitions.

q

query mode - check targets without updating them

The "q" option will cause Make to return a status of 1 if the application requires updating; it will return a status of 0 otherwise. Here is a example batch file using the "q" option:

Example:

```
wmake /q
if errorstatus 0 goto noupdate
wmake /q /h /n >\tmp\update.bat
call \tmp\update.bat
:noupdate
```

r

do not use default definitions

The default definitions are:

```
__MAKEOPTS__  = <options passed to WMAKE>
__MAKEFILES__  = <list of makefiles>
__VERSION__  = <version number>
__LOADDLL__ = defined if DLL loading supported
__MSDOS__  =  defined if MS/DOS version
__NT__  = defined if Windows NT version
__NT386__  = defined if x86 Windows NT version
__OS2__  = defined if OS/2 version
__QNX__  = defined if QNX version
__LINUX__  = defined if Linux version
__LINUX386__  = defined if x86 Linux version
__UNIX__  = defined if QNX or Linux version
MAKE = <name of file containing WMAKE>
#endif
# clear .EXTENSIONS list
.EXTENSIONS:

# In general,
# set .EXTENSIONS list as follows
.EXTENSIONS: .exe .nlm .dsk .lan .exp &
             .lib .obj &
             .i &
             .asm .c .cpp .cxx .cc .for .pas .cob &
             .h .hpp .hxx .hh .fi .mif .inc
```

For Microsoft NMAKE compatibility (when you use the "ms" option), the following default definitions are established.

```
# For Microsoft NMAKE compatibility switch,
# set .EXTENSIONS list as follows
.EXTENSIONS: .exe .obj .asm .c .cpp .cxx &
             .bas .cbl .for .f .f90 .pas .res .rc

%MAKEFLAGS=$(%MAKEFLAGS) $(_ _ MAKEOPTS_ _ )
MAKE=<name of file containing WMAKE>
AS=ml
BC=bc
CC=cl
COBOL=cobol
CPP=cl
CXX=cl
FOR=fl
PASCAL=pl
RC=rc
.asm.exe:
    $(AS) $(AFLAGS) $*.asm
.asm.obj:
    $(AS) $(AFLAGS) /c $*.asm
.c.exe:
    $(CC) $(CFLAGS) $*.c
.c.obj:
    $(CC) $(CFLAGS) /c $*.c
.cpp.exe:
    $(CPP) $(CPPFLAGS) $*.cpp
.cpp.obj:
    $(CPP) $(CPPFLAGS) /c $*.cpp
.cxx.exe:
    $(CXX) $(CXXFLAGS) $*.cxx
.cxx.obj:
    $(CXX) $(CXXFLAGS) $*.cxx
.bas.obj:
    $(BC) $(BFLAGS) $*.bas
.cbl.exe:
    $(COBOL) $(COBFLAGS) $*.cbl, $*.exe;
.cbl.obj:
    $(COBOL) $(COBFLAGS) $*.cbl;
.f.exe:
    $(FOR) $(FFLAGS) $*.f
.f.obj:
    $(FOR) /c $(FFLAGS) $*.f
.f90.exe:
    $(FOR) $(FFLAGS) $*.f90
.f90.obj:
    $(FOR) /c $(FFLAGS) $*.f90
.for.exe:
    $(FOR) $(FFLAGS) $*.for
.for.obj:
    $(FOR) /c $(FFLAGS) $*.for
.pas.exe:
    $(PASCAL) $(PFLAGS) $*.pas
.pas.obj:
    $(PASCAL) /c $(PFLAGS) $*.pas
.rc.res:
    $(RC) $(RFLAGS) /r $*
```

For OS/2, the __MSDOS__ macro will be replaced by __OS2__ and for Windows NT, the __MSDOS__ macro will be replaced by __NT__.

For UNIX make compatibility (when you use the "u" option), the following default definition is established.

```
.EXTENSIONS: .exe .obj .c .y .l .f

%MAKEFLAGS=$(%MAKEFLAGS) $(__MAKEOPTS__)
MAKE=<name of file containing WMAKE>
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LDFLAGS=
CC=cl
FC=fl
.asm.exe:
    $(AS) $(AFLAGS) $*.asm
.c.exe:
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<
.f.exe:
    $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<
.c.obj:
    $(CC) $(CFLAGS) -c $<
.f.obj:
    $(FC) $(FFLAGS) -c $<
.y.obj:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    del y.tab.c
    move y.tab.obj $@
.l.obj:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    del lex.yy.c
    move lex.yy.obj $@
.y.c:
    $(YACC) $(YFLAGS) $<
    move y.tab.c $@
.l.c:
    $(LEX) $(LFLAGS) $<
    move lex.yy.c $@
```

The "r" option will disable these definitions before processing any makefiles.

s

silent mode - do not print commands before execution

The "s" option is equivalent to the .SILENT directive.

sn

noisy mode - always print commands before execution

The "sn" option overrules all silencing controls. It can be used to assist in debugging a makefile.

t

touch files instead of executing commands

Sometimes there are changes which are purely cosmetic (adding a comment to a source file) that will cause targets to be updated needlessly thus wasting computer resources. The "t" option will make files appear younger without altering their contents. The "t" option is useful but should be used with caution.

u

UNIX compatibility mode

The "u" option will indicate to Make that the line continuation character should be a backslash "\" rather than an ampersand "&".

v

The "v" option enables a verbose listing of inline temporary files.

y

The "y" option enables the display of a progress line denoting which dependent file has caused a target to be updated. This is a useful option for helping to debug makefiles.

z

do not erase target after error/interrupt (disables prompting)

The "z" option will indicate to Make that if an error or interrupt occurs during makefile processing then the current target being made should not be deleted. The `.HOLD` directive in a makefile has the same effect as the "z" option.

10.2.4 Special Macros

Open Watcom Make has many different special macros. Here are some of the simpler ones.

Macro	*Expansion*
$$	represents the character "$"
$#	represents the character "#"
$@	full file name of the target
$*	target with the extension removed
$<	list of all dependents
$?	list of dependents that are younger than the target

The following macros are for more sophisticated makefiles.

Macro	*Expansion*
__MSDOS__	This macro is defined in the MS/DOS environment.
__NT__	This macro is defined in the Windows NT environment.
__OS2__	This macro is defined in the OS/2 environment.

__LINUX__	This macro is defined in the Linux environment.
__QNX__	This macro is defined in the QNX environment.
__UNIX__	This macro is defined in the Linux or QNX environment.
__MAKEOPTS__	contains all of the command line options that WMAKE was invoked with except for any use of the "f" or "n" options.
__MAKEFILES__	contains the names of all of the makefiles processed at the time of expansion (includes the file currently being processed)
MAKE	contains the full name of the file that contains WMAKE
__VERSION__	contains the wmake version.

The next three tables contain macros that are valid during execution of command lists for explicit rules, implicit rules, and the `.ERROR` directive. The expansion is presented for the following example:

Example:

```
a:\dir\target.ext : b:\dir1\dep1.ex1 c:\dir2\dep2.ex2
```

Macro	*Expansion*
$^@	a:\dir\target.ext
$^*	a:\dir\target
$^&	target
$^.	target.ext
$^:	a:\dir\

Macro	*Expansion*
$[@	b:\dir1\dep1.ex1
$[*	b:\dir1\dep1
$[&	dep1
$[.	dep1.ex1
$[:	b:\dir1\

Macro	*Expansion*
$]@	c:\dir2\dep2.ex2
$]*	c:\dir2\dep2
$]&	dep2
$].	dep2.ex2
$]:	c:\dir2\

10.3 Dependency Declarations

In order for Open Watcom Make to be effective, a list of file dependencies must be declared. The declarations may be entered into a text file of any name but Make will read a file called "MAKEFILE" by default if it is invoked as follows:

Example:
```
C>wmake
```

If you want to use a file that is not called "MAKEFILE" then the command line option "f" will cause Make to read the specified file instead of the default "MAKEFILE".

Example:
```
C>wmake /f myfile
```

We will now go through an example to illustrate how Make may be used for a simple application. Suppose we have an input file, a report file, and a report generator program then we may declare a dependency as follows:

```
#
# (a comment in a makefile starts with a "#")
# simple dependency declaration
#
balance.lst : ledger.dat
        doreport
```

Note that the dependency declaration starts at the beginning of a line while commands always have at least one blank or tab before them. This form of a dependency declaration is called an *explicit rule.* The file "BALANCE.LST" is called the *target* of the rule. The *dependent* of the rule is the file "LEDGER.DAT" while "DOREPORT" forms one line of the *rule command list.* The dependent is separated from the target by a colon.

> ***Hint:*** A good habit to develop is to always put spaces around the colon so that it will not be confused with drive specifications (e.g., a:).

The explicit rule declaration indicates to Make that the program "DOREPORT" should be executed if "LEDGER.DAT" is younger than "BALANCE.LST" or if "BALANCE.LST" does not yet exist. In general, if the dependent file has a more recent modification date and time than the target file then Open Watcom Make will execute the specified command.

> ***Note:*** The terminology employed here is used by S.I.Feldman of Bell Laboratories in *Make - A Program for Maintaining Computer Programs.* http://www.softlab.ntua.gr/facilities/documentation/unix/docs/make.txt has a copy of this seminal article. Confusion often arises from the use of the word "dependent". In this context, it means "a subordinate part". In the example, "LEDGER.DAT" is a subordinate part of the report "BALANCE.LST".

10.4 Multiple Dependents

Suppose that our report "BALANCE.LST" becomes out-of-date if any of the files "LEDGER.DAT", "SALES.DAT" or "PURCHASE.DAT" are modified. We may modify the dependency rule as follows:

```
#
# multiple dependents rule
#
balance.lst : ledger.dat sales.dat purchase.dat
        doreport
```

This is an example of a rule with multiple dependents. In this situation, the program "DOREPORT" should be executed if any of "LEDGER.DAT", "SALES.DAT" or "PURCHASE.DAT" are younger than "BALANCE.LST" or if "BALANCE.LST" does not yet exist. In cases where there are multiple dependents, if any of the dependent files has a more recent modification date and time than the target file then Open Watcom Make will execute the specified command.

10.5 Multiple Targets

Suppose that the "DOREPORT" program produces two reports. If both of these reports require updating as a result of modification to the dependent files, we could change the rule as follows:

```
#
# multiple targets and multiple dependents rule
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport
```

Suppose that you entered the command:

```
wmake
```

which causes Make to start processing the rules described in "MAKEFILE". In the case where multiple targets are listed in the makefile, Make will, by default, process only the first target it encounters. In the example, Make will check the date and time of "BALANCE.LST" against its dependents since this is the first target listed.

To indicate that some other target should be processed, the target is specified as an argument to the Make command.

Example:

```
wmake summary.lst
```

There are a number of interesting points to consider:

1. By default, Make will only check that the target file exists after the command ("DOREPORT" in this example) is executed. It does not check that the target's time-stamp shows it to be younger. If the target file does not exist after the command has been executed, an error is reported.

2. There is no guarantee that the command you have specified does update the target file. In other words, simply because you have stated a dependency does not mean that one exists.

3. Furthermore, it is not implied that other targets in our list will not be updated. In the case of our example, you can assume that we have designed the "doreport" command to update both targets.

10.6 Multiple Rules

A makefile may consist of any number of rules. Note that the following:

```
target1 target2 : dependent1 dependent2 dependent3
        command list
```

is equivalent to:

```
target1 : dependent1 dependent2 dependent3
        command list

target2 : dependent1 dependent2 dependent3
        command list
```

Also, the rules may depend on the targets of other rules.

```
#
# rule 1: this rule uses rule 2
#
balance.lst summary.lst : ledger.dat sales.dat purchase.dat
        doreport

#
# rule 2: used by rules 1 and 3
#
sales.dat : canada.dat england.dat usa.dat
        dosales

#
# rule 3: this rule uses rule 2
#
year.lst : ledger.dat sales.dat purchase.dat
        doyearly
```

The dependents are checked to see if they are the targets of any other rules in the makefile in which case they are updated. This process of updating dependents that are targets in other rules continues until a rule is reached that has only simple dependents that are not targets of rules. At this point, if the target does not exist or if any of the dependents is younger than the target then the command list associated with the rule is executed.

