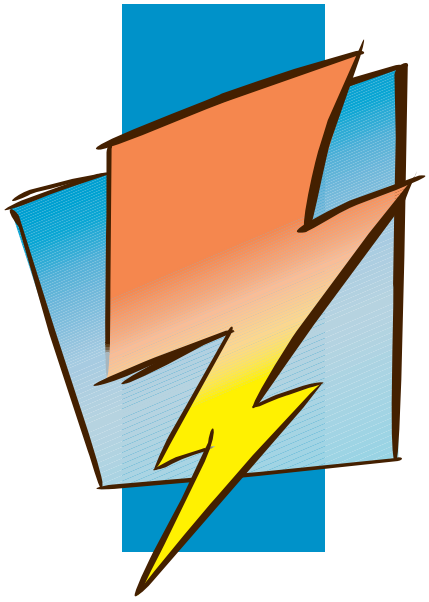
Open Watcom

Developer's Guide



Version 2.0

Open **Watcom**

Notice of Copyright

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

Table of Contents

Table of Contents

Introduction

1 Project Overview

This document serves as an introduction and a guide for developers of the Open Watcom compilers and tools. It is not particularly useful for the users (who are also developers) of Open Watcom compilers — they are encouraged to read the User's Guide, Programmer's Guide, C Language Reference and other user oriented books.

It should not be assumed that this book is in any way final or the ultimate reference. Readers are encouraged to add, change and modify this document to better reflect evolution of the Open Watcom project.

1.1 History

The history of the Open Watcom project is rather long, in terms of Interned years it would probably span millennia. The origins can be traced back to 1965. That summer a team of undergraduate students at the University of Waterloo develped a FORTRAN compiler (called WATFOR) that ran on the University's IBM 7040 systems. The compiler was soon ported to IBM 360 and later to the famous DEC PDP-11.

In early 1980s a brand new version of the compiler was created that supported the FORTRAN 77 language. It ran on two platforms, the IBM 370 and the emerging IBM PC. The PC version of WATFOR-77 was finished in 1985 and in the same year support for Japanese was added. In 1986, WATFOR-77 was ported to the QNX operating system.

The early compilers were written in a portable language called WSL or Watcom Systems Language. In late 1980s the developers rewrote the existing code in C and from then on all new developments were done on C, later with traces of C++ here and there.

In parallel to the FORTRAN compilers Watcom developed optimizing C compilers. When the first PC version (Watcom C 6.0) was introduced in 1987, it immediately attracted attention by producing faster code than other compilers available at that time.

In 1988 work started on an advanced highly optimizing code generator that supported both the C language and FORTRAN and was portable across multiple platforms. Generation of tight code, availability on multiple platforms (DOS, Windows, OS/2 and Windows NT in one package) and the ability to cross-compile made Watcom C and C++ compilers quite popular in mid-1990s. Around 1993-1996, nearly all DOS games were developed with Watcom C, including famous titles such as DOOM, Descent or Duke Nukem 3D.

Watcom International, Inc. had other successful products besides its highly acclaimed compilers. VX-REXX was a popular GUI RAD tool for OS/2 and Watcom SQL was a cross-platform "embeddable" SQL database.

In mid-1990s, Watcom International, Inc. was acquired by PowerSoft, the maker of Power++, PowerDesigner and other GUI RAD tools for the Windows platform. PowerSoft used Watcom compiler technology as a back-end for their GUI tools besides continuing to market and develop existing Watcom tools.

PowerSoft itself had merged with Sybase, Inc. in 1994. PowerSoft's development tools nicely complemented Sybase's database servers. Sybase was also interested in Watcom SQL which was enhanced and turned into Sybase SQL Anywhere.

Sybase continued to sell Watcom C/C++ and FORTRAN compilers version 11 but it was obvious that Sybase couldn't compete with Microsoft in the languages market. Sybase decided to end-of-life the Watcom compilers effective 2000.

But that's not the end of the story. Many customers did not want to give up the Watcom compilers because there was no suitable replacement in many areas. One of these customers was Kendall Bennett of SciTech Software, Inc. SciTech entered into negotiations with Sybase and in an unprecedented move, Sybase agreed upon open sourcing the Watcom compilers and tools. One of the reasons why this was possible at all was the fact that Watcom had very little reliance on third-party tools and source code and had developed practically everything in-house, from YACC to IDE.

The process of opening the source was longer than originally anticipated (all software related projects tend to work out this way for some inexplicable reason) but in the first half of 2002, the source was finally made available under the Sybase Open Watcom Public License version 1.0.

1.2 Guided Tour

This section will take you on a guided tour of the Open Watcom source tree, presenting an overview of its structure and highlighting some of the more memorable sights.

The Open Watcom directory structure mostly mirrors the layout used by the Watcom/Sybase build server but it has been cleaned up, straightened out and unified, although there still may be some warts.

The root of the Open Watcom directory tree can be in the root of any drive (if your OS uses drive letters) or in any directory, for instance `e:\ow`. Long filenames are not recommended if compatibility with DOS tools is desired. Directory names which include spaces are highly discouraged in any case.

