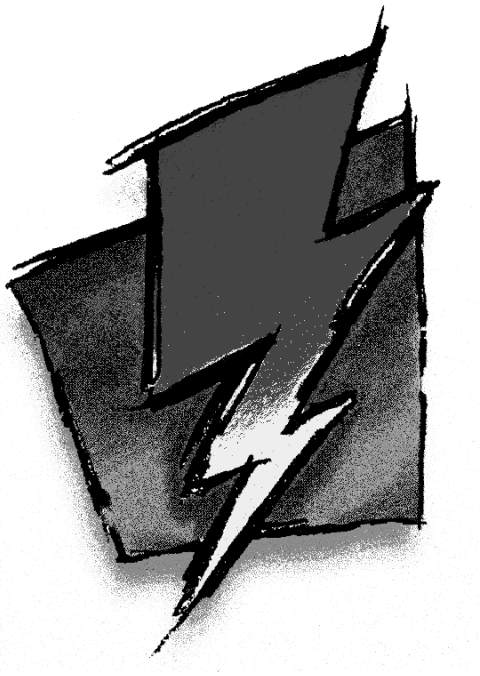
Watcom FORTRAN 77 Tools

User's Guide



Edition 11.0c

Notice of Copyright

Printed in U.S.A.

Preface

The *Watcom FORTRAN 77 Tools User's Guide* describes how to use Watcom's software development tools on Intel 80x86-based personal computers with DOS, Windows, Windows NT, or OS/2. The *Watcom FORTRAN 77 Tools User's Guide* describes the following tools:

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

Acknowledgements

This book was produced with the 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 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.

September, 2000.

Trademarks Used in this Manual

OS/2 is a trademark of International Business Machines Corp. IBM is a registered trademark of International Business Machines Corp.

Intel are registered trademarks of Intel Corp.

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

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

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

QNX is a registered trademark of QNX Software Systems Ltd.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

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

Table of Contents

Table of Contents

Table of Contents

The Watcom Compile and Link Utility

1 The Watcom FORTRAN 77 Compile and Link Utility

The Watcom FORTRAN 77 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 compiled; object files and libraries are simply included in the link phase. Options can be passed on to both the compiler and linker.

1.1 WFL/WFL386 Command Line Format

The format of the command line is:

```
WFL [files] [options]
WFL386 [files] [options]
```

The square brackets [ ] denote items which are optional.

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

WFL386 is the name of the Watcom Compile and Link utility that invokes the 32-bit compiler.

The files and options may be specified in any order. The 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 any other extension, including none at all, are assumed to be FORTRAN 77 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 "FOR" is assumed.

Options are prefixed with a slash (/) or a dash (–) and may be specified in any order. Options can include any of the Watcom F77 compiler options plus some additional options specific to

the Watcom Compile and Link utility. Certain options can include a "NO" prefix to disable an option. A summary of options is displayed on the screen by simply entering the "WFL" or "WFL386" command with no arguments.

1.2 Watcom Compile and Link Options Summary

General options:	***Description:***
C	compile the files only, do not link them
Y	ignore the WFL/WFL386 environment variable

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

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

OL	perform loop optimizations
OL+	perform loop optimizations with loop unrolling
OM	generate floating-point 80x87 math instructions in-line
ON	numeric optimizations
OP	precision optimizations
OR	instruction scheduling
OS	optimize for space
OT	optimize for time
OX	equivalent to OBP, ODO, OI, OK, OL, OM, OR, and OT (16-bit) or OB, OBP, ODO, OI, OK, OL, OM, OR, and OT (32-bit)
PRint	write listing file to printer
[NO]Quiet	operate quietly
[NO]Reference	issue unreferenced warning
[NO]RESource	messages in resource file
[NO]SAve	SAVE local variables
[NO]SC	(32-bit only) stack calling convention
[NO]SEpcomma	allow comma separator in formatted input
[NO]SG	(32-bit only) automatic stack growing
[NO]SHort	set default INTEGER/LOGICAL size to 2/1 bytes
[NO]SR	save/restore segment registers
[NO]SSfloats	(16-bit only) SS is not default data segment
[NO]STack	generate stack checking code
[NO]SYntax	syntax check only
[NO]TErminal	messages to terminal
[NO]TRace	generate code for run-time traceback
TYpe	write listing file to terminal
[NO]WArnings	issue warning messages
[NO]WILd	relax wild branch checking
[NO]WIndows	(16-bit only) compile code for Windows
[NO]XFloat	extend floating-point precision
[NO]XLine	extend line length to 132

Linker options:	***Description:***
FD[=<directive_file>]	keep directive file and, optionally, rename it (default name is "__WFL__.LNK").
FE=<executable>	name executable file
FI=<fn>	include additional directive file
FM[=<map_file>]	generate map file and, optionally, name it
K=<stack_size>	set stack size
LP	(16-bit only) create an OS/2 protected-mode program
LR	(16-bit only) create a DOS real-mode program

6 Watcom Compile and Link Options Summary

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.
ADS	32-bit AutoCAD ADS executables
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
EADI	32-bit AutoCAD ADI executables (emulation)
FADI	32-bit AutoCAD ADI executables (floating-point)
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
OS2V2	32-bit OS/2 executables
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 95 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 Watcom Linker initialization file, "WLSYSTEM.LNK". The Watcom Linker "SYSTEM" directives, found in this file, are used to specify default link options for particular (operating) systems. Users can augment the 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" Watcom Linker directive when linking the application.