> ***Hint:*** The term "updating", in this context, refers to the process of checking the time-stamps of dependents and running the specified command list whenever they are out-of-date. Whenever a dependent is the target of some other rule, the dependent must be brought up-to-date first. Stated another way, if "A" depends on "B" and "B" depends on "C" and "C" is younger than "B" then we must update "B" before we update "A".

Make will check to ensure that the target exists after its associated command list is executed. The target existence checking may be disabled in two ways:

1. use the command line option "c"
2. use the .NOCHECK directive.

The rule checking returns to the previous rule that had the target as a dependent. Upon returning to the rule, the command list is executed if the target does not exist or if any of the updated dependents are now younger than the target. If you were to type:

```
wmake
```

here are the steps that would occur with the previous makefile:

```
update(balance.lst) (rule 1)

  update(ledger.dat)             (not a target)
  update(sales.dat)              (found rule 2)

    update(canada.dat)           (not a target)
    update(england.dat)          (not a target)
    update(usa.dat)              (not a target)
    IF sales.dat does not exist                          OR
       any of (canada.dat,england.dat,usa.dat)
         is younger than sales.dat
    THEN execute "dosales"

  update(purchase.dat)           (not a target)
  IF balance.lst does not exist                          OR
     any of (ledger.dat,sales.dat,purchase.dat)
       is younger than (balance.lst)
  THEN execute "doreport"
```

The third rule in the makefile will not be included in this update sequence of steps. Recall that the default target that is "updated" is the first target in the first rule encountered in the makefile. This is the default action taken by Make when no target is specified on the command line. If you were to type:

```
wmake year.lst
```

then the file "YEAR.LST" would be updated. As Make reads the rules in "MAKEFILE", it discovers that updating "YEAR.LST" involves updating "SALES.DAT". The update sequence is similar to the previous example.

10.7 Command Lists

A command list is a sequence of one or more commands. Each command is preceded by one or more spaces or tabs. Command lists may also be used to construct inline files "on the fly". Macros substitute in command lists and in inline files. An inline file is introduced by "<<" in a command in a command list. Data to insert into that file is placed (left-justified) in the command list. The data is terminated by "<<" in the first column. It is not possible to place a line which starts "<<" in an inline file. More than one inline file may be created in a command. Data for each is placed in order of reference in the command.

In building the Open Watcom system, it is sometimes necessary to do some text substitution with a program called vi. This needs a file of instructions. The following simplifies an example used to build Open Watcom so that inline files may be shown. Without inline files, this is done as:

```
$(dllname).imp : $(dllname).lbc ../../trimlbc.vi
    cp $(dllname).lbc $(dllname).imp
    $(vi) -s ../../trimlbc.vi $(dllname).imp

where trimlbc.vi consists of
set magic
set magicstring = ()
atomic
%s/\.dll'//
%s/^(\+\+')(.*)('\.'.*')\.[0-9]+$/\1\2\3..'\2'/
x
```

A doubled "$" to produce a single dollar is notable when an inline file is used:

```
$(dllname).imp : $(dllname).lbc
    cp $(dllname).lbc $(dllname).imp
    $(vi) -s << $(dllname).imp
set magic
set magicstring = ()
atomic
%s/\.dll'//
%s/^(\+\+')(.*)('\.'.*')\.[0-9]+$$/\1\2\3..'\2'/
x
<<
```

A filename may follow a "<<" on a command line to cause a file with that name to be created. (Otherwise, 'WMAKE' chooses a name.) "keep" or "nokeep" may follow a terminating "<<" to show what to do with the file after usage. The default is "nokeep" which zaps it.

10.8 Final Commands (.AFTER)

The `.AFTER` directive specifies commands for Make to run after it has done all other commands. See the section entitled "Command List Directives" on page 137 for a full description of its use.

10.9 Ignoring Dependent Timestamps (.ALWAYS)

The `.ALWAYS` directive indicates to Make that the target should always be updated regardless of the timestamps of its dependents.

```
#
# .always directive
#

foo : bar .always
    wtouch $@
```

foo is updated each time Make is run.

10.10 Automatic Dependency Detection (.AUTODEPEND)

Explicit listing of dependencies in a makefile can often be tedious in the development and maintenance phases of a project. The Open Watcom C/C++ compiler will insert dependency information into the object file as it processes source files so that a complete snapshot of the files necessary to build the object file are recorded. Since all files do not have dependency information contained within them in a standard form, it is necessary to indicate to Make when dependencies are present.

To illustrate the use of the `.AUTODEPEND` directive, we will show its use in an implicit rule and in an explicit rule.

```
#
# .AUTODEPEND example
#
.c.obj: .AUTODEPEND
        wcc386 $[* $(compile_ options)

test.exe : a.obj b.obj c.obj test.res
        wlink FILE a.obj, b.obj, c.obj
        wrc /q /bt=windows test.res test.exe

test.res : test.rc test.ico .AUTODEPEND
        wrc /ad /q /bt=windows /r $[@ $^@
```

In the above example, Make will use the contents of the object file to determine whether the object file has to be built during processing. The Open Watcom Resource Compiler can also insert dependency information into a resource file that can be used by Make.

10.11 Initial Commands (.BEFORE)

The `.BEFORE` directive specifies commands for Make to run before it does any other command. See the section entitled "Command List Directives" on page 137 for a full description of its use.

10.12 Disable Implicit Rules (.BLOCK)

The `.BLOCK` directive and the "b" command line option are alternative controls to cause implicit rules to be ignored. See the section entitled "Command Line Options" on page 86 for a full description of its use.

10.13 Ignoring Errors (.CONTINUE)

The `.CONTINUE` directive and the "b" command line option are alternative controls to cause failing commands to be ignored. See the section entitled "Command Line Options" on page 86 for a full description of its use.

```
#
# .continue example
#

.continue

all: bad good
    @%null

bad:
    false

good:
    touch $@
```

Although the command list for bad fails, that for good is done. Without the directive, good is not built.

10.14 Default Command List (.DEFAULT)

The .DEFAULT directive provides a default command list for those targets which lack one. See the section entitled "Command List Directives" on page 137 for a full description of its use.

```
#
# .default example
#

.default
    @echo Using default rule to update target "$@"
    @echo because of dependent(s) "$<"
    wtouch $@

all: foo

foo:
    wtouch foo
```

"all" has no command list. The one supplied to the default directive is executed instead.

10.15 Erasing Targets After Error (.ERASE)

Most operating system utilities and programs have special return codes that indicate error conditions. Open Watcom Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. By default, Make will prompt for deletion of the current target. The .ERASE directive indicates to Make that the target should be deleted if an error occurs during the execution of the associated command list. No prompt is issued in this case. Here is an example of the .ERASE directive:

```
#
# .ERASE example
#
.ERASE
balance.lst : ledger.dat sales.dat purchase.dat
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will attempt to delete "BALANCE.LST".

10.16 Error Action (.ERROR)

The `.ERROR` directive supplies a command list for error conditions. See the section entitled "Command List Directives" on page 137 for a full description of its use.

```
#
# .error example
#

.error:
        @echo it is good that "$@" is known

all : .symbolic
        false
```

10.17 Ignoring Target Timestamp (.EXISTSONLY)

The `.EXISTSONLY` directive indicates to Make that the target should not be updated if it already exists, regardless of its timestamp.

```
#
# .existsonly directive
#

foo: .existsonly
        wtouch $@
```

If absent, this file creates foo; if present, this file does nothing.

10.18 Specifying Explicitly Updated Targets (.EXPLICIT)

The `.EXPLICIT` directive may me used to specify a target that needs to be explicitly updated. Normally, the first target in a makefule will be implicitly updated if no target is specified on Make command line. The `.EXPLICIT` directive prevents this, and is useful for instance when creating files designed to be included for other make files.

```
#
# .EXPLICIT example
#
target : .symbolic .explicit
        @echo updating first target

next : .symbolic
        @echo updating next target
```

In the above example, Make will not automatically update "target", despite the fact that it is the first one listed.

10.19 Defining Recognized File Extensions (.EXTENSIONS)

The .EXTENSIONS directive and its synonym, the .SUFFIXES directive declare which extensions are allowed to be used in implicit rules and how these extensions are ordered. .EXTENSIONS is the traditional Watcom name; .SUFFIXES is the corresponding POSIX name. The default .EXTENSIONS declaration is:

```
.EXTENSIONS:
.EXTENSIONS: .exe .nlm .dsk .lan .exp .lib .obj &
             .i .asm .c .cpp .cxx .cc .for .pas .cob &
             .h .hpp .hxx .hh .fi .mif .inc
```

A .EXTENSIONS directive with an empty list will clear the .EXTENSIONS list and any previously defined implicit rules. Any subsequent .EXTENSIONS directives will add extensions to the end of the list.

> ***Hint:*** The default .EXTENSIONS declaration could have been coded as:
>
> .EXTENSIONS:
> .EXTENSIONS: .exe
> .EXTENSIONS: .nlm .dsk .lan .exp
> .EXTENSIONS: .lib
> .EXTENSIONS: .obj
> .EXTENSIONS: .i .asm .c .cpp .cxx .cc
> .EXTENSIONS: .for .pas .cob
> .EXTENSIONS: .h .hpp .hxx .hh .fi .mif .inc
> .EXTENSIONS: .inc
>
> with identical results.

Make will not allow any implicit rule declarations that use extensions that are not in the current .EXTENSIONS list.

```
#
# .extensions and .suffixes directives
#

.suffixes : # Clear list
.extensions : .foo .bar

.bar.foo:
    copy $< $@

fubar.foo:

fubar.bar: .existsonly
    wtouch $@
```

The first time this example runs, Make creates fubar.foo. This example always ensures that fubar.foo is a copy of fubar.bar. Note the implicit connection beween the two files.

10.20 Approximate Timestamp Matching (.FUZZY)

The .FUZZY directive allows .AUTODEPEND times to be out by a minute without considering a target out of date. It is only useful in conjunction with the .JUST_ ENOUGH directive when Make is calculating the timestamp to set the target to.

10.21 Preserving Targets After Error (.HOLD)

Most operating system utilities and programs have special return codes that indicate error conditions. Open Watcom Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. By default, Make will prompt for deletion of the current target. The .HOLD directive indicates to Make that the target should not be deleted if an error occurs during the execution of the associated command list. No prompt is issued in this case. The .HOLD directive is similar to .PRECIOUS but applies to all targets listed in the makefile. Here is an example of the .HOLD directive:

```
#
# .HOLD example
#
.HOLD
balance.lst : ledger.dat sales.dat purchase.dat
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will not delete "BALANCE.LST".

10.22 Ignoring Return Codes (.IGNORE)

Some programs do not have meaningful return codes so for these programs we want to ignore the return code completely. There are different ways to ignore return codes namely,

1. use the command line option "i"
2. put a "–" in front of specific commands, or
3. use the .IGNORE directive.

In the following example, the rule:

```
#
# ignore return code example
#
balance.lst : ledger.dat sales.dat purchase.dat
        -doreport
```

will ignore the return status from the program "DOREPORT". Using the dash in front of the command is the preferred method for ignoring return codes because it allows Make to check all the other return codes.

The .IGNORE directive is used as follows:

```
#
# .IGNORE example
#
.IGNORE
balance.lst : ledger.dat sales.dat purchase.dat
        doreport
```

Using the .IGNORE directive will cause Make to ignore the return code for every command. The "i" command line option and the .IGNORE directive prohibit Make from performing any error checking on the commands executed and, as such, should be used with caution.

Another way to handle non-zero return codes is to continue processing targets which do not depend on the target that had a non-zero return code during execution of its associated command list. There are two ways of indicating to Make that processing should continue after a non-zero return code:

1. use the command line option "k"
2. use the .CONTINUE directive.

10.23 Minimising Target Timestamp (.JUST_ENOUGH)

The .JUST_ ENOUGH directive is equivalent to the "j" command line option. The timestamps of created targets are set to be the same as those of their youngest dependendents.

```
#
# .JUST_ ENOUGH example
#

.just_ enough

.c.exe:
        wcl386 -zq $<

hello.exe:
```

hello.exe is given the same timestamp as hello.c, and not the usual timestamp corresponding to when hello.exe was built.