The main subdirectories in the Open Watcom root are the following:

build contains main builder control files and other useful files. One extremely important file lives here: *makeinit*. This file controls the operation of wmake and is the key to understanding of the build process. Since wmake looks for *makeinit* along the `PATH`, the *build* directory should be placed at or near the start of your `PATH` environment variable.

build/bin contains binaries created during first build phase (bootstrap) and used during main building process (second build phase).

build/mif contains all main make files (global).

bld is *the* directory where it's at. It contains all the Open Watcom source code. It is so important (and huge) that it deserves its own section.

docs contains source files for the Open Watcom documentation as well as binaries needed to translate the sources into PostScript, HTML or various online help formats. The source files of this document are stored under this directory. For more information please refer the the chapter entitled Documentation later in this manual.

rel is the "release" directory is where the binaries and other files produced in the course of the build process end up. The structure of this directory mirrors the *WATCOM* directory of a typical Open Watcom installation.

1.3 The bld directory

Following is a brief description of all subdirectories of bld. Each subdirectory roughly corresponds to one "project". There's a lot of projects!

as the Alpha AXP and PowerPC assembler. The x86 assembler lives separately.

aui user interface library employed by the debugger and profiler.

bdiff binary diff and patch utilities.

bmp2eps a utility for converting Windows bitmap files into EPS format, used for building documentation.

brinfo part of the C++ source browser.

browser the GUI C++ source browser.

builder builder tool controlled by those *builder.ctl* files that are all over the place.

causeway the popular CauseWay DOS extender, in a form buildable with Open Watcom tools.

cc the C compiler front end.

cfloat utility function for conversion between various floating point binary formats.

cg Open Watcom code generators, the heart of the compilers. These are shared by all languages (C, C++, FORTRAN). Currently supported targets are 16-bit and 32-bit x86 as well as Alpha AXP.

clib the C runtime library. Pretty big project in itself.

cmdedit command line editing utilities, pretty much obsolete.

comp_cfg compiler configuration header files for various targets.

cpp a simple C style preprocessor used by several other projects.

ctest C compiler regression tests. Run them often.

cvpack the CV pack utility (for CodeView style debugging information).

diff Open Watcom version of the popular utility.

dig files used primarily by the debugger — this directory contains files that are shared between debugger, profiler, trap files and Dr. Watcom.

dip	Debug Information Processors, used by debugger. The DIPs provide an interface between the debugger and various debug information formats.
dmpobj	a simple OMF dump utility.
dwarf	library for reading and writing DWARF style debugging information.
editdll	interface modules between the IDE and external editors.
fpuemu	8087 and 80387 emulator library.
f77	FORTRAN 77 compiler front end, runtime library and samples. All the FORTRAN stuff is crowded in there.
f77test	FORTRAN 77 compiler regression tests.
fe_misc	miscellaneous compiler front-end stuff shared between projects.
fmedit	form edit library, part of the SDK tools.
graphlib	Open Watcom graphics library for DOS.
gui	GUI library used by IDE, debugger, source browser and other tools.
hdr	source files of header files distributed with the compilers.
help	character mode help viewer (WHELP).
ide	the Open Watcom IDE.
idebatch	batch processor for the IDE.
idedemo	IDE demo program.
lib_misc	miscellaneous files shared between clib and other tools.
mad	Machine Architecture Description used by debugger.
mathlib	the math library.
misc	stuff that didn't fit anywhere else. Not much really.
mstools	Microsoft clone tools, front ends for compilers and utilities.
ncurses	a version of the ncurses library used by Linux console tools.
ndisasm	the disassembler supporting variety of file format and instruction sets. Very handy.
nwlib	the library manager.
online	place for finished online help files and associated header files.
orl	Object Reader Library, reads OMF, COFF and ELF object files.

6 The bld directory

os2api	headers and libraries for the OS/2 API (both 16-bit and 32-bit).
owl	Object Writer Library, brother of ORL.
pgchart	presentation graphics and chart library for DOS (part of the graph library).
plusplus	another huge directory containing all C++ stuff. Compiler, runtime libraries, all that.
plustest	C++ regression test utilities. Extremely worthy of the attention of compiler developers.
pmake	parallel make, tool used in the build process to roughly control what gets built.
posix	a bunch of POSIX utilites like cp, rm and so on. Not suffering from creeping featuritis but they do the job and they're portable.
rcsdll	interface to various revision control systems, used by IDE and editor.
re2c	regular expression to C converter, used in C++ compiler build.
redist	miscellaneous redistributable files.
rtdll	C, C++ and math runtime DLLs.
sdk	SDK tools like resource editor, resource compiler or dialog editor. Also the home of wres library which is used by many other projects.
setupgui	source for the Open Watcom installer.
src	sample source code distributed with the compiler, some of it is used in the documentation.
ssl	internal tool used for debugger builds.
techinfo	ancient system information utility.
trap	trap files (both local and remote), the heart of the debugger containing platform specific debugging code. Heavy stuff.
trmem	memory tracker library (good for discovering and plugging memory leaks).
ui	user interface library.
vi	Open Watcom vi editor, clone of the popular (or not) Unix editor.
w16api	headers and libraries for the Windows 3.x API.
w32api	headers and libraries for the Win32 API.
w32loadr	loaders for OS independent (OSI) binaries.
wasm	the x86 assembler. Large parts of the source are shared between standalone wasm and inline assembler support for compilers targeting x86 platforms.
wasmtest	the x86 assembler regression tests.