*"<**linker directives**>"* specify additional linker directives

A summary of the option defaults follows:

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

8 Watcom Compile and Link Options Summary

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

1.3 WFL/WFL386 Environment Variables

The **WFL** environment variable can be used to specify commonly used ***WFL*** options. The **WFL386** environment variable can be used to specify commonly used ***WFL386*** options. These options are processed before options specified on the command line.

Example:

```
C>set wfl=/d1 /ot

C>set wfl386=/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 ***WFL*** or ***WFL386*** command is used.

The **WFL** environment variable is used by ***WFL*** only. The **WFL386** environment variable is used by ***WFL386*** only. Both ***WFL*** and ***WFL386*** pass the relevant options to the Watcom F77 compiler and linker. This environment variable is not examined by the Watcom F77 compiler or the linker when invoked directly.

> ***Hint:*** If you are running DOS and you use the same ***WFL*** or ***WFL386*** options all the time, you may find it handy to place the "SET WFL" or "SET WFL386" command in your DOS system initialization file, `AUTOEXEC.BAT`. If you are running OS/2 and you use the same ***WFL*** or ***WFL386*** options all the time, you may find it handy to place the "SET WFL" or "SET WFL386" command in your OS/2 system initialization file, `CONFIG.SYS`.

1.4 WFL/WFL386 Command Line Examples

For most small applications, the ***WFL*** or ***WFL386*** command will suffice. We have only scratched the surface in describing the capabilities of the ***WFL*** and ***WFL386*** commands. The following examples describe the ***WFL*** and ***WFL386*** commands in more detail.

Suppose that your application is contained in three files called `APDEMO.FOR`, `APUTILS.FOR`, and `APDATA.FOR`. We can compile and link all three files with one command.

Example 1:

```
C>wfl /d2 apdemo.for aputils.for apdata.for
C>wfl386 /d2 apdemo.for aputils.for apdata.for
```

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 FORTRAN 77 source files.

Example 2:

```
C>wfl /d2 *.for
C>wfl386 /d2 *.for
```

WFL or ***WFL386*** will locate all files with the ".for" filename extension and compile each of them. The name of the executable file will depend on which of the FORTRAN 77 source files

is found first. Since this is a somewhat haphazard approach to naming the executable file, ***WFL*** and ***WFL386*** have an option, "fe", which will allow you to specify the name to be used.

Example 3:

```
C>wfl /d2 /fe=apdemo *.for
C>wfl386 /d2 /fe=apdemo *.for
```

By using the "fe" option, the executable file will always be called `APDEMO.EXE` regardless of the order of the FORTRAN 77 source files in the directory.

If the directory contains other FORTRAN 77 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>wfl /d2 /fe=apdemo ap*.for
C>wfl386 /d2 /fe=apdemo ap*.for
```

Here we compile only those FORTRAN 77 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>wfl /d2 /fe=apdemo aputils.for ap*.obj
C>wfl386 /d2 /fe=apdemo aputils.for ap*.obj
```

The source file `APUTILS.FOR` 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>wfl /fe=demo *.for utility.obj
C>wfl386 /fe=demo *.for utility.obj
```

All of the FORTRAN 77 source files in the current directory are compiled and then linked with `UTILITY.OBJ` to generate `DEMO.EXE.`

Example 7:

```
C>set wfl=/mm /d1 /op /k=4096
C>wfl /fe=grdemo gr*.for graph.lib /fd=grdemo

C>set wfl386=/d1 /op /k=4096
C>wfl386 /fe=grdemo gr*.for graph.lib /fd=grdemo
```

All FORTRAN 77 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 ***WFL*** or ***WFL386*** 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>set wfl=/mm /lp
C>wfl 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.

12 WFL/WFL386 Command Line Examples

The Watcom Assembler

2 The Watcom Assembler

2.1 Introduction

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

The 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 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 Watcom Assembler are:

{0,1,2,3,4,5}{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")
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"

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
zq or q	operate quietly
? or h	print this message
w<number>	set warning level number
we	treat all warnings as errors

2.2 Assembly Directives and 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
.8086           .8087           aaa             aad
aam             aas             abs             adc
add             addr            ah              al
alias           align           .alpha          and
arpl            assume          at              ax
basic           bh              bl              bound
bp              .break          bsf             bsr
bswap           bt              btc             btr
bts             bx              byte            c
call            callf           casemap         catstr
cbw             cdq             ch              cl
clc             cld             cli             clts
cmc             cmp             cmps            cmpsb
cmpsd           cmpsw           cmpxchg         cmpxchg8b
.code           comm            comment         common
compact         .const          .continue       cpuid
cr0             cr2             cr3             cr4
.cref           cs              cwd             cwde
cx              daa             das             .data
.data?          db              dd              dec
df              dh              di              div
dl              .dosseg         dp              dq
dr0             dr1             dr2             dr3
dr6             dr7             ds              dt
dup             dw              dword           dx
eax             ebp             ebx             echo
ecx             edi             edx             else
elseif          end             endif           endp
ends            .endw           enter           eq
equ             equ2            .err            .errb
.errdef         .errdif         .errdifi        .erre
.erridn         .erridni        .errnb          .errndef
.errnz          error           es              esi
esp             even            .exit           export
extern          externdef       extrn           f2xm1
fabs            fadd            faddp           far
.fardata        .fardata?       farstack        fbld
```