10.24 Updating Targets Multiple Times (.MULTIPLE)

The .MULTIPLE directive is used to update a target multiple times. Normally, Make will only update each target once while processing a makefile. The .MULTIPLE directive is useful if a target needs to be updated more than once, for instance in case the target is destroyed during processing of other targets. Consider the following example:

```
#
# example not using .multiple
#

all: targ1 targ2

target:
        wtouch target

targ1: target
        rm target
        wtouch targ1

targ2: target
        rm target
        wtouch targ2
```

This makefile will fail because "target" is destroyed when updating "targ1", and later is implicitly expected to exist when updating "targ2". Using the `.MULTIPLE` directive will work around this problem:

```
#
# .MULTIPLE example
#

all : targ1 targ2

target : .multiple
        wtouch target

targ1 : target
        rm target
        wtouch targ1

targ2 : target
        rm target
        wtouch targ2
```

Now Make will attempt to update "target" again when updating "targ2", discover that "target" doesn't exist, and recreate it.

10.25 Ignoring Target Timestamp (.NOCHECK)

The `.NOCHECK` directive is used to disable target existence checks in a makefile. See the section entitled "Command Line Options" on page 86 for a full description of its use.

10.26 Cache Search Path (.OPTIMIZE)

The `.OPTIMIZE` directive and the equivalent "o" command line option cause Make to use a circular path search. If a file is found in a particular directory, that directory will be the first searched for the next file. See the section entitled "Command Line Options" on page 86 for a full description of its use.

10.27 Preserving Targets (.PRECIOUS)

Most operating system utilities and programs have special return codes that indicate error conditions. Open Watcom Make will check the return code for every command executed. If the return code is non-zero, Make will stop processing the current rule and optionally delete the current target being updated. If a file is precious enough that this treatment of return codes is not wanted then the `.PRECIOUS` directive may be used. The `.PRECIOUS` directive indicates to Make that the target should not be deleted if an error occurs during the execution of the associated command list. Here is an example of the `.PRECIOUS` directive:

```
#
# .PRECIOUS example
#
balance summary : sales.dat purchase.dat .PRECIOUS
        doreport
```

If the program "DOREPORT" executes and its return code is non-zero then Make will not attempt to delete "BALANCE" or "SUMMARY". If only one of the files is precious then the makefile could be coded as follows:

```
#
# .PRECIOUS example
#
balance : .PRECIOUS
balance summary : sales.dat purchase.dat
        doreport
```

The file "BALANCE.LST" will not be deleted if an error occurs while the program "DOREPORT" is executing.

10.28 Name Command Sequence (.PROCEDURE)

The `.PROCEDURE` directive may be used to construct "procedures" in a makefile.

```
#
# .procedure example
#

all: .symbolic
    @%make proc

proc: .procedure
    @echo Executing procedure "proc"
```

10.29 Re-Checking Target Timestamp (.RECHECK)

Make will re-check the target's timestamp, rather than assuming it was updated by its command list. This is useful if the target is built by another make- style tool, as in the following example:

```
#
# .RECHECK example
#
foo.gz : foo
        gzip foo

foo : .ALWAYS .RECHECK
        nant -buildfile:foo.build
```

foo's command list will always be run, but foo will only be compressed if the timestamp is actually changed.

10.30 Suppressing Terminal Output (.SILENT)

As commands are executed, Open Watcom Make will print out the current command before it is executed. It is possible to execute the makefile without having the commands printed. There are three ways to inhibit the printing of the commands before they are executed, namely:

1. use the command line option "s"
2. put an "@" in front of specific commands, or
3. use the `.SILENT` directive.

In the following example, the rule:

```
#
# silent command example
#
balance summary : ledger.dat sales.dat purchase.dat
        @doreport
```

will prevent the string "doreport" from being printed on the screen before the command is executed.

The `.SILENT` directive is used as follows:

```
#
# .SILENT example
#
.SILENT
balance summary : ledger.dat sales.dat purchase.dat
        doreport
```

Using the `.SILENT` directive or the "s" command line option will inhibit the printing of all commands before they are executed. The "sn" command line option can be used to veto any silencing control.

At this point, most of the capability of Make may be realized. Methods for making makefiles more succinct will be discussed.

10.31 Defining Recognized File Extensions (.SUFFIXES)

The `.SUFFIXES` directive declares which extensions are allowed to be used in implicit rules and how these extensions are ordered. It is a synonym for the `.EXTENSIONS` directive. See the section entitled "Defining Recognized File Extensions (.EXTENSIONS)" on page 102 for a full description of both directives.

10.32 Targets Without Any Dependents (.SYMBOLIC)

There must always be at least one target in a rule but it is not necessary to have any dependents. If a target does not have any dependents, the command list associated with the rule will always be executed if the target is updated.

You might ask, "What may a rule with no dependents be used for?". A rule with no dependents may be used to describe actions that are useful for the group of files being maintained. Possible uses include backing up files, cleaning up files, or printing files.

To illustrate the use of the `.SYMBOLIC` directive, we will add two new rules to the previous example. First, we will omit the `.SYMBOLIC` directive and observe what will happen when it is not present.

```
#
# rule 4: backup the data files
#
backup :
        echo "insert backup disk"
        pause
        copy *.dat a:
        echo "backup complete"
#
# rule 5: cleanup temporary files
#
cleanup :
        del *.tmp
        del \tmp\*.*
```

and then execute the command:

```
wmake backup
```

Make will execute the command list associated with the "backup" target and issue an error message indicating that the file "BACKUP" does not exist after the command list was executed. The same thing would happen if we typed:

```
wmake cleanup
```

In this makefile we are using "backup" and "cleanup" to represent actions we want performed. The names are not real files but rather they are symbolic names. This special type of target may be declared with the `.SYMBOLIC` directive. This time, we show rules 4 and 5 with the appropriate addition of `.SYMBOLIC` directives.

```
#
# rule 4: backup the data files
#
backup : .SYMBOLIC
        echo "insert backup disk"
        pause
        copy *.dat a:
        echo "backup complete"
#
# rule 5: cleanup temporary files
#
cleanup : .SYMBOLIC
        del *.tmp
        del \tmp\*.*
```

The use of the `.SYMBOLIC` directive indicates to Make that the target should always be updated internally after the command list associated with the rule has been executed. A short form for the common idiom of singular `.SYMBOLIC` targets like:

```
target : .SYMBOLIC
        commands
```

is:

```
target
        commands
```

This kind of target definition is useful for many types of management tasks that can be described in a makefile.

10.33 Macros

Open Watcom Make has a simple macro facility that may be used to improve makefiles by making them easier to read and maintain. A macro identifier may be composed from a string of alphabetic characters and numeric characters. The underscore character is also allowed in a macro identifier. If the macro identifier starts with a "%" character, the macro identifier represents an environment variable. For instance, the macro identifier "%path" represents the environment variable "path".

Macro identifiers	*Valid?*
2morrow	yes
stitch_in_9	yes
invalid~id	no
2b_or_not_2b	yes
%path	yes
reports	yes
!@#*%	no

We will use a programming example to show how macros are used. The programming example involves four C/C++ source files and two header files. Here is the initial makefile (before macros):

```
#
# programming example
# (before macros)
#
plot.exe : main.obj input.obj calc.obj output.obj
        wlink @plot

main.obj : main.c defs.h globals.h
        wcc386 main /mf /d1 /w3

calc.obj : calc.c defs.h globals.h
        wcc386 calc /mf /d1 /w3

input.obj : input.c defs.h globals.h
        wcc386 input /mf /d1 /w3

output.obj : output.c defs.h globals.h
        wcc386 output /mf /d1 /w3
```

Macros become useful when changes must be made to makefiles. If the programmer wanted to change the compiler options for the different compiles, the programmer would have to make a global change to the makefile. With this simple example, it is quite easy to make the change but try to imagine a more complex example with different programs having similar options. The global change made by the editor could cause problems by changing the options for other programs. A good habit to develop is to define macros for any programs that have command line options. In our example, we would change the makefile to be:

```
#
# programming example
# (after macros)
#
link_ options =
compiler = wcc386
compile_ options = /mf /d1 /w3

plot.exe : main.obj input.obj calc.obj output.obj
        wlink $(link_ options) @plot

main.obj : main.c defs.h globals.h
        $(compiler) main $(compile_ options)

calc.obj : calc.c defs.h globals.h
        $(compiler) calc $(compile_ options)

input.obj : input.c defs.h globals.h
        $(compiler) input $(compile_ options)

output.obj : output.c defs.h globals.h
        $(compiler) output $(compile_ options)
```

A macro definition consists of a macro identifier starting on the beginning of the line followed by an "=" which in turn is followed by the text to be replaced. A macro may be redefined, with the latest declaration being used for subsequent expansions (no warning is given upon redefinition of a macro). The replacement text may contain macro references.

A macro reference may occur in two forms. The previous example illustrates one way to reference macros whereby the macro identifier is delimited by "$(" and ")". The parentheses are optional so the macros "compiler" and "compile_options" could be referenced by:

```
main.obj : main.c defs.h globals.h
        $compiler main $compile_ options
```

Certain ambiguities may arise with this form of macro reference. For instance, examine this makefile fragment:

Example:

```
temporary_ dir = \tmp\
temporary_ file = $temporary_ dirtmp000.tmp
```

The intention of the declarations is to have a macro that will expand into a file specification for a temporary file. Make will collect the largest identifier possible before macro expansion occurs. The macro reference is followed by text that looks like part of the macro identifier ("tmp000") so the macro identifier that will be referenced will be "temporary_dirtmp000". The incorrect macro identifier will not be defined so an error message will be issued.

If the makefile fragment was:

```
temporary_ dir = \tmp\
temporary_ file = $(temporary_ dir)tmp000.tmp
```

there would be no ambiguity. The preferred way to reference macros is to enclose the macro identifier by "$(" and ")".

Macro references are expanded immediately on dependency lines (and thus may not contain references to macros that have not been defined) but other macro references have their expansion deferred until they are used in a command. In the previous example, the macros "link_options", "compiler", and "compile_options" will not be expanded until the commands that reference them are executed.

Another use for macros is to replace large amounts of text with a much smaller macro reference. In our example, we only have two header files but suppose we had very many header files. Each explicit rule would be very large and difficult to read and maintain. We will use the previous example makefile to illustrate this use of macros.

```
#
# programming example
# (with more macros)
#
link_ options =
compiler = wcc386
compile_ options = /mf /d1 /w3

header_ files = defs.h globals.h
object_ files = main.obj input.obj calc.obj &
                output.obj

plot.exe : $(object_ files)
        wlink $(link_ options) @plot

main.obj : main.c $(header_ files)
        $(compiler) main $(compile_ options)
```

```
calc.obj : calc.c $(header_ files)
        $(compiler) calc $(compile_ options)

input.obj : input.c $(header_ files)
        $(compiler) input $(compile_ options)

output.obj : output.c $(header_ files)
        $(compiler) output $(compile_ options)
```

Notice the ampersand ("&") at the end of the macro definition for "object_files". The ampersand indicates that the macro definition continues on the next line. In general, if you want to continue a line in a makefile, use an ampersand ("&") at the end of the line.

There are special macros provided by Make to access environment variable names. To access the **PATH** environment variable in a makefile, we use the macro identifier "%path". For example, if we have the following line in a command list:

Example:

```
        echo $(%path)
```

it will print out the current value of the **PATH** environment variable when it is executed.

There are two other special environment macros that are predefined by Make. The macro identifier "%cdrive" will expand into one letter representing the current drive. Note that it is operating system dependent whether the cd command changes the current drive. The macro identifier "%cwd" will expand into the current working directory. These macro identifiers are not very useful unless we can specify that they be expanded immediately. The complementary macros "$+" and "$–" respectively turn on and turn off immediate expansion of macros. The scope of the "$+" macro is the current line after which the default macro expansion behaviour is resumed. A possible use of these macros is illustrated by the following example makefile.

```
#
# $(%cdrive), $(%cwd), $+, and $- example
#
dir1 = $(%cdrive):$(%cwd)
dir2 = $+ $(dir1) $-
example : .SYMBOLIC
        cd ..
        echo $(dir1)
        echo $(dir2)
```

Which would produce the following output if the current working directory is C:\WATCOM\SOURCE\EXAMPLE:

Example:

```
(command output only)
C:\WATCOM\SOURCE
C:\WATCOM\SOURCE\EXAMPLE
```

The macro definition for "dir2" forces immediate expansion of the "%cdrive" and "%cwd" macros thus defining "dir2" to be the current directory that Make was invoked in. The macro "dir1" is not expanded until execution time when the current directory has changed from the initial directory.