watcom	contains internal headers and libraries shared by many projects.
wclass	an Open Watcom C++ class library.
wdisasm	old x86 disassembler, nearly obsolete.
whpcvt	Watcom Help Converter used for producing online documentation.
wic	utility for converting include files between various languages.
win386	the Windows 386 extender.
wl	the Open Watcom linker, also contains the overlay manager library.
wmake	the make utility.
womp	Watcom Object Module Processor, primarily for conversion between debug info formats. Some source files are shared with other projects.
wpack	simple file compression/decompression utility.
wpi	macros and helper functions for facilitating development of Windows and OS/2 GUI programs from single source code.
wprof	the Open Watcom profiler.
wsample	the execution sampler, companion tool to the profiler.
wstrip	strip utility for detaching or attaching debug information and/or resources.
wstub	stub program for DOS/4GW.
wtouch	a touch utility.
wv	the debugger (used to be called WVIDEO, hence the name).
yacc	Watcom's version of YACC used for building compilers/assemblers.

As you can see, there's a lot of stuff! Some of these projects contain specific documentation pertaining to them, usually located in a directory called 'doc' or somesuch. For the most part, the truly uptodate and comprehensive documentation is the source code.

2 First Steps

This chapter briefly describes the prerequisite steps necessary to build and/or contribute to the Open Watcom project — how to get the source code and how to set up the build environment.

2.1 Connecting up

The most uptodate version of the Open Watcom source code lives on the Open Watcom Perforce server. It is possible to go straight to the Perforce repository but most people will find it much easier to get a source archive first. The source archives can be found at the Open Watcom web site, `http://www.openwatcom.org/` along with latest information on Perforce setup. You will generally need a working installation of the previous release of Open Watcom C/C++ and some free disk space to burn (one gigabyte should do).

The Open Watcom source tree can be located in any directory on any drive. After extracting the source archive you will find a very important batch file called *setvars* in your Open Watcom root directory. This will set up a bunch of necessary environment variables but first you'll have to edit it to reflect your directory structure etc. It also contains the necessary Perforce settings.

Now is the time to connect to Perforce. Again, most uptodate information can be found on the Open Watcom web site. If you followed the instructions correctly, no servers are down and no other unpredictable (or maybe predictable) things happened, you will have brought your source tree to the latest revision (aka tip or head revision).

2.2 Gearing up for Building

Before you start building the Open Watcom tools proper, you will need to build several helper tools: *builder*, *pmake*, *cdsay* and a few others. These tools have to be built manually because the build process won't work without them.

The tools can be found in appropriately named subdirectory of the *bld* directory, which is named *builder* (showing complete lack of imagination).

To build the required executables, go to a subdirectory of the project *builder* directory which sounds like it would be appropriate for your host platform and run *wmake*. If you set up everything correctly, you will end up with working binaries that were automatically copied into the right subdirectory of the *build* directory, and that directory is already on the `PATH`. If not, it's back to square one — the most likely source of problems is incorrectly set up *setvars* batch file.

If you've got this far — congratulations, you've finished the one-time steps. You shouldn't need to redo them unless you decide to start from scratch, your harddrive decides to die or some similarly catastrophic event occurs.

You should now read the next chapter that describes the build architecture and also lists the magic incantations necessary to invoke builds.

Building

3 Build Architecture

In an effort to clean up the build process, make it easier for projects to compile on various people's machines and allow for easier ports to other architectures, every project which is developed under the Open Watcom Project should follow certain conventions as far as makefile layout is concerned. This section describes the conventions and requirements for these makefiles, as well as the steps needed to get projects to compile.

For those who do not desire a lecture on the preparation and maintenance of makefiles, feel free to skip straight to the Executive Summary at the end.

Every development and build machine must have the mif project (*build\mif*) installed. That is taken care of by uncompressing the Open Watcom source archive and/or syncing up with Perforce.

3.1 Makeinit

All the magic starts with *makeinit.* Every development machine must have a *makeinit* file with the following defined therein:

mif_dir: must point to the directory in which the mif project has been installed

For each project with name X you wish to have on the build machine, `X_dir` must be set to the directory containing the project. That is, if you want the code generator on your machine (and who wouldn't?), it is officially named cg (see Project Names below) and so you would define `cg_dir.`

Alternatively, if all of your projects are in directories which correspond to their project names under a common directory, you can set dev_dir and *!include cdirs.mif* in your *makeinit.* This is the recommended setup and default for Open Watcom. You do not have to take any extra action to use it.

Alternatively, you can do the above and then redefine `X_dir` for any projects which are not under the `dev_dir.`

3.2 Project Names

Each project must be given a unique name, which should also be a valid directory name under FAT file systems (8.3 convention).

3.3 Makefiles

Each makefile should be located in the object file directory - ie. no more of this silly cd'ing into the object directory based on crystal-balls and what not. The makefile must define the following:

host_os: os which the resulting executable code will run on

host_cpu: architecture which the resulting executable code will run on.

proj_name: the project name

Valid values for `host_cpu` are 386, i86, axp, mps, ppc, x64. These should be self-explanatory. Valid values for `host_os` are dos, nt, os2, nov, qnx, win, osi, linux. These should be self-explanatory for the most part, with one possible exception: osi stands for OS Independent, the executables can run on multiple OSes if appropriate loader stub is provided.