```
fbstp       fchs        fclex       fcom
fcomp       fcompp      fcos        fdecstp
fdisi       fdiv        fdivp       fdivr
fdivrp      feni        ffree       fiadd
ficom       ficomp      fidiv       fidivr
fild        fimul       fincstp     finit
fist        fistp       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      for
forc        fortran     fpatan      fprem
fprem1      fptan       frndint     frstor
frstord     frstorw     fs          fsave
fsaved      fsavew      fscale      fsetpm
fsin        fsincos     fsqrt       fst
fstcw       fstenv      fstenvd     fstenvw
fstp        fstsw       fsub        fsubp
fsubr       fsubrp      ftst        fucom
fucomp      fucompp     fwait       fword
fxam        fxch        fxtract     fyl2x
fyl2xp1     ge          global      group
gs          gt          high        highword
hlt         huge        idiv        if
if1         if2         ifb         ifdef
ifdif       ifdifi      ife         ifidn
ifidni      ifnb        ifndef      ignore
imul        in          inc         include
includelib  ins         insb        insd
insw        int         into        invd
invlpg      invoke      iret        iretd
irp         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          label
lahf        lar         large       lds
le          lea         leave       length
lengthof    les         .lfcond     lfs
lgdt        lgs         lidt        .list
```

```
.listall        .listif         .listmacro      .listmacroall
lldt            lmsw            local           lock
lods            lodsb           lodsd           lodsw
loop            loope           loopne          loopnz
loopz           low             lowword         lroffset
lsl             lss             lt              ltr
macro           mask            medium          memory
mod             .model          mov             movs
movsb           movsd           movsw           movsx
movzx           mul             name            ne
near            nearstack       neg             no87
.nocref         .nolist         nop             not
nothing         offset          opattr          option
or              org             os_dos          os_os2
out             outs            outsb           outsd
outsw           para            pascal          pop
popa            popad           popcontext      popf
popfd           private         proc            proto
ptr             public          purge           push
pusha           pushad          pushcontext     pushf
pushfd          pword           qword           .radix
rcl             rcr             rdmsr           rdtsc
readonly        record          rep             repe
.repeat         repne           repnz           repz
ret             retf            retn            rol
ror             rsm             sahf            sal
.sall           sar             sbb             sbyte
scas            scasb           scasd           scasw
sdword          seg             segment         .seq
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            .sfcond         sgdt
shl             shld            short           shr
shrd            si              sidt            size
sizeof          sldt            small           smsw
sp              ss              st              .stack
.startup        stc             std             stdcall
sti             stos            stosb            stosd
stosw           str             struc           struct
sub             sword           syscall         tbyte
test            textequ         .tfcond         this
tiny            tr3             tr4             tr5
tr6             tr7             typedef         union
```

20 Unsupported Directives

```
.until          use16           use32           uses
vararg          verr            verw            wait
watcom_c        wbinvd          .while          width
word            wrmsr           xadd            xchg
.xcref          xlat            xlatb           .xlist
xor
```

2.3 Unsupported Directives

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

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

2.4 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

22 Watcom Assembler Diagnostic Messages

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

24 Watcom Assembler Diagnostic Messages

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

26 Watcom Assembler Diagnostic Messages

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 already 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 -

28 Watcom Assembler Diagnostic Messages

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 0 written

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

30 Watcom Assembler Diagnostic Messages

Object File Utilities

3 The Watcom Library Manager

3.1 Introduction

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

- DOS
- OS/2
- QNX
- Windows

3.2 The Watcom Library Manager Command Line

The following describes the 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 Watcom Library Manager, options are preceded by a "/" or "—" character. If you are using a QNX-hosted version of the 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 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 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
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)
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

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 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".

3.3 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 Watcom Library Manager, a file extension of "obj" is assumed if none is specified. If you are using a QNX-hosted version of the 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 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 Watcom Library Manager will assume you are adding an object file.

3.4 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 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 Watcom Library Manager will assume you are removing an object module.

3.5 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 Watcom Library Manager will assume you are replacing an object module.

3.6 Extracting a Module from a Library File

A module can be extracted from a library file by specifying a ***mod_name** command for a DOS, OS/2 or Windows-hosted version of the Watcom Library Manager or a **:mod_name** command for a QNX-hosted version of the 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        QNX-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 Watcom Library Manager, a file extension of "obj" will be used. If you are running a QNX-hosted version of the 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    QNX-hosted
```

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

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 Watcom Library Manager or a **:-mod_name** command for a QNX-hosted version of the 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   QNX-hosted
```

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

3.7 Creating Import Libraries

The 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 Watcom Library Manager will assume you are adding an object file.

3.8 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.`

3.9 Commands from a File or Environment Variable

The 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 Watcom Library Manager.

3.10 Watcom Library Manager Options

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

3.10.1 Suppress Creation of Backup File - "b" Option

The "b" option tells the 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.

3.10.2 Case Sensitive Symbol Names - "c" Option

The "c" option tells the 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".

3.10.3 Specify Output Directory - "d" Option

The "d" option tells the 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 Watcom Library Manager,

the module will be placed in the file "\obj\mymod.obj". If you are running a QNX-hosted version of the 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         QNX-hosted
```

3.10.4 Specify Output Format - "f" Option

The "f" option tells the 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

fm output MLIB format library

fo output OMF format library

3.10.5 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.