Combining the $+ and $– special macros with the special macro identifiers "%cdrive" and "%cwd" is a useful makefile technique. The $+ and $– special macros are general enough to be used in many different ways.

Constructing other macros is another use for the $+ and $− special macros. Make allows macros to be redefined and combining this with the $+ and $− special macros, similar looking macros may be constructed.

```
#
# macro construction with $+ and $-
#
template = file1.$(ext) file2.$(ext) file3.$(ext) file4.$(ext)
ext = dat
data_ files = $+ $(template) $-
ext = lst
listing_ files = $+ $(template) $-

example : .SYMBOLIC
        echo $(data_ files)
        echo $(listing_ files)
```

This makefile would produce the following output:

Example:

```
file1.dat file2.dat file3.dat file4.dat
file1.lst file2.lst file3.lst file4.lst
```

Adding more text to a macro can also be done with the $+ and $− special macros.

```
#
# macro addition with $+ and $-
#
objs = file1.obj file2.obj file3.obj
objs = $+$(objs)$- file4.obj
objs = $+$(objs)$- file5.obj

example : .SYMBOLIC
        echo $(objs)
```

This makefile would produce the following output:

Example:

```
file1.obj file2.obj file3.obj file4.obj file5.obj
```

Make provides a shorthand notation for this type of macro operation. Text can be added to a macro by using the "+=" macro assignment. The previous makefile can be written as:

```
#
# macro addition with +=
#
objs  = file1.obj file2.obj file3.obj
objs += file4.obj
objs += file5.obj

example : .SYMBOLIC
        echo $(objs)
```

and still produce the same results. The shorthand notation "+=" supported by Make provides a quick way to add more text to macros.

Make provides the "!inject" preprocessor directive to append a "word" (one or more graphic characters) to one or more macros. The previous makefile is adapted to show the usage:

```
#
# macro construction with !inject
#
!inject file1.obj objs objs12 objs13 objs14 objs15
!inject file2.obj objs objs12 objs13 objs14 objs15
!inject file3.obj objs        objs13 objs14 objs15
!inject file4.obj objs               objs14 objs15
!inject file5.obj objs                      objs15

example : .SYMBOLIC
        echo $(objs)
        echo $(objs12)
        echo $(objs13)
        echo $(objs14)
        echo $(objs15)
```

This makefile would produce the following output:

Example:

```
file1.obj file2.obj file3.obj file4.obj file5.obj
file1.obj file2.obj
file1.obj file2.obj file3.obj
file1.obj file2.obj file3.obj file4.obj
file1.obj file2.obj file3.obj file4.obj file5.obj
```

The "!inject" preprocessor directive supported by Make provides a way to append a word to several macros.

There are instances when it is useful to have macro identifiers that have macro references contained in them. If you wanted to print out an informative message before linking the executable that was different between the debugging and production version, we would express it as follows:

```
#
# programming example
#  (macro selection)
#
version = debugging           # debugging version

msg_ production = linking production version ...
msg_ debugging = linking debug version ...

link_ options_ production =
link_ options_ debugging = debug all
link_ options = $(link_ options_ $(version))

compiler = wcc386
compile_ options_ production = /mf /w3
compile_ options_ debugging = /mf /d1 /w3
compile_ options = $(compile_ options_ $(version))
```

```
header_ files = defs.h globals.h
object_ files = main.obj input.obj calc.obj &
                output.obj

plot.exe : $(object_ files)
        echo $(msg_ $(version))
        wlink $(link_ options) @plot

main.obj : main.c $(header_ files)
        $(compiler) main $(compile_ options)

calc.obj : calc.c $(header_ files)
        $(compiler) calc $(compile_ options)

input.obj : input.c $(header_ files)
        $(compiler) input $(compile_ options)

output.obj : output.c $(header_ files)
        $(compiler) output $(compile_ options)
```

Take notice of the macro references that are of the form:

```
$(<partial_ macro_ identifier>$(version))
```

The expansion of a macro reference begins by expanding any macros seen until a matching right parenthesis is found. The macro identifier that is present after the matching parenthesis is found will be expanded. The other form of macro reference namely:

```
$<macro_ identifier>
```

may be used in a similar fashion. The previous example would be of the form:

```
$<partial_ macro_ identifier>$version
```

Macro expansion occurs until a character that cannot be in a macro identifier is found (on the same line as the "$") after which the resultant macro identifier is expanded. If you want two macros to be concatenated then the line would have to be coded:

```
$(macro1)$(macro2)
```

The use of parentheses is the preferred method for macro references because it completely specifies the order of expansion.

In the previous example, we can see that the four command lines that invoke the compiler are very similar in form. We may make use of these similarities by denoting the command by a macro reference. We need to be able to define a macro that will expand into the correct command when processed. Fortunately, Make can reference the first member of the dependent list, the last member of the dependent list, and the current target being updated with the use of some special macros. These special macros have the form:

```
$<file_ specifier><form_ qualifier>
```

where <file_specifier> is one of:

"^" represents the current target being updated

"[" represents the first member of the dependent list

"]" represents the last member of the dependent list

and <form_qualifier> is one of:

"@" full file name

"*" file name with extension removed

"&" file name with path and extension removed

"." file name with path removed

":" path of file name

If the file "D:\DIR1\DIR2\NAME.EXT" is the current target being updated then the following example will show how the form qualifiers are used.

Macro	*Expansion for D:\DIR1\DIR2\NAME.EXT*
$^@	D:\DIR1\DIR2\NAME.EXT
$^*	D:\DIR1\DIR2\NAME
$^&	NAME
$^.	NAME.EXT
$^:	D:\DIR1\DIR2\

These special macros provide the capability to reference targets and dependents in a variety of ways.

```
#
# programming example
#  (more macros)
#
version = debugging           # debugging version

msg_ production = linking production version ...
msg_ debugging = linking debug version ...

link_ options_ production =
link_ options_ debugging = debug all
link_ options = $(link_ options_ $(version))
```

```
compile_ options_ production = /mf /w3
compile_ options_ debugging = /mf /d1 /w3
compile_ options = $(compile_ options_ $(version))

compiler_ command = wcc386 $[* $(compile_ options)

header_ files = defs.h globals.h
object_ files = main.obj input.obj calc.obj &
                output.obj

plot.exe : $(object_ files)
        echo $(msg_ $(version))
        wlink $(link_ options) @$^*

main.obj : main.c $(header_ files)
        $(compiler_ command)

calc.obj : calc.c $(header_ files)
        $(compiler_ command)

input.obj : input.c $(header_ files)
        $(compiler_ command)

output.obj : output.c $(header_ files)
        $(compiler_ command)
```

This example illustrates the use of the special dependency macros. Notice the use of "$^*" in the linker command. The macro expands into the string "plot" since "plot.exe" is the target when the command is processed. The use of the special dependency macros is recommended because they make use of information that is already contained in the dependency rule.

At this point, we know that macro references begin with a "$" and that comments begin with a "#". What happens if we want to use these characters without their special meaning? Make has two special macros that provide these characters to you. The special macro "$$" will result in a "$" when expanded and "$#" will expand into a "#". These special macros are provided so that you are not forced to work around the special meanings of the "$" and "#" characters.

There is also a simple macro text substitution facility. We have previously seen that a macro call can be made with $(macroname). The construct $(macroname:string1=string2) substitutes macroname with each occurrence of string1 replaced by string2. We have already seen that it can be useful for a macro to be a set of object file names separated by spaces. The file directive in wlink can accept a set of names separated by commas.

```
#
# programming example
#  (macro substitution)
#

.c.obj:
        wcc386 -zq $*.c

object_ files = main.obj input.obj calc.obj output.obj

plot.exe : $(object_ files)
        wlink name $@ file $(object_ files: =,)
```

Note that macro substitution cannot be used with special macros.

It is also worth noting that although the above example shows a valid approach, the same problem, that is, providing a list of object files to wlink, can be solved without macro subsitutions. The solution is using the {} syntax of wlink, as shown in the following example. Refer to the Open Watcom Linker Guide for details.

```
#
# programming example
# (not using macro substitution)
#

.c.obj:
        wcc386 -zq $*.c

object_ files = main.obj input.obj calc.obj output.obj

plot.exe : $(object_ files)
        wlink name $@ file { $(object_ files) }
```

10.34 Implicit Rules

Open Watcom Make is capable of accepting declarations of commonly used dependencies. These declarations are called "implicit rules" as opposed to "explicit rules" which were discussed previously. Implicit rules may be applied only in instances where you are able to describe a dependency in terms of file extensions.

> ***Hint:*** Recall that a file extension is the portion of the file name which follows the period. In the file specification:
>
> `c:\dos\ansi.sys`
>
> the file extension is "SYS".

An implicit rule provides a command list for a dependency between files with certain extensions. The form of an implicit rule is as follows:

```
.<dependent_ extension>.<target_ extension>:
        <command_ list>
```

Implicit rules are used if a file has not been declared as a target in any explicit rule or the file has been declared as a target in an explicit rule with no command list. For a given target file, a search is conducted to see if there are any implicit rules defined for the target file's extension in which case Make will then check if the file with the dependent extension in the implicit rule exists. If the file with the dependent extension exists then the command list associated with the implicit rule is executed and processing of the makefile continues.

Other implicit rules for the target extension are searched in a similar fashion. The order in which the dependent extensions are checked becomes important if there is more than one implicit rule declaration for a target extension. If we have the following makefile fragment:

Example:

```
.pas.obj:
        (command list)
.c.obj:
        (command list)
```

an ambiguity arises. If we have a target file "TEST.OBJ" then which do we check for first, "TEST.PAS" or "TEST.C"? Make handles this with the previously described `.EXTENSIONS` directive. Returning to our makefile fragment:

```
.pas.obj:
        (command list)
.c.obj:
        (command list)
```

and our target file "TEST.OBJ", we know that the `.EXTENSIONS` list determines in what order the dependents "TEST.PAS" and "TEST.C" will be tried. If the .EXTENSIONS declaration is:

Example:

```
.EXTENSIONS:
.EXTENSIONS: .exe .obj .asm .pas .c .cpp .for .cob
```

we can see that the dependent file "TEST.PAS" will be tried first as a possible dependent with "TEST.C" being tried next.

One apparent problem with implicit rules and their associated command lists is that they are used for many different targets and dependents during the processing of a makefile. The same problem occurs with commands constructed from macros. Recall that there is a set of special macros that start with "$^", "$[", or "$]" that reference the target, first dependent, or last dependent of an explicit dependency rule. In an implicit rule there may be only one dependent or many dependents depending on whether the rule is being executed for a target with a single colon ":" or double colon "::" dependency. If the target has a single colon or double colon dependency, the "$^", "$[", and "$]" special macros will reflect the values in the rule that caused the implicit rule to be invoked. Otherwise, if the target does not have a dependency rule then the "$[" and "$]" special macros will be set to the same value, namely, the file found in the implicit rule search.

We will use the last programming example to illustrate a possible use of implicit rules.

```
#
# programming example
# (implicit rules)
#
version = debugging           # debugging version

msg_ production = linking production version ...
msg_ debugging = linking debug version ...

link_ options_ production =
link_ options_ debugging = debug all
link_ options = $(link_ options_ $(version))
```

```
compiler = wcc386
compile_ options_ production = /mf /w3
compile_ options_ debugging = /mf /d1 /w3
compile_ options = $(compile_ options_ $(version))

header_ files = defs.h globals.h
object_ files = main.obj input.obj calc.obj &
                output.obj

plot.exe : $(object_ files)
        echo $(msg_ $(version))
        wlink $(link_ options) @$^*

.c.obj:
        $(compiler) $[* $(compile_ options)

main.obj : main.c $(header_ files)

calc.obj : calc.c $(header_ files)

input.obj : input.c $(header_ files)

output.obj : output.c $(header_ files)
```

As this makefile is processed, any time an object file is found to be older than its associated source file or header files then Make will attempt to execute the command list associated with the explicit rule. Since there are no command lists associated with the four object file targets, an implicit rule search is conducted. Suppose "CALC.OBJ" was older than "CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".c.obj" implicit rule to be invoked for "CALC.OBJ". The file "CALC.C" is found to exist so the commands

```
wcc386 calc /mf /d1 /w3
echo linking debug version ...
wlink debug all @plot
```

are executed. The last two commands are a result of the compilation of "CALC.C" producing a "CALC.OBJ" file that is younger than the "PLOT.EXE" file that in turn must be generated again.