The makefile must then include *cproj.mif*. This will define all sorts of make variables, which can then be used to build the project. A list of the most important of these variables and what they can be used for is included below.

A makefile should also include *defrule.mif*, which has the default build rules for C, C++ and assembly sources, and *deftarg.mif*, for definition of the required clean target.

A makefile is free to override these defaults as long as it follows the following conventions:

1. Tools which have macros defined for them must be referred to by the macros - these are currently (any additions should be brought to my attention):

   $(CC): The C compiler

   $(CPP): The C++ compiler

   $(LINKER): The linker

   $(LIBRARIAN): The librarian

   $(AS): The assembler, if applicable

   $(RC): The resource compiler

   $(EDIT): Our VI editor

   $(YACC): Our version of yacc

   $(RE2C): The regular-expression to C compiler

2. When referring to other projects, a makefile should use the `X_dir` macro, where X is the name of the project.

3.4 Requirements To Build

A project should be able to build either a -d2 (if `$(proj_name)_release` != 1) or releaseable (if `$(proj_name)_release` == 1) executable providing the following are done:

- the project is uptodate and `$(proj_name)_dir` is set correctly
- the mif project is uptodate and make knows to look for .mif files in there
- all depended upon projects are uptodate and have `$(proj_name)_dir` set correctly
- all depended upon projects have been built
- any required executables from under *build/bin* are in the path

Note that there are no other requirements here — it is very annoying when a project requires you to define handles for tools, create directories in which it can deposit stuff, scrounge up obscure tools from who knows where or pretend to be Jim Welch in order to get a debuggable version of the executable.

There is more than one way to switch between development and release build. A `OWDEBUGBUILD` environment variable provides global control. When set to 1, debug builds are produced, otherwise release builds are created. When building individual projects with wmake, it is also possible to give the *release* macro on the wmake command line (0 means debug build, 1 means release build).

Perhaps it should be noted that "releasable" build still contains debugging information, but only at the -d1 level and in a separate .sym file. In case of crashes or other highly unusual behaviour, release build should be enough to point you in the right direction but usually not sufficient to fully diagnose and rectify the problem.

Now, if you wish to allow certain abberant behaviours based upon cryptic make variables, that is fine, as long as the project can build both a debuggable (ie full -d2) version as well as a release (ie no -d2, -d1 only and no memory tracker) version without these things being set. That is, if you want stupid stuff in your *makeinit* — fine, but don't require others to do this in order to build the project.

Any non-standard makefile variables which you do use should be prepended by the name of your project and an underscore, to prevent namespace clutter and clashes.

Tools required to build are an issue that will have to be handled on a case-by-case basis. For example, stuff to bind up DOS protected mode apps will likely be added to the standard suite of tools available, and macros made for them. Before we do this, we should standardize on one extender and use it wherever possible. Any small special purpose tools should be checked in along with the project and built as part of the build process (so that we don't have to check in zillions of binaries for all supported platforms). An important future consideration will be the ability to build on a different architecture. Please try and avoid weirdo tools that have no hope of running on an Alpha or PPC running NT or on Linux. These tools should be referenced from the makefile as `$(bld_dir)\tool.` If your tool cannot run under a particular OS, you should at least put a batchfile in that bin which echoes a message to that effect (to alert people to the fact that you've just made their life difficult). More general tools (yacc, re2c, w32bind) that are likely to be used by several projects should be copied up into the build/bin directory.

3.5 The Runtime DLL Libraries

If you set `$(proj_name)_rtdll` = 1, the -br switch should be thrown for you automatically, providing the target os supports it.

3.6 Memory Trackers

The memory tracker is an useful development aid — it tracks all dynamic memory allocations and deallocations, making it easy to spot memory leaks and helping to pinpoint heap corruption or other unsociable behaviour that so rarely happens in our code.

If the memory tracker is an optional part of your project, and independant of the release mode, it is suggested that you enable it if `$(proj_name)_trmem` is set to 1, and disable it otherwise.

The source to the memory tracker can be found in *bld\trmem.*

3.7 The Clean Target

Each makefile should support a clean target. This should not be part of the default target list, and should delete every makefile generated file. Which means that after running "wmake clean", the directory should look exactly like a new installation of the project on a bare drive. !including *deftarg.mif* should do for most people who do not get creative with file extensions or generated source files. If you do get creative, you may still use the default clean rule if you define the `additional_cleanup` macro that will identify your fancy file names and/or extensions.

Do not underestimate the importance of proper cleanup. It guarantees that every part of a project can be built from scratch, ensuring that there will be no nasty surprises when stuff breaks for people after a clean install just because you had a generated file hanging around and never discovered that it can no longer be made.