3.10.6 Creating a Listing File - "l" Option

The "l" (lower case "L") option instructs the 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 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 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
```

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

The "m" option instructs the 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.

3.10.8 Always Create a New Library - "n" Option

The "n" option tells the 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".

3.10.9 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.

3.10.10 Specifying a Library Record Size - "p" Option

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 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.

3.10.11 Operate Quietly - "q" Option

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

Example:

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

3.10.12 Strip Line Number Records - "s" Option

The "s" option tells the 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
```

3.10.13 Trim Module Name - "t" Option

The "t" option tells the 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
```

3.10.14 Operate Verbosely - "v" Option

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

Example:

```
wlib -v -l mylib
```

3.10.15 Explode Library File - "x" Option

The "x" option tells the 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 Watcom Library Manager, the module will be placed in the "\obj" directory. If you are running a QNX-hosted version of the 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      QNX-hosted
```

3.11 Librarian Error Messages

The following messages may be issued by the 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 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 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.

4 The Object File Disassembler

4.1 Introduction

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

The 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 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 Watcom Disassembler are:

a	write assembly instructions only to the listing file
e	include list of external names
fp	do not use instruction name pseudonyms
fr	do not use register name pseudonyms [Alpha only]
fi	use alternate indexing format [80(x)86 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
o	print list of operands beside instructions
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.

4.2 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
```

4.3 The Assembly Format Option - "a"

The "a" option controls the format of the output produced to the listing file. When specified, the 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 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.

4.4 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 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
----------------
CALENDAR         000000cf 0000008b 00000047
CLEARSCREEN      0000000a
GETDAT           00000018
POSCURSOR        000000e8
-------------------------------------------------------------
```

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.

4.5 The No Instruction Name Pseudonyms Option - "fp"

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

4.6 The No Register Name Pseudonyms Option - "fr"

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

4.7 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]
```

4.8 The Uppercase Instructions/Registers Option - "fu"

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

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

By default, the Watcom Disassembler produces its output to the terminal. The "l" (lowercase L) option instructs the 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 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 Watcom Disassembler is instructed to disassemble the contents of the file `CALENDAR.OBJ` and produce the output to a listing file called `CALENDAR.LIS`.

4.10 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 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.

```
List of public symbols

SYMBOL          GROUP           SEGMENT          ADDRESS
-------------------------------------------------------
BOX                             BOX_TEXT         00000000
CALENDAR                        CALENDAR_TEXT    00000000
CLEARSCREEN                     CLEARSCREEN_TEXT 00000000
FMAIN                           FMAIN_TEXT       00000000
LINE                            LINE_TEXT        00000000
POSCURSOR                       POSCURSOR_TEXT   00000000

-------------------------------------------------------------
```

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

4.11 Retain C++ Mangled Names - "m"

The "m" option instructs the 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.

4.12 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.FOR` with "d1" debugging information. We then disassemble it as follows:

Example:

```
C>wdis mysrc /s /l
```

In the above example, the 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.FOR`.

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.for /l
```

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

4.13 An Example

Consider the following program contained in the file `HELLO.FOR`.

```
program main
print *, 'Hello world'
end
```

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 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.FOR`. The register names will be displayed in upper case (the "fu" option was specified). The output, shown below, is the result of using the Watcom F77 compiler.

```
Module: hello.for
Group: 'DGROUP' CONST,_DATA,_BSS
Group: 'FLAT'

Segment: 'FMAIN_TEXT' BYTE USE32  00000014 bytes

        program main
        print *, 'Hello world'
 0000  e8 00 00 00 00    FMAIN           call    RT@IOWrite
 0005  b8 00 00 00 00                    mov     EAX,offset L2
 000a  e8 00 00 00 00                    call    RT@OutCHAR
 000f  e9 00 00 00 00                    jmp     RT@EndIO
```

```
No disassembly errors

List of external symbols

Symbol
----------------
RT@EndIO         00000010
RT@IOWrite       00000001
RT@OutCHAR       0000000b
-------------------------------------------------------------

Segment: 'CONST' WORD USE32  0000000b bytes
 0000  48 65 6c 6c 6f 20 77 6f L1              - Hello wo
 0008  72 6c 64                                - rld
No disassembly errors

-------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000008 bytes
 0000  00 00 00 00             L2              DD      DGROUP:L1
 0004  0b 00 00 00                             - ....
No disassembly errors

-------------------------------------------------------------
List of public symbols

SYMBOL          GROUP           SEGMENT          ADDRESS
-------------------------------------------------------------
FMAIN                           FMAIN_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 Watcom F77 compiler.