The use of implicit rules is straightforward when all the files that the makefile deals with are in the current directory. Larger applications may have files that are in many different directories. Suppose we moved the programming example files to three sub-directories.

Files	***Sub-directory***
***.H**	\EXAMPLE\H
***.C**	\EXAMPLE\C
rest	\EXAMPLE\O

Now the previous makefile (located in the \EXAMPLE\O sub-directory) would look like this:

```
#
# programming example
# (implicit rules)
#
h_dir   = \example\h\ #sub-directory containing header files
c_dir   = \example\c\ #sub-directory containing C/C++ files
version = debugging   # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

link_options_production =
link_options_debugging = debug all
link_options = $(link_options_$(version))

compiler = wcc386
compile_options_production = /mf /w3
compile_options_debugging = /mf /d1 /w3
compile_options = $(compile_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.c.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(c_dir)main.c $(header_files)

calc.obj : $(c_dir)calc.c $(header_files)

input.obj : $(c_dir)input.c $(header_files)

output.obj : $(c_dir)output.c $(header_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" was older than "\EXAMPLE\C\CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".c.obj" implicit rule to be invoked for "CALC.OBJ". At this time, the file "\EXAMPLE\O\CALC.C" is not found so an error is reported indicating that "CALC.OBJ" could not be updated. How may implicit rules be useful in larger applications if they will only search the current directory for the dependent file? We must specify more information about the dependent extension (in this case ".C"). We do this by associating a path with the dependent extension as follows:

```
.<dependent_extension> : <path_specification>
```

This allows the implicit rule search to find the files with the dependent extension.

> ***Hint:*** A valid path specification is made up of directory specifications separated by semicolons (";"). Here are some path specifications:
>
> ```
> D:;C:\DOS;C:\UTILS;C:\WC
> C:\SYS
> A:\BIN;D:
> ```
>
> Notice that these path specifications are identical to the form required by the operating system shell's "PATH" command.

Our makefile will be correct now if we add the new declaration as follows:

```
#
# programming example
# (implicit rules)
#
h_dir   = \example\h\ #sub-directory containing header files
c_dir   = \example\c\ #sub-directory containing C/C++ files
version = debugging    # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

link_options_production =
link_options_debugging = debug all
link_options = $(link_options_$(version))

compiler = wcc386
compile_options_production = /mf /w3
compile_options_debugging = /mf /d1 /w3
compile_options = $(compile_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.c:     $(c_dir)
.c.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(c_dir)main.c $(header_files)

calc.obj : $(c_dir)calc.c $(header_files)

input.obj : $(c_dir)input.c $(header_files)

output.obj : $(c_dir)output.c $(header_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" is older than "\EXAMPLE\C\CALC.C". The lack of a command list in the explicit rule with "CALC.OBJ" as a target will cause the ".c.obj" implicit rule to be invoked for "CALC.OBJ". The dependent extension ".C" has a path associated with it so the file "\EXAMPLE\C\CALC.C" is found to exist. The commands

```
wcc386 \EXAMPLE\C\CALC /mf /d1 /w3
echo linking debug version ...
wlink debug all @plot
```

are executed to update the necessary files.

If the application requires many source files in different directories Make will search for the files using their associated path specifications. For instance, if the current example files were setup as follows:

Sub-directory Contents

\EXAMPLE\H
DEFS.H, GLOBALS.H

\EXAMPLE\C\PROGRAM
MAIN.C, CALC.C

\EXAMPLE\C\SCREEN

```
INPUT.C, OUTPUT.C
```

\EXAMPLE\O

```
PLOT.EXE, MAKEFILE, MAIN.OBJ, CALC.OBJ, INPUT.OBJ, OUTPUT.OBJ
```

the makefile would be changed to:

```
#
# programming example
# (implicit rules)
#
h_dir        = ..\h\    # sub-directory with header files
                        # sub-directories with C/C++ source files
program_dir  = ..\c\program\ # - MAIN.C, CALC.C
screen_dir   = ..\c\screen\  # - INPUT.C, OUTPUT.C
version      = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

link_options_production =
link_options_debugging = debug all
link_options = $(link_options_$(version))

compiler = wcc386
compile_options_production = /mf /w3
compile_options_debugging = /mf /d1 /w3
compile_options = $(compile_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.c:     $(program_dir);$(screen_dir)
.c.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(program_dir)main.c $(header_files)

calc.obj : $(program_dir)calc.c $(header_files)

input.obj : $(screen_dir)input.c $(header_files)

output.obj : $(screen_dir)output.c $(header_files)
```

Suppose that there is a change in the "DEFS.H" file which causes all the source files to be recompiled. The implicit rule ".c.obj" is invoked for every object file so the corresponding ".C" file must be found for each ".OBJ" file. We will show where Make searches for the C/C++ source files.

```
update    main.obj
  test    ..\c\program\main.c            (it does exist)
  execute wcc386 ..\c\program\main /mf /d1 /w3

update    calc.obj
  test    ..\c\program\calc.c            (it does exist)
  execute wcc386 ..\c\program\calc /mf /d1 /w3
```

```
update    input.obj
  test    ..\c\program\input.c       (it does not exist)
  test    ..\c\screen\input.c            (it does exist)
  execute wcc386 ..\c\screen\input /mf /d1 /w3

update    output.obj
  test    ..\c\program\output.c      (it does not exist)
  test    ..\c\screen\output.c           (it does exist)
  execute wcc386 ..\c\screen\output /mf /d1 /w3

etc.
```

Notice that Make checked the sub-directory "..\C\PROGRAM" for the files "INPUT.C" and "OUTPUT.C". Make optionally may use a circular path specification search which may save on disk activity for large makefiles. The circular path searching may be used in two different ways:

1. use the command line option "o"
2. use the `.OPTIMIZE` directive.

Make will retain (for each suffix) what sub-directory yielded the last successful search for a file. The search for a file is resumed at this directory in the hope that wasted disk activity will be minimized. If the file cannot be found in the sub-directory then Make will search the next sub-directory in the path specification (cycling to the first sub-directory in the path specification after an unsuccessful search in the last sub-directory).

Changing the previous example to include this feature, results in the following:

```
#
# programming example
# (optimized path searching)
#
.OPTIMIZE

h_dir        = ..\h\    # sub-directory with header files
                        # sub-directories with C/C++ source files
program_dir  = ..\c\program\ # - MAIN.C, CALC.C
screen_dir   = ..\c\screen\  # - INPUT.C, OUTPUT.C
version      = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

link_options_production =
link_options_debugging = debug all
link_options = $(link_options_$(version))

compiler = wcc386
compile_options_production = /mf /w3
compile_options_debugging = /mf /d1 /w3
compile_options = $(compile_options_$(version))

header_files = $(h_dir)defs.h $(h_dir)globals.h
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.c:     $(program_dir);$(screen_dir)
.c.obj:
        $(compiler) $[* $(compile_options)
```

```
main.obj : $(program_ dir)main.c $(header_ files)

calc.obj : $(program_ dir)calc.c $(header_ files)

input.obj : $(screen_ dir)input.c $(header_ files)

output.obj : $(screen_ dir)output.c $(header_ files)
```

Suppose again that there is a change in the "DEFS.H" file which causes all the source files to be recompiled. We will show where Make searches for the C/C++ source files using the optimized path specification searching.

```
update    main.obj
  test    ..\c\program\main.c             (it does exist)
  execute wcc386 ..\c\program\main /mf /d1 /w3

update    calc.obj
  test    ..\c\program\calc.c             (it does exist)
  execute wcc386 ..\c\program\calc /mf /d1 /w3

update    input.obj
  test    ..\c\program\input.c       (it does not exist)
  test    ..\c\screen\input.c             (it does exist)
  execute wcc386 ..\c\screen\input /mf /d1 /w3

update    output.obj
  test    ..\c\screen\output.c            (it does exist)
  execute wcc386 ..\c\screen\output /mf /d1 /w3

etc.
```

Make did not check the sub-directory "..\C\PROGRAM" for the file "OUTPUT.C" because the last successful attempt to find a ".C" file occurred in the "..\C\SCREEN" sub-directory. In this small example, the amount of disk activity saved by Make is not substantial but the savings become much more pronounced in larger makefiles.

> ***Hint:*** The simple heuristic method that Make uses for optimizing path specification searches namely, keeping track of the last successful sub-directory, is very effective in reducing the amount of disk activity during the processing of a makefile. A pitfall to avoid is having two files with the same name in the path. The version of the file that is used to update the target depends on the previous searches. Care should be taken when using files that have the same name with path specifications.

Large makefiles for projects written in C/C++ may become difficult to maintain with all the header file dependencies. Ignoring header file dependencies and using implicit rules may reduce the size of the makefile while keeping most of the functionality intact. The previous example may be made smaller by using this idea.

```
#
# programming example
# (no header dependencies)
#
.OPTIMIZE

h_dir        = ..\h\    # sub-directory with header files
                        # sub-directories with C/C++ source files
program_dir  = ..\c\program\ # - MAIN.C, CALC.C
screen_dir   = ..\c\screen\  # - INPUT.C, OUTPUT.C
version      = debugging     # debugging version

msg_production = linking production version ...
msg_debugging = linking debug version ...

link_options_production =
link_options_debugging = debug all
link_options = $(link_options_$(version))

compiler = wcc386
compile_options_production = /mf /w3
compile_options_debugging = /mf /d1 /w3
compile_options = $(compile_options_$(version))

object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.c:     $(program_dir);$(screen_dir)
.c.obj:
        $(compiler) $[* $(compile_options)
```

Implicit rules are very useful in this regard providing you are aware that you have to make up for the information that is missing from the makefile. In the case of C/C++ programs, you must ensure that you force Make to compile any programs affected by changes in header files. Forcing Make to compile programs may be done by touching source files (not recommended), deleting object files, or using the "a" option and targets on the command line. Here is how the files "INPUT.OBJ" and "MAIN.OBJ" may be recompiled if a change in some header file affects both files.

Example:

```
del input.obj
del main.obj
wmake
```

or using the "a" option

Example:

```
wmake /a input.obj main.obj
```

The possibility of introducing bugs into programs is present when using this makefile technique because it does not protect the programmer completely from object modules becoming out-of-date. The use of implicit rules without header file dependencies is a viable makefile technique but it is not without its pitfalls.

10.35 Double Colon Explicit Rules

Single colon ":" explicit rules are useful in many makefile applications. However, the single colon rule has certain restrictions that make it difficult to express more complex dependency relationships. The restrictions imposed on single colon ":" explicit rules are:

1. only one command list is allowed for each target
2. after the command list is executed, the target is considered up to date

The first restriction becomes evident when you want to update a target in different ways (i.e., when the target is out of date with respect to different dependents). The double colon explicit rule removes this restriction.

```
#
# multiple command lists
#
target1 :: dependent1 dependent2
        command1

target1 :: dependent3 dependent4
        command2
```

Notice that if "target1" is out of date with respect to either "dependent1" or "dependent2" then "command1" will be executed. The double colon "::" explicit rule does not consider the target (in this case "target1") up to date after the command list is executed. Make will continue to attempt to update "target1". Afterwards "command2" will be executed if "target1" is out of date with respect to either "dependent3" or "dependent4". It is possible that both "command1" and "command2" will be executed. As a result of the target not being considered up to date, an implicit rule search will be conducted on "target1" also. Make will process the double colon "::" explicit rules in the order that they are encountered in the makefile. A useful application of the double colon "::" explicit rule involves maintaining and using prototype information generated by a compiler.

```
#
# double colon "::" example
#
compiler = wcc386
options = /w3

# generate macros for the .OBJ and .DEF files
template = module1.$(ext) module2.$(ext) module3.$(ext)
ext = obj
objs = $+ $(template) $-
ext = def
defs = $+ $(template) $-

# add .DEF to the extensions list
.EXTENSIONS:
.EXTENSIONS: .exe .obj .def .c
```

```
# implicit rules for the .OBJ and .DEF files
.c.obj:
        $(compiler) $[* $(options)

# generate the prototype file (only do a syntax check)
.c.def:
        $(compiler) $[* $(options) /v/zs

program.exe :: $(defs)
        erase *.err

program.exe :: $(objs)
        wlink @$^*
```

The ".OBJ" files are updated to complete the update of the file "PROGRAM.EXE". It is important to keep in mind that Make does not consider the file "PROGRAM.EXE" up to date until it has conducted a final implicit rule search. The double colon "::" explicit rule is useful when describing complex update actions.