3.8 Pmake Support

Every makefile should contain a pmake line at the top. Pmake is a tool which was invented in order to make life easier with the clib project — most people are not interested in building all 40+ versions of the clib when they're working on just one version. Pmake, when run from a root directory, crawls down all subdirectories looking for files called *makefile.* When it finds one, it checks to see if there is a wmake comment which looks like:

```
#pmake: <some identifiers>
```

If there is such a comment, and any identifiers in the list given to pmake appear in the list after the colon, then wmake is invoked in that directory. This provides a handy way to control selective builds and destroys. Some tokens which should be used by the appropriate makefiles are:

all	is implicit in every makefile and does not need to be listed explicitly
build	indicates that wmake should be run in this directory as part of the build process
os_x	for each x in the list of the valid host_os tokens (os_nt, os_dos, etc)
cpu_x	for each x in the list of the valid host_cpu tokens (cpu_386, cpu_ppc, etc)
target_x	for each x in the list of valid host_cpu tokens (for compilers and targetted apps)

tiny, small, compact, medium, large, huge, flat, nomodel
the memory model

inline, calls whether an app uses inline 8087 stuff or fp calls

For example, an executable which is going to run on the PPC version of NT should have a pmake line which contains, at a minimum:

```
#pmake: build os_nt cpu_ppc
```

Pmake also supports the concept of priority. The priority is specified as /nnn after the #pmake but before the colon (:) like so:

```
#pmake/50: build os_nt cpu_ppc
```

Makefiles with lower priority are visited first. The default priority if not explicitly specified is 100. Pmake will visit subdirectories in depth first traversal order unless changed by the *-r* option or the *priority* value.

You are free to add as many mnemonic identifiers as you want, of course, but anything which you feel is an abstract classification that would apply to other projects, please bring to our collective attention and if deemed useful, it will get added to the appropriate places (and the list above).

For an example of where this is useful, if we suddenly found out that our NT headers were bogus and everything including them needed a recompile, we could do the following on the build machine: "pmake os_nt -h clean & pmake os_nt -h".

Another very useful property of this setup is that it allows people to build libraries/binaries only for their host platform. This is especially convenient if they don't have all the necessary SDKs, Toolkits and whatnot installed and/or cannot run some or all of the platform specific tools required during builds. And this situation is the norm rather than exception — only dedicated build servers usually have all necessary files in place.

3.9 Misc Conventions

To make it easy to see what projects are required to make a given project, all needed projects should be listed in a makefile comment in the main makefile of the dependant project. Hopefully, this main makefile should be called *master.mif* and be in the root directory, or a mif subdirectory thereof, of the project.

Also, it is suggested that the object file directory name be a combination of the `host_os` followed by the `host_cpu,` if convenient. For example, NT versions for the PPC should be genned into a ntppc directory. If a directory structure which is different than this is used for some reason, then comments explaining exactly what is built where would be nice in the *master.mif* file.

Things get more interesting if cross compilers are thrown into the mix. In that case three components are required in the name: for instance a *ntaxp.386* directory can hold the Alpha AXP NT compiler producing 386 code.

This is also why the macro names are somewhat counterintuitive — most people would think of the `host_os` and `host_cpu,` as target OS and CPU. However, the 'target' designation is reserved for the target architecture of the generated binary. In the above case of a compiler that runs on Alpha AXP NT and produces 386 code, the makefile contains:

```
host_os    = nt
host_cpu   = axp
target_cpu = 386
```

3.10 DLLs and Windowed Apps

Set `host_os` and `host_cpu` as normal, and then, if creating a windowed app, set `sys_windowed` = 1. If creating a DLL, set `sys_dll` = 1. Delightfully simple.

3.11 Include Paths

The `inc_path` macro is composed of several other variables. Projects are able to hook any of these variables by redefining them after *cproj.mif* is included. The current structure looks like this:

```
inc_path      = inc_dirs | inc_dirs_$(host_os) | inc_dirs_sys
inc_dirs_sys  = inc_dirs_lang | inc_dirs_sys_$(host_os)
```

$(inc_dirs_lang) contains headers delivered with the compiler.

$(inc_dirs_sys_$(host_os)) contains OS specific headers typically delivered by OS vendor.

So, a project should put any include directories it needs into *inc_dirs* — note that this does not include `$(watcom_dir)\h` which is part of the default include directory set.

If it needs to, a project can override any and all of these — for instance, the clib needs to be built with the next release header files, and so would redefine `inc_dirs_lang.`

Any OS-specific header files needed by the project can be set in `inc_dirs_$(host_os)` — again, this should not include the standard system header files, which will be defined in `inc_dirs_sys_$(host_os).`

Note that the build system previously used to set the `INCLUDE` environment variable to hold the contents of *inc_dirs* macro. This mechanism is now considered obsolete and should no longer used. Instead, include paths are passed directly on the command line. This also means that all include paths must be prepended with a -I switch, for example:

```
inc_dirs_sys_nt    = -I$(lang_root)\h\nt
```

3.12 Executive Summary

In order to convert a project to this new structure or create a new (and conforming) project, do the following:

1. Create an object file directory for each combination of host_os/host_cpu under your project.
2. Give your project a name, for instance Foo.
3. Create a *master.mif* in the root of your project.

4. Put all the normal makefile gear in this *master.mif*.

5. Add `proj_name` = Foo to the top of master.mif

6. Include the following files (in this order) *cproj.mif, defrule.mif, deftarg.mif* in *master.mif*

7. Add `inc_dirs` = {list of directories, separated by spaces and each prepended with -I, which your project needs in include path - this does not include OS-specific includes (ie *\lang\h\win*)}

8. Add `extra_c_flags` = {list of c flags, not including optimization, -w4, -zq. -we and memory model info, needed to compile your application} These should be host_os/host_cpu independent.

9. Add `extra_l_flags` = {list of linker directives, not incuding system or debug directives} Should be host_os/host_cpu independent.