```
.387
.386p
                PUBLIC  FMAIN
                EXTRN   ‘RT@IOWrite‘:BYTE
                EXTRN   ‘RT@OutCHAR‘:BYTE
                EXTRN   ‘RT@EndIO‘:BYTE
                EXTRN   __init_error:BYTE
                EXTRN   __init_english:BYTE
                EXTRN   __init_387_emulator:BYTE
                EXTRN   _cstart_:BYTE
DGROUP          GROUP   CONST,_DATA,_BSS
CONST           SEGMENT PARA PUBLIC USE32 ’DATA’
L$1:
                DB      0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f
                DB      0x72, 0x6c, 0x64

CONST           ENDS
_DATA           SEGMENT PARA PUBLIC USE32 ’DATA’
L$2:
                DD      L$1
                DB      0x0b, 0x00, 0x00, 0x00

_DATA           ENDS
_BSS            SEGMENT PARA PUBLIC USE32 ’BSS’
_BSS            ENDS

_TEXT           SEGMENT BYTE PUBLIC USE32 ’CODE’
                ASSUME CS:_TEXT, DS:DGROUP, SS:DGROUP
FMAIN:
    CALL      near ptr ‘RT@IOWrite‘
    MOV       EAX,offset L$2
    CALL      near ptr ‘RT@OutCHAR‘
    JMP       near ptr ‘RT@EndIO‘
_TEXT           ENDS
                END
```

5 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 Watcom Linker.

The Watcom C, C++ and FORTRAN 77 compilers automatically enable the far call optimization. The 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 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.`

5.1 Far Call Optimizations for Non-Watcom Object Modules

The far call optimization is automatically enabled when object modules created by the 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.

5.1.1 The Watcom Far Call Optimization Enabling Utility

Only DOS, OS/2 and Windows-hosted versions of the 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 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 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".

Executable Image Utilities

6 The Watcom Patch Utility

6.1 Introduction

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

6.2 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 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 Watcom.

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

```
bpatch wlink.a
```

The 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".

6.3 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 Watcom patch file

The patch file is not of the required format. The required header information is not present.

'%s' is not a valid 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.

7 The Watcom Strip Utility

7.1 Introduction

The 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 Watcom Linker or the Watcom Strip Utility. Resource information is placed at the end of an executable by a resource compiler or the Watcom Strip Utility.

Once a program has been debugged, the 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 Watcom Linker can be specified as input to the Watcom Strip Utility. Note that for executable files created for Novell's NetWare 386 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 Watcom Strip Utility currently runs under the following operating systems.

- DOS
- OS/2
- QNX
- Windows NT
- Windows 95

7.2 The Watcom Strip Utility Command Line

The 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 Watcom Strip Utility will assume one of the following extensions: "exe", "dll", "exp", "rex", "nlm", "dsk", "lan", "nam", "qnx" or no file extension. Note that the order specified in the list of file extensions is the order in which the 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 Watcom Strip Utility will create a temporary file, ensuring that a file by the chosen name does not already exist.

7.3 Strip Utility Messages

The following messages may be issued by the 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.

72 Strip Utility Messages

The Make/Touch Utilities

8 The Watcom Make Utility

8.1 Introduction

The 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. 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 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.

8.2 Watcom Make Reference

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

8.2.1 Watcom Make Command Line Format

The formal 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 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 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.

targets is one or more targets described in the makefile.

8.2.2 Watcom Make Options Summary

In this section, we present a terse summary of the 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
/t	touch files instead of executing commands
/u	UNIX compatibility mode
/z	do not erase target after error/interrupt (disables prompting)

8.2.3 Command Line Options

Command line options, available with 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 may be used to disable target existence 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". 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 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
__WINDOWS__ = defined if Windows version
__NT__ = defined if Windows NT version
__NT386__ = defined if 32-bit Windows NT version
__OS2__ = defined if OS/2 version
__QNX__ = defined if QNX version
#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

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__`. 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.

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 "&".

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.

8.2.4 Special Macros

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.
__MAKEOPTS__	contains all of the command line options that WMAKE was invoked with except for the "f" options
__MAKEFILES__	contains the names of all of the makefiles processed at the time of expansion (includes the file currently being processed)

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\

8.3 Dependency Declarations

In order for 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 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.* 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".

8.4 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 Watcom Make will execute the specified command.

8.5 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.

8.6 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.

8.7 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 Watcom F77 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
#
.for.obj: .AUTODEPEND
        wfc386 $[* $(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 Watcom Resource Compiler can also insert dependency information into a resource file that can be used by Make.

8.8 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.

8.9 Preserving Targets (.PRECIOUS)

Most operating system utilities and programs have special return codes that indicate error conditions. 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.lst summary.lst : ledger.dat 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.LST" or "SUMMARY.LST". If only one of the files is precious then the makefile could be coded as follows:

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

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

8.10 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 summary.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 summary.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.

8.11 Erasing Targets After Error (.ERASE)

Most operating system utilities and programs have special return codes that indicate error conditions. 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 summary.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" or "SUMMARY.LST" depending on which it was updating.

8.12 Preserving Targets After Error (.HOLD)

Most operating system utilities and programs have special return codes that indicate error conditions. 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 summary.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" or "SUMMARY.LST".

8.13 Suppressing Terminal Output (.SILENT)

As commands are executed, 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.lst summary.lst : 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.lst summary.lst : 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.

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

8.14 Macros

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 FORTRAN 77 source files and two include 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.for defs.fi globals.fi
        wfc386 main /mf /d1 /warn

calc.obj : calc.for defs.fi globals.fi
        wfc386 calc /mf /d1 /warn

input.obj : input.for defs.fi globals.fi
        wfc386 input /mf /d1 /warn

output.obj : output.for defs.fi globals.fi
        wfc386 output /mf /d1 /warn
```

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 = wfc386
compile_options = /mf /d1 /warn