10.36 Preprocessing Directives

One of the primary objectives in using a make utility is to improve the development and maintenance of projects. A programming project consisting of many makefiles in different sub-directories may become unwieldy to maintain. The maintenance problem stems from the amount of duplicated information scattered throughout the project makefiles. Make provides a method to reduce the amount of duplicated information present in makefiles. Preprocessing directives provide the capability for different makefiles to make use of common information.

10.36.1 File Inclusion

A common solution to the "duplicated information" problem involves referencing text contained in one file from many different files. Make supports file inclusion with the `!include` preprocessing directive. The development of object libraries, using 16-bit Open Watcom C/C++, for the different 80x86 16-bit memory models provides an ideal example to illustrate the use of the `!include` preprocessing directive.

Sub-directory ***Contents***

\WINDOW `WINDOW.CMD, WINDOW.MIF`

\WINDOW\H `PROTO.H, GLOBALS.H, BIOS_ DEF.H`

\WINDOW\C `WINDOW.C, KEYBOARD.C, MOUSE.C, BIOS.C`

\WINDOW\SCSD
small model object files, `MAKEFILE, WINDOW_ S.LIB`

\WINDOW\SCBD
compact model object files, `MAKEFILE, WINDOW_ C.LIB`

\WINDOW\BCSD
medium model object files, `MAKEFILE, WINDOW_ M.LIB`

\WINDOW\BCBD
large model object files, MAKEFILE, WINDOW_ L.LIB

\WINDOW\BCHD
huge model object files, MAKEFILE, WINDOW_ L.LIB

The WLIB command file "WINDOW.CMD" contains the list of library operations required to build the libraries. The contents of "WINDOW.CMD" are:

```
-+window
-+bios
-+keyboard
-+mouse
```

The "-+" library manager command indicates to WLIB that the object file should be replaced in the library.

The file "WINDOW.MIF" contains the makefile declarations that are common to every memory model. The ".MIF" extension will be used for all the Make Include Files discussed in this manual. This extension is also in the default extension list so it is a recommended extension for Make include files. The contents of the "WINDOW.MIF" file is as follows:

```
#
# example of a Make Include File
#
common = /d1 /w3          # common options
objs = window.obj bios.obj keyboard.obj mouse.obj

.c: ..\c
.c.obj:
        wcc $[* $(common) $(local) /m$(model)

window_$(model).lib : $(objs)
        wlib window_$(model) @..\window
```

The macros "model" and "local" are defined by the file "MAKEFILE" in each object directory. An example of the file "MAKEFILE" in the medium memory model object directory is:

```
#
# !include example
#
model = m         # memory model required
local =           # memory model specific options
!include ..\window.mif
```

Notice that changes that affect all the memory models may be made in one file, namely "WINDOW.MIF". Any changes that are specific to a memory model may be made to the "MAKEFILE" in the object directory. To update the medium memory model library, the following commands may be executed:

Example:

```
C>cd \window\bcsd
C>wmake
```

A DOS ".BAT" or OS/2 ".CMD" file may be used to update all the different memory models. If the following DOS "MAKEALL.BAT" (OS/2 "MAKEALL.CMD") file is located somewhere in the "PATH", we may update all the libraries.

```
cd \window\scsd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\scbd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bcsd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bcbd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
cd \window\bchd
wmake %1 %2 %3 %4 %5 %6 %7 %8 %9
```

The batch file parameters are useful if you want to specify options to Make. For instance, a global recompile may be done by executing:

Example:

```
C>makeall /a
```

The `!include` preprocessing directive is a good way to partition common information so that it may be maintained easily.

Another use of the `!include` involves program generated makefile information. For instance, if we have a program called "WMKMK" that will search through source files and generate a file called "WMKMK.MIF" that contains:

```
#
# program generated makefile information
#
C_ to_ OBJ = $(compiler) $[* $(compile_ options)

OBJECTS = WINDOW.OBJ BIOS.OBJ KEYBOARD.OBJ MOUSE.OBJ

WINDOW.OBJ : ..\C\WINDOW.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_ to_ OBJ)
BIOS.OBJ : ..\C\BIOS.C ..\H\BIOS_ DEF.H ..\H\GLOBALS.H
    $(C_ to_ OBJ)
KEYBOARD.OBJ : ..\C\KEYBOARD.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_ to_ OBJ)
MOUSE.OBJ : ..\C\MOUSE.C ..\H\PROTO.H ..\H\GLOBALS.H
    $(C_ to_ OBJ)
```

In order to use this program generated makefile information, we use a "MAKEFILE" containing:

```
#
# makefile that makes use of generated makefile information
#
compile_ options = /mf /d1 /w3

first_ target : window.lib .SYMBOLIC
        echo done

!include wmkmk.mif

window.lib : $(OBJECTS)
        wlib window $(OBJECTS)

make : .SYMBOLIC
        wmkmk /r ..\c\*.c+..\c\*.cpp+..\h
```

Notice that there is a symbolic target "first_target" that is used as a "place holder". The default behaviour for Make is to "make" the first target encountered in the makefile. The symbolic target "first_target" ensures that we have control over what file will be updated first (in this case "WINDOW.LIB"). The use of the `!include` preprocessing directive simplifies the use of program generated makefile information because any changes are localized to the file "MAKEFILE". As program development continues, the file "WMKMK.MIF" may be regenerated so that subsequent invocations of WMAKE benefit from the new makefile information. The file "MAKEFILE" even contains the command to regenerate the file "WMKMK.MIF". The symbolic target "make" has an associated command list that will regenerate the file "WMKMK.MIF". The command list can be executed by typing the following command:

Example:

```
C>wmake make
```

The use of the `!include` preprocessing directive is a simple way to reduce maintenance of related makefiles.

> ***Hint:*** Macros are expanded on `!include` preprocessor control lines. This allows many benefits like:
>
> *!include $(%env_var)*
>
> so that the files that Make will process can be controlled through many different avenues like internal macros, command line macros, and environment variables.
>
> Another way to access files is through the suffix path feature of Make. A definition like
>
> *.mif : c:\mymifs;d:\some\more\mifs*
>
> will cause Make to search different paths for any make include files.

10.36.2 Conditional Processing

Open Watcom Make has conditional preprocessing directives available that allow different declarations to be processed. The conditional preprocessing directives allow the makefile to

1. check whether a macro is defined, and
2. check whether a macro has a certain value.

The macros that can be checked include

1. normal macros "$(<macro_identifier>)"
2. environment macros "$(%<environment_var>)"

The conditional preprocessing directives allow a makefile to adapt to different external conditions based on the values of macros or environment variables. We can define macros on the WMAKE command line as shown in the following example.

Example:

```
C>wmake "macro=some text with spaces in it"
```

Alternatively, we can include a makefile that defines the macros if all the macros cannot fit on the command line. This is shown in the following example:

Example:

```
C>wmake /f macdef.mif /f makefile
```

Also, environment variables can be set before WMAKE is invoked. This is shown in the following example:

Example:

```
C>set macro=some text with spaces in it
C>wmake
```

Now that we know how to convey information to Make through either macros or environment variables, we will look at how this information can be used to influence makefile processing.

Make has conditional preprocessing directives that are similar to the C preprocessor directives. Make supports these preprocessor directives:

```
!ifeq
!ifneq
!ifeqi
!ifneqi
!ifdef
!ifndef
```

along with

```
!else
!endif
```

Together these preprocessor directives allow selection of makefile declarations to be based on either the value or the existence of a macro.

Environment variables can be checked by using an environment variable name prefixed with a "%". A common use of a conditional preprocessing directive involves setting environment variables.

```
#
# setting an environment variable
#
!ifndef %lib

.BEFORE
        set lib=c:\watcom\lib386;c:\watcom\lib386\dos
!endif
```

If you are writing portable applications, you might want to have:

```
#
# checking a macro
#
!include version.mif

!ifdef OS2
machine = /2          # compile for 286
!else
machine = /0          # default: 8086
!endif
```

The `!ifdef` ("if defined") and `!ifndef` ("if not defined") conditional preprocessing directives are useful for checking boolean conditions. In other words, the `!ifdef` and `!ifndef` are useful for "yes-no" conditions. There are instances where it would be useful to check a macro against a value. In order to use the value checking preprocessor directives, we must know the exact value of a macro. A macro definition is of the form:

```
<macro_ identifier> = <text> <comment>
```

Make will first strip any comment off the line. The macro definition will then be the text following the equal "=" sign with leading and trailing blanks removed. Initially this might not seem like a sensible way to define a macro but it does lend itself well to defining macros that are common in makefiles. For instance, it allows definitions like:

```
#
# sample macro definitions
#
link_ options    = debug line   # line number debugging
compile_ options = /d1 /s       # line numbers, no stack checking
```

These definitions are both readable and useful. A makefile can handle differences between compilers with the `!ifeq, !ifneq, !ifeqi` and `!ifneqi` conditional preprocessing directives. The first two perform case sensitive comparisons while the last two perform case insensitive comparisons. One way of setting up adaptive makefiles is:

```
#
# options made simple
#
compiler        = wcc386

stack_ overflow  = No   # yes -> check for stack overflow
line_ info       = Yes  # yes -> generate line numbers

!ifeq compiler wcc386
!ifneqi stack_ overflow   yes
stack_ option    =       /s
!endif
!ifeqi line_ info         yes
line_ option     =       /d1
!endif
!endif
```

```
!ifeq compiler tcc
!ifeqi stack_ overflow    yes
stack_ option    =       -N
!endif
!ifeqi line_ info         yes
line_ option     =       -y
!endif
!endif
#
# make sure the macros are defined
#
!ifndef stack_ option
stack_ option    =
!endif
!ifndef line_ option
line_ option    =
!endif

example : .SYMBOLIC
        echo $(compiler) $(stack_ option) $(line_ option)
```

The conditional preprocessing directives can be very useful to hide differences, exploit similarities, and organize declarations for applications that use many different programs.

Another directive is the `!define` directive. This directive is equivalent to the normal type of macro definition (i.e., macro = text) but will make C programmers feel more at home. One important distinction is that the `!define` preprocessor directive may be used to reflect the logical structure of macro definitions in conditional processing. For instance, the previous makefile could have been written in this style:

```
!ifndef stack_ option
!  define stack_ option
!endif
!ifndef line_ option
!  define line_ option
!endif
```

The "!" character must be in the first column but the directive keyword can be indented. This freedom applies to all of the preprocessing directives. The `!else` preprocessing directive benefits from this type of style because `!else` can also check conditions like:

```
!else ifeq
!else ifneq
!else ifeqi
!else ifneqi
!else ifdef
!else ifndef
```

so that logical structures like:

```
!ifdef %version
!  ifeq %version debugging
!    define option debug all
!  else ifeq %version beta
!    define option debug line
!  else ifeq %version production
!    define option debug
!  else
!    error invalid value in VERSION
!  endif
!endif
```

can be used. The above example checks the environment variable "VERSION" for three possible values and acts accordingly.

Another derivative from the C language preprocessor is the `!error` directive which has the form of

```
!error <text>
```

in Make. This directive will print out the text and terminate processing of the makefile. It is very useful in preventing errors from macros that are not defined properly. Here is an example of the `!error` preprocessing directive.

```
!ifndef stack_ option
!  error stack_ option is not defined
!endif
!ifndef line_ option
!  error line_ option is not defined
!endif
```

There is one more directive that can be used in a makefile. The `!undef` preprocessing directive will clear a macro definition. The `!undef` preprocessing directive has the form:

```
!undef <macro_ identifier>
```

The macro identifier can represent a normal macro or an environment variable. A macro can be cleared after it is no longer needed. Clearing a macro will reduce the memory requirements for a makefile. If the macro identifier represents an environment variable (i.e., the identifier has a "%" prefix) then the environment variable will be deleted from the current environment. The `!undef` preprocessing directive is useful for deleting environment variables and reducing the amount of internal memory required during makefile processing.