10. Use following to compile: `$(cc) $(cflags)` filename etc...

11. Use following to link: `$(linker) $(lflags)` file { list of obj files }

12. Use following to create libraries: `$(librarian)`

13. In each object file directory, create a makefile which looks like the following:

```
#pmake: build os_x cpu_y
host_os  = x
host_cpu = y
!include ../master.mif
```

That's it! The only downside is that sticking to these guidelines will make everyone's life less exciting.

4 Technical Notes

4.1 32-bit Windows run-time DLLs

Most of Open Watcom run-time Windows DLLs have predefined loading address. Bellow is table with address for each DLL.

```
0x69000000  wppdxxxx.dll  (C++ compiler)
0x69400000  wccdxxxx.dll  (C compiler)
0x69800000  wrc.dll       (Resource compiler)
0x69900000  wr.dll        (Resource library)
0x69c00000  wlinkd.dll    (Linker)
0x6a000000  wlibd.dll     (Librarian)
0x6e800000  javavm.dll    (Debugger DIP)
0x6e900000  all trap dlls (Debugger TRAP)
0x6eb00000  madx86.dll    (Debugger MAD)
0x6ec00000  export.dll    (Debugger DIP)
0x6ed00000  codeview.dll  (Debugger DIP)
0x6ee00000  watcom.dll    (Debugger DIP)
0x6ef00000  dwarf.dll     (Debugger DIP)
0x6fa00000  wrtxxxx.dll   (run-time DLL combined C, math and C++
library)
0x6fd00000  plbxxxx.dll   (run-time DLL C++ library)
0x6fe00000  clbxxxx.dll   (run-time DLL C library)
0x6ff00000  mtxxxx.dll    (run-time DLL math library)
```

You shouldn’t use these addresses for your own DLLs.

5 Build Process

We use the (Open) Watcom C/C++ compilers and Watcom *wmake* to build our tools, but at the top level we have a custom tool which oversees traversing the build tree, deciding which projects to build for what platforms, logging the results to a file, and copying the finished software into the release tree (rel), making fully automated builds a possibility. If nothing goes wrong that is.

5.1 Builder

This wondrous tool is called *builder*. You can see *bld\builder\builder.doc* for detailed info on the tool and/or look at the source if the documentation doesn't satisfy you.

So how does builder work? Each project has a *lang.ctl* builder script file. If you go to a project directory and run builder, it will make only that project; if you go to *bld* and run builder, it will build everything under the sun. The overall build uses *bat\lang.ctl* which includes all of the individual project *lang.ctl* files that we use. Note that if you run builder, it will traverse directories upwards until it finds a *lang.ctl* (or it hits the root and still doesn't find anything, but then you must have surely done something wrong). Results are logged to *build.log* in the current project directory and the previous *build.log* file is copied to *build.lo1*. The log file contains captured console output (both stdout and stderr).

Common commands:

builder build — build the software

builder rel — build the software, and copy it into the "rel" release tree

builder clean — erase object files, executables, etc. so you can build from scratch

5.2 Pmake

Many of the projects use the "pmake" features of builder (see *builder.doc*) or standalone pmake tool. If you want to see its guts, the *pmake* source is in *bld\pmake.*

Each makefile has a comment line at the top of the file which is read by *pmake.* Most of our *lang.ctl* files will have a line similar to this:

```
pmake -d build -h ...
```

this will cause *wmake* to be run in every subdirectory where the makefile contains `"build"` on the `#pmake` line. See for instance the C compiler makefiles (in *bld\cc)* for an example.

You can also specify more parmeters to build a smaller subset of files. This is especially useful if you do not have all required tools/headers/libraries for all target platforms.

For example:

```
builder rel os_nt
```

will (generally) build only the NT version of the tools.

A word of warning: running a full build may take upwards of two hours on a 1GHz machine. There is a LOT to build! This is not your ol' OS kernel or a single-host, single-target C/C++ compiler.

It is generally possible to build specific binaries/libraries by going to their directory and running *wmake*. For instance to build the OS/2 version of *wlink* you can go to *bld\wl\os2386* and run *wmake* there (note that the process won't successfully finish unless several required libraries had been built). Builder is useful for making full "release" builds while running *wmake* in the right spot is the thing to do during development.

Happy Building!

6 Testing

There is undoubtedly a question on your mind: Now that I have the Open Watcom compilers, libraries and tools built, what do I do next? The answer is simpler than you may have expected: Ensure that what you built actually works.

Fortunately there is a number of more or less automated test available in the source tree. Currently these tests are not part of the build process per se, although that might (and perhaps should) change in future.

There are two major classes of situations when the tests should be run:

- After building on a fresh system for the first time, running (and passing) the tests verifies that what was built behaves at least somewhat as expected. In this case it might be prudent to run as many tests as possible, especially when building on a new, not yet widely tested platform.

- After making modifications to a particular tool or library, run the appropriate tests exercising the component (if available) to ensure that the changes didn't cause any serious regressions.

If a bug is discovered and fixed, it is a good practice to code up a simple test verifying the fix. That way we can prevent (or at least expediently discover) regressions in future builds. In other words, we won't be embarrassed by the same bug cropping up again. Just because commercial compiler vendors occasionally have this problem doesn't mean we have to as well!