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

main.obj : main.for defs.fi globals.fi
        $(compiler) main $(compile_options)

calc.obj : calc.for defs.fi globals.fi
        $(compiler) calc $(compile_options)

input.obj : input.for defs.fi globals.fi
        $(compiler) input $(compile_options)

output.obj : output.for defs.fi globals.fi
        $(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.for defs.fi globals.fi
        $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 include files but suppose we had very many include 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 = wfc386
compile_options = /mf /d1 /warn

include_files = defs.fi globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj
```

```
plot.exe : $(object_files)
        wlink $(link_options) @plot

main.obj : main.for $(include_files)
        $(compiler) main $(compile_options)

calc.obj : calc.for $(include_files)
        $(compiler) calc $(compile_options)

input.obj : input.for $(include_files)
        $(compiler) input $(compile_options)

output.obj : output.for $(include_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. 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.

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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = defs.fi globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @plot

main.obj : main.for $(include_files)
        $(compiler) main $(compile_options)

calc.obj : calc.for $(include_files)
        $(compiler) calc $(compile_options)

input.obj : input.for $(include_files)
        $(compiler) input $(compile_options)

output.obj : output.for $(include_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 /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

compiler_command = wfc386 $[* $(compile_options)

include_files = defs.fi globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj
```

```
plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

main.obj : main.for $(include_files)
        $(compiler_command)

calc.obj : calc.for $(include_files)
        $(compiler_command)

input.obj : input.for $(include_files)
        $(compiler_command)

output.obj : output.for $(include_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.

8.15 Implicit Rules

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)
.for.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.FOR"? Make handles this with the `.EXTENSIONS` directive. The `.EXTENSIONS` directive declares which extensions are allowed to be used in implicit rules and how these extensions are ordered. The default `.EXTENSIONS` declaration is:

```
.EXTENSIONS:
.EXTENSIONS: .exe .exp .lib .obj .asm .c .for .pas .cob .h .fi .mif
```

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:  .exp
.EXTENSIONS:  .lib
.EXTENSIONS:  .obj
.EXTENSIONS:  .asm .c .for .pas .cob .h .fi .mif
```

with identical results.

Make will not allow any implicit rule declarations that use extensions that are not in the current `.EXTENSIONS` list. Returning to our makefile fragment:

```
.pas.obj:
        (command list)
.for.obj:
        (command list)
```

and our target file "TEST.OBJ", we now know that the `.EXTENSIONS` list determines in what order the dependents "TEST.PAS" and "TEST.FOR" will be tried. If the .EXTENSIONS declaration is:

Example:

```
.EXTENSIONS:
.EXTENSIONS: .exe .obj .asm .pas .for .c .cob
```

we can see that the dependent file "TEST.PAS" will be tried first as a possible dependent with "TEST.FOR" 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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = defs.fi globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for.obj:
        $(compiler) $[* $(compile_options)

main.obj : main.for $(include_files)

calc.obj : calc.for $(include_files)

input.obj : input.for $(include_files)

output.obj : output.for $(include_files)
```

As this makefile is processed, any time an object file is found to be older than its associated source file or include 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.FOR". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".for.obj" implicit rule to be invoked for "CALC.OBJ". The file "CALC.FOR" is found to exist so the commands

```
wfc386 calc /mf /d1 /warn
echo linking debug version ...
wlink debug all @plot
```

are executed. The last two commands are a result of the compilation of "CALC.FOR" 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***
include files	`\EXAMPLE\INC`
source files	`\EXAMPLE\SRC`
rest	`\EXAMPLE\O`

Now the previous makefile (located in the \EXAMPLE\O sub-directory) would look like this:

```
#
# programming example
# (implicit rules)
#
i_dir   = \example\inc\ #sub-directory containing include files
s_dir   = \example\src\ #sub-directory containing source 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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = $(i_dir)defs.fi $(i_dir)globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj
```

```
plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(s_dir)main.for $(include_files)

calc.obj : $(s_dir)calc.for $(include_files)

input.obj : $(s_dir)input.for $(include_files)

output.obj : $(s_dir)output.for $(include_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" was older than "\EXAMPLE\SRC\CALC.FOR". The lack of a command list in the explicit rule with "CALC.OBJ" as a target causes the ".for.obj" implicit rule to be invoked for "CALC.OBJ". At this time, the file "\EXAMPLE\O\CALC.FOR" 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 ".FOR"). 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)
#
i_dir   = \example\inc\ #sub-directory containing include files
s_dir   = \example\src\ #sub-directory containing source 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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = $(i_dir)defs.fi $(i_dir)globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for:     $(s_dir)
.for.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(s_dir)main.for $(include_files)

calc.obj : $(s_dir)calc.for $(include_files)

input.obj : $(s_dir)input.for $(include_files)

output.obj : $(s_dir)output.for $(include_files)
```

Suppose "\EXAMPLE\O\CALC.OBJ" is older than "\EXAMPLE\SRC\CALC.FOR". The lack of a command list in the explicit rule with "CALC.OBJ" as a target will cause the ".for.obj" implicit rule to be invoked for "CALC.OBJ". The dependent extension ".FOR" has a path associated with it so the file "\EXAMPLE\SRC\CALC.FOR" is found to exist. The commands

```
wfc386 \EXAMPLE\SRC\CALC /mf /d1 /warn
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\INC
DEFS.FI, GLOBALS.FI

\EXAMPLE\SRC\PROGRAM
MAIN.FOR, CALC.FOR

\EXAMPLE\SRC\SCREEN
INPUT.FOR, OUTPUT.FOR

\EXAMPLE\O
PLOT.EXE, MAKEFILE, MAIN.OBJ, CALC.OBJ, INPUT.OBJ, OUTPUT.OBJ

the makefile would be changed to:

```
#
# programming example
# (implicit rules)
#
i_dir        = ..\inc\  # sub-directory with include files
                        # sub-directories with FORTRAN 77 source files
program_dir  = ..\for\program\ # - MAIN.FOR, CALC.FOR
screen_dir   = ..\for\screen\  # - INPUT.FOR, OUTPUT.FOR
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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = $(i_dir)defs.fi $(i_dir)globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for:     $(program_dir);$(screen_dir)
.for.obj:
        $(compiler) $[* $(compile_options)
```

```
main.obj : $(program_dir)main.for $(include_files)

calc.obj : $(program_dir)calc.for $(include_files)

input.obj : $(screen_dir)input.for $(include_files)

output.obj : $(screen_dir)output.for $(include_files)
```

Suppose that there is a change in the "DEFS.FI" file which causes all the source files to be recompiled. The implicit rule ".for.obj" is invoked for every object file so the corresponding ".FOR" file must be found for each ".OBJ" file. We will show where Make searches for the FORTRAN 77 source files.

```
update    main.obj
  test    ..\for\program\main.for            (it does exist)
  execute wfc386 ..\for\program\main /mf /d1 /warn

update    calc.obj
  test    ..\for\program\calc.for            (it does exist)
  execute wfc386 ..\for\program\calc /mf /d1 /warn

update    input.obj
  test    ..\for\program\input.for       (it does not exist)
  test    ..\for\screen\input.for            (it does exist)
  execute wfc386 ..\for\screen\input /mf /d1 /warn

update    output.obj
  test    ..\for\program\output.for      (it does not exist)
  test    ..\for\screen\output.for           (it does exist)
  execute wfc386 ..\for\screen\output /mf /d1 /warn

etc.
```

Notice that Make checked the sub-directory "..\SRC\PRORGAM" for the files "INPUT.FOR" and "OUTPUT.FOR". 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

i_dir        = ..\inc\  # sub-directory with include files
                        # sub-directories with FORTRAN 77 source files
program_dir  = ..\for\program\ # - MAIN.FOR, CALC.FOR
screen_dir   = ..\for\screen\  # - INPUT.FOR, OUTPUT.FOR
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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

include_files = $(i_dir)defs.fi $(i_dir)globals.fi
object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for:     $(program_dir);$(screen_dir)
.for.obj:
        $(compiler) $[* $(compile_options)

main.obj : $(program_dir)main.for $(include_files)

calc.obj : $(program_dir)calc.for $(include_files)

input.obj : $(screen_dir)input.for $(include_files)

output.obj : $(screen_dir)output.for $(include_files)
```

Suppose again that there is a change in the "DEFS.FI" file which causes all the source files to be recompiled. We will show where Make searches for the FORTRAN 77 source files using the optimized path specification searching.

```
update    main.obj
  test    ..\for\program\main.for            (it does exist)
  execute wfc386 ..\for\program\main /mf /d1 /warn

update    calc.obj
  test    ..\for\program\calc.for            (it does exist)
  execute wfc386 ..\for\program\calc /mf /d1 /warn
```

```
update    input.obj
  test    ..\for\program\input.for       (it does not exist)
  test    ..\for\screen\input.for            (it does exist)
  execute wfc386 ..\for\screen\input /mf /d1 /warn

update    output.obj
  test    ..\for\screen\output.for           (it does exist)
  execute wfc386 ..\for\screen\output /mf /d1 /warn

etc.
```

Make did not check the sub-directory "..\SRC\PROGRAM" for the file "OUTPUT.FOR" because the last successful attempt to find a ".FOR" file occurred in the "..\SRC\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 FORTRAN 77 may become difficult to maintain with all the include file dependencies. Ignoring include 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 include dependencies)
#
.OPTIMIZE

i_dir        = ..\inc\  # sub-directory with include files
                        # sub-directories with FORTRAN 77 source files
program_dir  = ..\for\program\ # - MAIN.FOR, CALC.FOR
screen_dir   = ..\for\screen\  # - INPUT.FOR, OUTPUT.FOR
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 = wfc386
compile_options_production = /mf /warn
compile_options_debugging = /mf /d1 /warn
compile_options = $(compile_options_$(version))

object_files = main.obj input.obj calc.obj &
               output.obj

plot.exe : $(object_files)
        echo $(msg_$(version))
        wlink $(link_options) @$^*

.for:     $(program_dir);$(screen_dir)
.for.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 FORTRAN 77 programs, you must ensure that you force Make to compile any programs affected by changes in include 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 include 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.

8.16 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.

8.17 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.