10.36.3 Loading Dynamic Link Libraries

Open Watcom Make supports loading of Dynamic Link Library (DLL) versions of Open Watcom software through the use of the `!loaddll` preprocessing directive. This support is available on Win32 and 32-bit OS/2 platforms. Performance is greatly improved by avoiding a reload of the software for each file to be processed. The syntax of the `!loaddll` preprocessing directive is:

```
!loaddll $(exename) $(dllname)
```

where `$(exename)` is the command name used in the makefile and `$(dllname)` is the name of the DLL to be loaded and executed in its place. For example, consider the following makefile which contains a list of commands and their corresponding DLL versions.

```
# Default compilation macros for sample programs
#
# Compile switches that are enabled

CFLAGS  = -d1
CC      = wpp386 $(CFLAGS)

LFLAGS  = DEBUG ALL
LINK    = wlink  $(LFLAGS)

!ifdef __LOADDLL__
!  loaddll wcc      wccd
!  loaddll wccaxp   wccdaxp
!  loaddll wcc386   wccd386
!  loaddll wpp      wppdi86
!  loaddll wppaxp   wppdaxp
!  loaddll wpp386   wppd386
!  loaddll wlink    wlinkd
!  loaddll wlib     wlibd
!endif

.c.obj:
        $(CC) $*.c
```

The `__LOADDLL__` symbol is defined for versions of Open Watcom Make that support the `!loaddll` preprocessing directive. The `!ifdef __LOADDLL__` construct ensures that the makefile can be processed by an older version of Open Watcom Make.

Make will look up the `wpp386` command in its DLL load table and find a match. It will then attempt to load the corresponding DLL (i.e., `wppd386.dll`) and pass it the command line for processing. The lookup is case insensitive but must match in all other respects. For example, if a path is included with the command name then the same path must be specified in the `!loaddll` preprocessing directive. This problem can be avoided through the use of macros as illustrated below.

```
# Default compilation macros for sample programs
#
# Compile switches  that are enabled
#
cc286   = wpp
cc286d  = wppdi86
cc386   = wpp386
cc386d  = wppd386
linker  = wlink
linkerd = wlinkd

CFLAGS  = -d1
CC      = $(cc386) $(CFLAGS)

LFLAGS  = DEBUG ALL
LINK    = wlink  $(LFLAGS)

!ifdef __LOADDLL__
!loaddll $(cc286)   $(cc286d)
!loaddll $(cc386)   $(cc386d)
!loaddll $(linker)  $(linkerd)
!endif

.c.obj:
        $(CC) $*.c
```

A path and/or extension may be specified with the DLL name if desired.

10.37 Command List Directives

Open Watcom Make supports special directives that provide command lists for different purposes. If a command list cannot be found while updating a target then the directive `.DEFAULT` may be used to provide one. A simple `.DEFAULT` command list which makes the target appear to be updated is:

```
.DEFAULT
        wtouch $^@
```

The Open Watcom Touch utility sets the time-stamp on the file to the current time. The effect of the above rule will be to "update" the file without altering its contents.

In some applications it is necessary to execute some commands before any other commands are executed and likewise it is useful to be able to execute some commands after all other commands are executed. Make supports this capability by checking to see if the `.BEFORE` and `.AFTER` directives have been used. If the `.BEFORE` directive has been used, the `.BEFORE` command list is executed before any commands are executed. Similarly the `.AFTER` command list is executed after processing is finished. It is important to note that if all the files are up to date and no commands must be executed, the `.BEFORE` and `.AFTER` command lists are never executed. If some commands are executed to update targets and errors are detected (non-zero return status, macro expansion errors), the `.AFTER` command list is not executed (the `.ERROR` directive supplies a command list for error conditions and is discussed in this section). These two directives may be used for maintenance as illustrated in the following example:

```
#
# .BEFORE and .AFTER example
#
.BEFORE
        echo .BEFORE command list executed
.AFTER
        echo .AFTER command list executed
#
# rest of makefile follows
#
        .
        .
        .
```

If all the targets in the makefile are up to date then neither the `.BEFORE` nor the `.AFTER` command lists will be executed. If any of the targets are not up to date then before any commands to update the target are executed, the `.BEFORE` command list will be executed. The `.AFTER` command list will be executed only if there were no errors detected during the updating of the targets. The `.BEFORE, .DEFAULT,` and `.AFTER` command list directives provide the capability to execute commands before, during, and after the makefile processing.

Make also supports the `.ERROR` directive. The `.ERROR` directive supplies a command list to be executed if an error occurs during the updating of a target.

```
#
# .ERROR example
#
.ERROR
        beep
#
# rest of makefile follows
#
        .
        .
        .
```

The above makefile will audibly signal you that an error has occurred during the makefile processing. If any errors occur during the `.ERROR` command list execution, makefile processing is terminated.

10.38 MAKEINIT File

As you become proficient at using Open Watcom Make, you will probably want to isolate common makefile declarations so that there is less duplication among different makefiles. Make will search for a file called "MAKEINIT" (or "TOOLS.INI" when the "ms" option is set) and process it before any other makefiles. The search for the "MAKEINIT" file will occur along the current "PATH". If the file "MAKEINIT" is not found, processing continues without any errors. By default, Make defines a set of data described at the "r" option. The use of a "MAKEINIT" file will allow you to reuse common declarations and will result in simpler, more maintainable makefiles.

10.39 Command List Execution

Open Watcom Make is a program which must execute other programs and operating system shell commands. There are three basic types of executable files in DOS and RDOS.

1. `.COM` files
2. `.EXE` files
3. `.BAT` files

There are two basic types of executable files in Windows NT.

1. `.EXE` files
2. `.BAT` files

There are two basic types of executable files in OS/2.

1. `.EXE` files
2. `.CMD` files

The `.COM` and `.EXE` files may be loaded into memory and executed. The `.BAT` files must be executed by the DOS command processor or shell, "COMMAND.COM". The `.CMD` files must be executed by the OS/2 command processor or shell, "CMD.EXE" Make will search along the "PATH" for the command and depending on the file extension the file will be executed in the proper manner.

If Make detects any input or output redirection characters (these are ">", "<", and "|") in the command, it will be executed by the shell.

Under DOS, an asterisk prefix (*) will cause Make to examine the length of the command argument. If it is too long (> 126 characters), it will take the command argument and stuff it into a temporary environment variable and then execute the command with "@env_var" as its argument. Suppose the following sample makefile fragment contained a very long command line argument.

```
#
# Asterisk example
#
    *foo myfile /a /b /c ... /x /y /z
```

Make will perform something logically similar to the following steps.

```
set TEMPVAR001=myfile /a /b /c ... /x /y /z
foo @TEMPVAR001
```

The command must, of course, support the "@env_var" syntax. Typically, DOS commands do not support this syntax but many of the Open Watcom tools do.

The exclamation mark prefix (!) will force a command to be executed by the shell. Also, the command will be executed by the shell if the command is an internal shell command from the following list:

break	check for Ctrl+Break
call	nest batch files
chdir	change current directory
cd	change current directory

cls	clear the screen
cmd	start NT or OS/2 command processor
command	start DOS command processor
copy	copy or combine files
ctty	DOS redirect input/output to COM port
d:	change drive where "d" represents a drive specifier
date	set system date
del	erase files
dir	display contents in a directory
echo	display commands as they are processed, intercepted by WMAKE
erase	erase files
for	repetitively process commands, intercepted by WMAKE
if	allow conditional processing of commands, intercepted by WMAKE
md	make directory
mkdir	make directory, intercepted by WMAKE
path	set search path
pause	suspend batch operations
prompt	change command prompt
ren	rename files
rename	rename files
rmdir	remove directory, intercepted by WMAKE
rd	remove directory
rm	erase files or directories, intercepted by WMAKE
set	set environment variables, intercepted by WMAKE
time	set system time
type	display contents of a file
ver	display the operating system version number
verify	set data verification
vol	display disk volume label

Below is description of all commands intercepted by Make. Their syntax and functionality is derived from DOS version of these commands. Some of them is not available on DOS OS, these commands syntax and functionality is derived from POSIX standard.

Any of these commands have not the same limitations as on appropriate OS and support features necessary for all supported OSes.

- no length limit for commands
- long file name (LFN) support for file names
- mixed forward and backward slash support for directory and file names

10.39.1 echo command

The operating system shell "echo" command is intercepted by Make. It uses following syntax:

```
echo [<value>]
```

The "echo" command may be used to output any string to standard output without length limitation.

10.39.2 set command

The operating system shell "set" command is intercepted by Make. It uses following syntax:

```
set <name> =[<value>]    (if no value then variable is deleted)
```

The "set" command may be used to set environment variables to values required during makefile processing. The environment variable changes are only valid during makefile processing and do not affect the values that were in effect before Make was invoked. The "set" command may be used to initialize environment variables necessary for the makefile commands to execute properly. The setting of environment variables in makefiles reduces the number of "set" commands required in the system initialization file. Here is an example with the Open Watcom C/C++ compiler.

```
#
# set example
#
.BEFORE
        set include=c:\special\h;$(%include)
        set lib=c:\watcom\lib386;c:\watcom\lib386\dos
#
# rest of makefile follows
#
        .
        .
        .
```

The first "set" command will set up the **INCLUDE** environment variable so that the Open Watcom C/C++ compiler may find header files. Notice that the old value of the **INCLUDE** environment variable is used in setting the new value.

The second "set" command indicates to the Open Watcom Linker that libraries may be found in the indicated directories.

Environment variables may be used also as dynamic variables that may communicate information between different parts of the makefile. An example of communication within a makefile is illustrated in the following example.

```
#
# internal makefile communication
#
.BEFORE
        set message=message text 1
        echo *$(%message)*
        set message=
        echo *$(%message)*

.example : another_target .SYMBOLIC
        echo *$(%message)*

another_target : .SYMBOLIC
        set message=message text 2
```

The output of the previous makefile would be:

```
(command output only)
*message text 1*
**
*message text 2*
```

Make handles the "set" command so that it appears to work in an intuitive manner similar to the operating system shell's "set" command. The "set" command also may be used to allow commands to relay information to commands that are executed afterwards.

10.39.3 for command

The operating system shell "for" command is intercepted by Make. It uses following syntax:

```
for %[%]<var> in (<set>) do <command>
```

DOS has a fixed limit for the size of a command thus making it unusable for large makefile applications. One such application that can be done easily with Make is the construction of a WLINK command file from a makefile. The idea behind the next example is to have one file that contains the list of object files. Anytime this file is changed, say, after a new module has been added, a new linker command file will be generated which in turn, will cause the linker to relink the executable. First we need the makefile to define the list of object files, this file is "OBJDEF.MIF" and it declares a macro "objs" which has as its value the list of object files in the application. The content of the "OBJDEF.MIF" file is:

```
#
# list of object files
#
objs = &
     window.obj &
     bios.obj &
     keyboard.obj &
     mouse.obj
```

The main makefile ("MAKEFILE") is:

```
#
# for command example
#
!include objdef.mif

plot.exe : $(objs) plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        echo NAME $^& >$^@
        echo DEBUG all >>$^@
        for %i in ($(objs)) do echo FILE %i >>$^@
```

This makefile would produce a file "PLOT.LNK" automatically whenever the list of object files is changed (anytime "OBJDEF.MIF" is changed). For the above example, the file "PLOT.LNK" would contain:

```
NAME plot
DEBUG all
FILE window.obj
FILE bios.obj
FILE keyboard.obj
FILE mouse.obj
```

10.39.4 if command

The operating system shell "if" command is intercepted by Make. It uses following syntax:

```
             { errorlevel <number> }
if [not]     { <str1> == <str2>    } <command>
             { exist <file>        }
```

It handles file names consistently with other Make commands.

10.39.5 rm command

The commands "rm" is intercepted by Make. It uses following syntax:

```
rm [-frv] <files/directories>
```

The "rm" command may be used to delete files or directories. Make "rm" command is simplified implementation of the POSIX rm command. It handles file/directory names consistently with other Make commands. Following options are support.

-f force deletion of read-only files, no diagnostics messages about missing items
-r deletion of directories
-v verbose operation

10.39.6 mkdir command

The commands "mkdir" is intercepted by Make. It uses following syntax:

```
mkdir [-p] <directory>
```

The "mkdir" command may be used to create a directory. Make "mkdir" command is simplified implementation of the POSIX "mkdir" command. It handles directory names consistently with other Make commands. Following options are support.

-p force creation of all parent directories

10.39.7 rmdir command

The commands "rmdir" is intercepted by Make. It uses following syntax:

```
rmdir <directory>
```

The "rmdir" command may be used to delete a directory. Make "rmdir" command is is simplified implementation of the POSIX "rmdir" command. It handles directory names consistently with other Make commands.