Passing the automated tests can never completely guarantee that everything works perfectly as designed, but it does let you sleep easier at night, comfortable in the knowledge that there aren't any really major problems.

6.1 Running the tests

This section maps the major test nests and gives brief description on how to run the tests and how they're constructed. There is often a single batch file or script that will build and run all the tests for a given project, the end result being either "all set to go" or "we have a bug infestation problem at location xyz, send out bug swat team immediately".

To make automated testing feasible, the test programs do not require user input (unless they were testing user input of course). Some test programs do their work and then decide whether everything worked as expected or not and output a message to that effect. Other test programs ouput messages mapping their progress as they go and the output is then compared with a file containing the 'good' output. Which method exactly is used depends mostly on what is being tested. When testing error and warning messages printed by the compilers and tools, it is natural to compare the captured output to the expected text. When testing the runtime library for instance, it makes sense for the test program itself to decide whether the function call results are what was expected.

Now we'll go through the projects alphabetically and make a stop whenever there's something interesting to see. Note that not all of the tests are automated, the really extensive tests are however. Being a lazy folk, programmers are likely to bang together an automated test suite if that helps them avoid babysitting the tests.

as	In *bld\as\alpha\test* there are several tests exercising the Alpha AXP assembler, using C wrappers for infrastructure.
aui	Not a real test, nevertheless the sample programs in *bld\aui\sample* are useful in demonstrating and informally testing the user interface library.
brinfo	In *bld\brinfo\test* there is a simple browser information test program.
browser	Tests exercising the class browser are located in *bld\browser\test.*
cg	In *cg\test\far16* there is a test exercising the `__far16` keyword. Real code generator tests are found elsewhere.
clib	The C runtime library tests are located in *bld\clib\qa.* These tests are not terribly comprehensive but they do verify the basic C runtime functionality.
gui	Again not a real test, there is a GUI library sample in *bld\gui\sample.*
ndisasm	Tests for the 'new' disassembler (not many at this point) are located in *bld\ndisasm\test.*
orl	The Object Reader Library tests are in *bld\orl\test.*
plustest	This project holds *the* test suite. Ostensibly designed to exercise the C++ compiler, the tests also verify the functionality of the code generator and some are designed for the C compiler. Running these tests can take a while as there are over a thousand test cases. Highly recommended.
ssl	In *bld\ssl\test* there are several simple test scripts for SSL.
trmem	While the memory tracker is not a test, it bears mentioning here. This can be used for testing many other projects for memory leaks, memory overwrites and other cases of rude behaviour.
viprdemo	Again not a test per se, the 'Viper demo' is a good way to verify basic IDE functionality.
wasm	Extensive assembler test can be found (rather predictably) in *bld\wasmtest.*
wdisasm	Tests for the 'old' disassembler are located in *bld\wdisasm\test.*
wmake	Extensive make utility tests can be found in *bld\wmake\regress.*
wprof	A profiler test program is located in *bld\wprof\test.*
yacc	Several sample/test YACC input files are in *bld\yacc\y.*

Style

7 Programming Style

Programming style is, unfortunately, a religious matter. Many holy wars have been fought over it with no clear result (because the losing side usually survives). Still, with a project the size of Open Watcom (that is, a very big project) there is a clear need for common programming style.

Conformance of all projects to a common style has several benefits. Programmers who know this style will be easily able to find their way around any of the multitude of projects. Various projects will easily fit together. And last but not least, consistent style looks good.

Note: the fact that certain Open Watcom projects do not adhere to the common programming style should not be construed as an endorsement of non-conformance. It is simply a result of the long and varied history of the project.

The following sections examine various aspects of programming practice and give specific guidelines where applicable. Note that these are guidelines, not rules or laws. Violating them is not a crime and not even a mortal sin. In fact, you might have a very good reason not to stick to the guidelines, and we always prefer common sense to fixed rules. However breaking the guidelines with no good reason is bad for your karma. Don't do it!

7.1 Source file structure

First a few words on source file structure. Every source file should start with a copyright header. This only applies to source and include files (regardless of programming language used). Other types of files such as makefiles, scripts, etc. do not need a copyright header. The header should also contain a short description of the source file, one or two lines is usually enough. Longer comments explaining specifics of the implementation should be placed after the header.

The rest of the source file structure depends on the language used. Here we will only examine the most common kind, which is a C source file. The overall structure is as follows:

- copyright header
- ***#include*** directives
- function declarations and global variable definitions
- function implementation

As you can see, nothing fancy. Many programmers prefer to order functions so as to minimize forward declarations, ie. a `main()` function would be located at the end and every function fully defined before it is first used.

You can use ***extern*** declarations but you should be very careful. It is strongly encouraged that all declarations of external functions and variable be located in header files which are included in modules that use them as well as modules that define them. The reason is simple — this way the compiler will tell you if

you change the definition but not the header file. If you use ad-hoc ***extern*** declarations, you better make sure they're in sync with the actual definitions.

7.2 Help the compiler and it will help you

While the compiler is a rather sophisticated piece of software, it cannot divine your intentions. Hence you have to help it a bit by giving hints here and there. And if you help the compiler in this way, it will be better able to help you.

First of all, always compile with maximum warning level. Just because a message is called a warning and not error doesn't mean you can ignore it. In fact many projects treat warnings as errors and will not build if any warnings are present. That is a Good Thing.