8.17.1 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 Watcom FORTRAN 77, 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\INC
PROTO.FI, GLOBALS.FI, BIOS_DEF.FI

\WINDOW\SRC
WINDOW.FOR, KEYBOARD.FOR, MOUSE.FOR, BIOS.FOR

\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 /warn          # common options
objs = window.obj bios.obj keyboard.obj mouse.obj

.for: ..\src
.for.obj:
        wfc $[* $(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\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
#
FOR_to_OBJ = $(compiler) $[* $(compile_options)

OBJECTS = WINDOW.OBJ BIOS.OBJ KEYBOARD.OBJ MOUSE.OBJ

WINDOW.OBJ : ..\SRC\WINDOW.FOR ..\INC\PROTO.FI ..\INC\GLOBALS.FI
    $(FOR_to_OBJ)
BIOS.OBJ : ..\SRC\BIOS.FOR ..\INC\BIOS_DEF.FI ..\INC\GLOBALS.FI
    $(FOR_to_OBJ)
KEYBOARD.OBJ : ..\SRC\KEYBOARD.FOR ..\INC\PROTO.FI ..\INC\GLOBALS.FI
    $(FOR_to_OBJ)
MOUSE.OBJ : ..\SRC\MOUSE.FOR ..\INC\PROTO.FI ..\INC\GLOBALS.FI
    $(FOR_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 /warn

first_target : window.lib .SYMBOLIC
        echo done

!include wmkmk.mif

window.lib : $(OBJECTS)
        wlib window $(OBJECTS)

make : .SYMBOLIC
        wmkmk /r ..\src\*.for+..\inc
```

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.

8.17.2 Conditional Processing

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\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 /nostack # 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        = wfc386

stack_overflow  = No   # yes -> check for stack overflow
line_info       = Yes  # yes -> generate line numbers

!ifeq compiler wfc386
!ifneqi stack_overflow   yes
stack_option    =       /nostack
!endif
!ifeqi line_info         yes
line_option     =       /d1
!endif
!endif

!ifeq compiler fl32
!ifeqi stack_overflow    yes
stack_option    =       -Ge
!endif
!ifeqi line_info         yes
line_option     =       -Zd
!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.

8.17.3 Loading Dynamic Link Libraries

Watcom Make supports loading of Dynamic Link Library (DLL) versions of 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    wlink
!  loaddll wlib     wlibd
!endif

.c.obj:
        $(CC) $*.c
```

The `__LOADDLL__` symbol is defined for versions of Watcom Make that support the `!loaddll` preprocessing directive. The `!ifdef __LOADDLL__` construct ensures that the makefile can be processed by an older version of 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 = wlink

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.

8.18 Command List Directives

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 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.

8.19 MAKEINIT File

As you become proficient at using 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" 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. The only default declaration that Make provides is equivalent to a "MAKEINIT" file containing:

```
__MAKEOPTS__ = <options passed to WMAKE>
__MAKEFILES__ = <list of makefiles>
__MSDOS__ =
# clear .EXTENSIONS list
.EXTENSIONS:

# set .EXTENSIONS list
.EXTENSIONS: .exe .exp .lib .obj .asm .c .for .pas .cob .h .fi .mif
```

For OS/2, the __MSDOS__ macro will be replaced by __OS2__ and for Windows NT, the __MSDOS__ macro will be replaced by __NT__. The use of a "MAKEINIT" file will allow you to reuse common declarations and will result in simpler, more maintainable makefiles.

8.20 Command List Execution

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.

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 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)
erase (erase files)
for (repetitively process commands, intercepted by WMAKE)
if (allow conditional processing of commands)
md (make directory)
mkdir (make directory)
path (set search path)
pause (suspend batch operations)
prompt (change command prompt)
ren (rename files)
rename (rename files)
rmdir (remove directory)
rd (remove directory)
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)

The operating system shell "SET" command is intercepted by Make. 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 Watcom F77 compiler.

```
#
# SET example
#
.BEFORE
        set finclude=c:\special\inc;$(%finclude)
        set lib=c:\watcom\lib386\dos
#
# rest of makefile follows
#
        .
        .
        .
```

The first "SET" command will set up the **FINCLUDE** environment variable so that the Watcom F77 compiler may find header files. Notice that the old value of the **FINCLUDE** environment variable is used in setting the new value.

The second "SET" command indicates to the 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.

The DOS "FOR" command is intercepted by Make. The reason for this is that 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
```

Make supports eight internal commands:

1. %null
2. %stop
3. %quit
4. %abort
5. %create
6. %write

7. %append
8. %make

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 the 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. The %quit internal command will terminate execution of Make and return to the operating system shell with an exit code of zero. The %abort internal command is identical to %quit except that a non-zero exit code is returned by WMAKE.

The %create, %write, and %append 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 %create internal command will create or truncate a file so that the file does not contain any text. The %create internal command has the form:

```
%create <file>
```

where <file> is a file specification. The %write internal command will create or truncate a file and write one line of text into it. The %append internal command will append a text line to the end of a file (which will be created if it does not exist). The %write and %append internal commands have the same form, namely:

```
%write <file> <text>
%append <file> <text>
```

where <file> is a file specification and <text> is arbitrary text. 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.

```
#
# %create %append 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
```

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. The %make internal command 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
```

8.21 Compatibility Between Watcom Make and UNIX Make

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 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.

8.22 Watcom Make Diagnostic Messages

This section lists the various warning and error messages that may be issued by the 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

140 Watcom Make Diagnostic Messages

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

9 The Touch Utility

9.1 Introduction

This chapter describes the Watcom Touch utility. 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 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 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

9.2 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. 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.

.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W