10.39.8 Make internal commands

Make supports nine internal commands:

1. %abort
2. %append
3. %create
4. %erase
5. %make
6. %null
7. %quit
8. %stop
9. %write

The %abort and %quit internal commands terminate execution of Make and return to the operating system shell: %abort sets a non-zero exit code; %quit sets a zero exit code.

```
#
# %abort and %quit example
#
done_ enough :
        %quit

suicide :
        %abort
```

The %append, %create, %erase, and %write internal commands allow WMAKE to generate files under makefile control. This is useful for files that have contents that depend on makefile contents. Through the use of macros and the "for" command, Make becomes a very powerful tool in maintaining lists of files for other programs.

The %append internal command appends a text line to the end of a file (which is created if absent) while the %write internal command creates or truncates a file and writes one line of text into it. Both commands have the same form, namely:

```
%append <file> <text>
%write <file> <text>
```

where <file> is a file specification and <text> is arbitrary text.

The %create internal command will create or truncate a file so that the file does not contain any text while the %erase internal command will delete a file. Both commands have the same form, namely:

```
%create <file>
%erase <file>
```

where <file> is a file specification.

Full macro processing is performed on these internal commands so the full power of WMAKE can be used. The following example illustrates a common use of these internal commands.

```
#
# %append, %create, %erase, and %write example
#
!include objdef.mif

plot.exe : $(objs) plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        %create $^@
        %append $^@ NAME $^&
        # Next line equivalent to previous two lines.
        %create $^@ NAME $^&
        %append $^@ DEBUG all
        for %i in ($(objs)) do %append $^@ FILE %i

clean : .SYMBOLIC
        %erase plot.lnk
```

The above code demonstrates a valuable technique that can generate directive files for WLINK, WLIB, and other utilities.

The %make internal command permits the updating of a specific target and has the form:

```
%make <target>
```

where <target> is a target in the makefile.

```
#
# %make example
#
!include objdef.mif

plot.exe : $(objs)
        %make plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        %create $^@
        %append $^@ NAME $^&
        %append $^@ DEBUG all
        for %i in ($(objs)) do %append $^@ FILE %i
```

There seem to be other ways of doing the same thing. Among them is putting plot.lnk into the list of dependencies:

```
#
# %make counter-example
#
!include objdef.mif

plot.exe : $(objs) plot.lnk
        wlink @plot

plot.lnk : objdef.mif
        %create $^@
        %append $^@ NAME $^&
        %append $^@ DEBUG all
        for %i in ($(objs)) do %append $^@ FILE %i
```

and using a make variable:

```
#
# %make counter-example
#
!include objdef.mif

plot.exe : $(objs)
        wlink NAME $^& DEBUG all FILE { $(objs) }
```

The `%null` internal command does absolutely nothing. It is useful because Make demands that a command list be present whenever a target is updated.

```
#
# %null example
#
all : application1 application2 .SYMBOLIC
        %null

application1 : appl1.exe .SYMBOLIC
        %null

application2 : appl2.exe .SYMBOLIC
        %null

appl1.exe : (dependents ...)
        (commands)

appl2.exe : (dependents ...)
        (commands)
```

Through the use of the `%null` internal command, multiple application makefiles may be produced that are quite readable and maintainable.

The `%stop` internal command will temporarily suspend makefile processing and print out a message asking whether Makefile processing should continue. Make will wait for either the "y" key (indicating that the Makefile processing should continue) or the "n" key. If the "n" key is pressed, makefile processing will stop. The `%stop` internal command is very useful for debugging makefiles but it may be used also to develop interactive makefiles.

```
#
# %stop example
#
all : appl1.exe .SYMBOLIC
        %null

appl1.exe : (dependents ...)
        @echo Are you feeling lucky? Punk!
        @%stop
        (commands)
```

10.40 Compatibility Between Open Watcom Make and UNIX Make

Open Watcom Make was originally based on the UNIX Make utility. The PC's operating environment presents a base of users which may or may not be familiar with the UNIX operating system. Make is designed to be a PC product with some UNIX compatibility. The line continuation in UNIX Make is a backslash ("\") at the end of the line. The backslash ("\") is used by the operating system for directory specifications and as such will be confused with line continuation. For example, you could type:

```
cd \
```

along with other commands ... and get unexpected results. However, if your makefile does not contain path separator characters ("\") and you wish to use "\" as a line continuation indicator then you can use the Make "u" (UNIX compatibility mode) option.

Also, in the UNIX operating system there is no concept of file extensions, only the concept of a file suffix. Make will accept the UNIX Make directive `.SUFFIXES` for compatibility with UNIX makefiles. The UNIX compatible special macros supported are:

Macro	*Expansion*
$@	full name of the target
$*	target with the extension removed
$<	list of all dependents
$?	list of dependents that are younger than the target

The extra checking of makefiles done by Make will require modifications to UNIX makefiles. The UNIX Make utility does not check for the existence of targets after the associated command list is executed so the "c" or the `.NOCHECK` directive should be used to disable this checking. The lack of a command list to update a target is ignored by the UNIX Make utility but Open Watcom Make requires the special internal command `%null` to specify a null command list. In summary, Make supports many of the features of the UNIX Make utility but is not 100% compatible.

10.41 Open Watcom Make Diagnostic Messages

This section lists the various warning and error messages that may be issued by the Open Watcom Make. In the messages below, %? character sequences indicate places in the message that are replaced with some other string.

1 Out of memory

2 Make execution terminated

3 Option %c%c invalid

4 %c%c must be followed by a filename

5 No targets specified

6 Ignoring first target in MAKEINIT

7 Expecting a %M

8 Invalid macro name %E

9 Ignoring out of place %M

10 Macros nested too deep

11 Unknown internal command

12 Program name is too long

13 No control characters allowed in options

14 Cannot execute %E: %Z

15 Syntax error in %s command

16 Nested %s loops not allowed

17 Token too long, maximum size is %d chars

18 Unrecognized or out of place character '%C'

19 Target %E already declared %M

20 Command list does not belong to any target

21 Extension(s) %E not defined

22 No existing file matches %E

23 Extensions reversed in implicit rule

24 More than one command list found for %E

25 Extension %E declared more than once

26 Unknown preprocessor directive: %s

27 Macro %E is undefined

28 !If statements nested too deep

29 !%s has no matching !if

30 Skipping !%1 block after !%2

31 %1 not allowed after !%2

32 Opening file %E: %Z

34 !%s pending at end of file

35 Trying to !%s an undefined macro

36 Illegal attempt to update special target %E

37 Target %E is defined recursively

38 %E does not exist and cannot be made from existing files

39 Target %E not mentioned in any makefile

40 Could not touch %E

41 No %s commands for making %E

42 Last command making (%L) returned a bad status

43 Deleting %E: %Z

44 %s command returned a bad status

45 Maximum string length exceeded

46 Illegal character value %xH in file

47 Assuming target(s) are .%s

48 Maximum %%make depth exceeded

49 Opening (%s) for write: %Z

50 Unable to write: %Z

51 CD'ing to %E: %Z

52 Changing to drive %C:

53 DOS memory inconsistency detected! System may halt ...

53 OS corruption detected

54 While reading (%s): %Z

59 !IF Parse Error

60 TMP Path/File Too Long

61 Unexpected End of File

62 Only NO(KEEP) allowed here

63 Non-matching "

64 Invalid String Macro Substitution

65 File Name Length Exceeded

66 Redefinition of .DEFAULT Command List

67 Non-matching { In Implicit Rule

68 Invalid Implicit Rule Definition

69 Path Too Long

70 Cannot Load/Unload DLL %E

71 Initialization of DLL %E returned a bad status

72 DLL %E returned a bad status

73 Illegal Character %C in macro name

74 in closing file %E

75 in opening file %E

76 in writing file %E

77 User Break Encountered

78 Error in Memory Tracking Encountered

79 Makefile may be Microsoft try /ms switch

11 The Touch Utility

11.1 Introduction

This chapter describes the Open Watcom Touch utility. Open Watcom Touch will set the time-stamp (i.e., the modification date and time) of one or more files. The new modification date and time may be the current date and time, the modification date and time of another file, or a date and time specified on the command line. This utility is normally used in conjunction with the Open Watcom Make utility. The rationale for bringing a file up-to-date without altering its contents is best understood by reading the chapter which describes the Make utility.

The Open Watcom Touch command line syntax is:

```
WTOUCH [options] file_spec [file_spec...]
```

The square brackets [ ] denote items which are optional.

options is a list of valid options, each preceded by a slash ("/") or a dash ("−"). Options may be specified in any order.

file_spec is the file specification for the file to be touched. Any number of file specifications may be listed. The wild card characters "*" and "?" may be used.

The following is a description of the options available.

c	do not create an empty file if the specified file does not exist
d <date>	specify the date for the file time-stamp in "mm-dd-yy" format
f <file>	use the time-stamp from the specified file
i	increment time-stamp before touching the file
q	suppress informational messages
r	touch file even if it is marked read-only
t <time>	specify the time for the file time-stamp in "hh:mm:ss" format
u	use USA date/time format regardless of country
?	display help screen

11.2 WTOUCH Operation

WTOUCH is used to set the time-stamp (i.e., the modification date and time) of a file. The contents of the file are not affected by this operation. If the specified file does not exist, it will be created as an empty file. This behaviour may be altered with the "c" option so that if the file is not present, a new empty file will not be created.

Example:

```
(will not create myfile.dat)
C>wtouch /c myfile.dat
```

If a wild card file specification is used and no files match the pattern, no files will have their time-stamps altered. The date and time that all the specified files are set to is determined as follows:

1. The current date and time is used as a default value.

2. A time-stamp from an "age file" may replace the current date and time. The "f" option is used to specify the file that will supply the time-stamp.

   Example:

   ```
   (use the date and time from file "last.tim")
   C>wtouch /f last.tim file*.dat
   ```

3. The date and/or time may be specified from the command line to override a part of the time-stamp that will be used. The "d" and "t" options are used to override the date and time respectively.

   Example:

   ```
   (use current date but use different time)
   C>wtouch /t 2:00p file*.dat
   (completely specify date and time)
   C>wtouch /d 10-31-90 /t 8:00:00 file*.dat
   (use date from file "last.tim" but set time)
   C>wtouch /f last.tim /t 12:00 file*.dat
   ```

   The format of the date and time on the command line depends on the country information provided by the host operating system. Open Watcom Touch should accept dates and times in a similar format to any operating system utilities (i.e., the DATE and TIME utilities provided by DOS). The "a" and "p" suffix is an extension to the time syntax for specifying whether the time is A.M. or P.M., but this is only available if the operating system is not configured for military or 24-hour time.

The IDE2MAKE Batch Utility

12 The IDE2MAKE Utility

12.1 Introduction

This chapter describes the IDE2MAKE utility. IDE2MAKE loads an IDE project file and, using the associated .tgt files, generates make files that can be invoked with WMAKE. If the .wpj file does not exist, a default project is used. If any tgtfile(s) are specified, they are used as the targets in creating the make files.

The IDE2MAKE command line syntax is:

```
ide2make [options] [tgtfile]
```

The square brackets [ ] denote items which are optional. At least one item must be specified, otherwise the program usage is shown.

options is a list of valid options, each preceded by a slash ("/") or a dash ("−"). Options may be specified in any order.

tgtfile is the file specification for the targetfile to be used. Any number of file specifications may be listed.

The following is a description of the options available.

p <wpjfile> loads wpjfile.wpj (project.wpj by default)
c <cfgfile> loads cfgfile instead of ide.cfg
i directory to search configuration files
d generate makefiles using development switch set
l generate makefiles with long lines (no length limit)
r generate makefiles using release switch set
h <number> generate makefiles for selected host OS (default is current host)
0 - Windows 3.x
1 - OS/2 PM
2 - Windows NT
3 - Win-OS/2
4 - Windows 95
5 - Japanese Windows 3.x on an IBM
6 - Japanese Windows 3.x on a Nec98
7 - Dec Alpha (Windows NT)
8 - DOS
9 - Linux

12.2 IDE2MAKE Operation

IDE2MAKE is used to create makefiles from IDE project and targetfiles. If no targetfile is specified, makefiles for all targets are generated.

.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W