Use the ***static*** keyword often because it is good. This is a multi-faceted keyword and its exact semantics depend on where it is applied:

globals here the ***static*** modifier does not change the storage class but makes the variables local to the module where they are defined, that is they won't be visible from other modules. If a variable needs to be global but doesn't have to be accessed from other modules, you should definitely make it static. That way you will not needlessly pollute the global namespace (reducing the chance of name clashes) and if you stop using the variable, the compiler will warn you. If you have a non-static global variable that is unused, the compiler cannot warn you as it has to assume that it is used from other modules.

locals in this case the ***static*** keyword changes the storage class and the variable will be stored in data segment instead of on stack. If you need a variable that has global lifetime but only needs to be accessible from one function, making it local and static is a good choice.

functions the effect of the ***static*** keyword here is similar to the global variables. The function will not be visible from other modules. Again, the compiler will warn you if you stop using such a function. But declaring a function static also helps the optimizer. For instance if the optimizer inlines a static function, it can completely remove the non-inlined version. If the function weren't static, the non-inlined version always has to be present for potential external callers.

Similar in vein to the ***static*** keyword is the ***const*** keyword. Again it can be applied in various ways and it helps the compiler and optimizer by giving it hints about how you intend to use a particular variable or argument. Saying that a variable is constant is essentially the same as saying that it's read-only. The compiler/linker might place such variable in a read-only segment. While it is possible to circumvent the ***const*** modifier by taking the address of a constant variable and modifying it through a pointer, this is a bad thing to do and it may not work at all because the variable might be not be physically writable. It is perhaps worthwhile to remark that there are three possible outomes of applying the ***const*** modifier to a pointer:

- a constant pointer, that is the value of the pointer is constant but the data it points to isn't
- a pointer to a constant, that is the pointer itself is not constant but the value it points to cannot be modified through it
- a constant pointer to a constant, that is both the pointer itself and the value it points to are constant.

The ***const*** keyword is especially useful when used in function declarations. Consider the following typical declaration:

```
char *strcpy( char *s1, const char *s2 );
```

Here we have a function which takes two arguments that are both pointers to char but one of them is a pointer to a constant. In the function body the compiler will not let you modify the contents of `*s2` but the declaration is also important for the caller. In the calling funtion, the optimizer now knows that the data the `s2` argument points to will not be modified by `strcpy()` and it can take advantage of this knowledge.

Documentation

8 Producing Documentation

The purpose of this document is twofold: to provide an overview of the Open Watcom documentation system togehter with the steps necessary for editing and producing online or printed documents, and at the same time serve as an example of the documentation system usage.

It is useful to note that the online documentation is almost, but not quite, independent of the rest of the Open Watcom compilers and tools. One important exception is online help files for Open Watcom GUI tools. Formatting online documentation generates include files containing symbolic constants designating help entries. These are used during building of the tools binaries. If the binaries are not built with the right header files, the online help will be out of sync and not all that helpful.

There's one other link going in the other direction: certain documentation files live with their respective projects and not in the documents tree. This is especially true for error message documentation for the compilers and tools.

8.1 Setting up

A Win32 or OS/2 system can be used to produce most of the documentation. OS/2 Warp is required for the final step in producing the OS/2 online help files and Win32 system is needed for producing Windows help files (unless you can run the required help compilers on your host platform). DOS may work for producing some of the documentation but is untested at this time.

The environment variable `doc_root` must point to the root of the docunmentation tree. Add `%doc_root%\cmds` to your `PATH`. Your `PATH` must also contain the Open Watcom C/C++ binary directories appropriate for your host platform (for wmake). This is taken care of automatically by using `setvars.cmd/setvars.bat`.

Note that to produce Windows and/or OS/2 online documentation, you will need the appropriate SDKs and Toolkits containing the platform specific online help compilers.

8.2 Building PostScript Documentation

Here are the steps to formatting a book for printing on a PostScript printer.

```
cd %doc_root%\ps
wmake hbook=<bn>
```

where `<bn>` is one of

```
devguide    Developer's Guide (this document)
c_readme    C/C++ Read Me First
cguide      C/C++ User's Guide
cguideq     C/C++ User's Guide for QNX
clib        C Library Reference (for all systems except QNX)
clibqnx     C Library Reference for QNX (stale)
cpplib      C++ Class Library Reference
ctools      C/C++ Tools User's Guide
cw          CauseWay User's Guide
f77graph    F77 Graphics Library Reference
f_readme    F77 Read Me First
fpguide     F77 Programmer's Guide
ftools      F77 Tools User's Guide
fuguide     F77 User's Guide
guitool     Graphical Tools User's Guide
ide         IDE User's Guide
lguide      Linker User's Guide
pguide      C/C++ Programmer's Guide
wd          Debugger User's Guide
```

The output file is of type `.ps.` You should be able to send this file to any PostScript printer or view it in GhostScript or convert it to PDF or do whatever it is you do with PostScript files.

8.3 Building Online Help Documentation

For Microsoft Help format (old Windows 3.x help format):

- Switch to the appropriate directory:

  ```
  cd %doc_root%\win
  ```

- Run `wmake` to create all online help.

- Note that you must have the Microsoft Help Compiler (HC) installed.

For Microsoft Help format ("new" Windows NT/95 help format):

- Switch to the appropriate directory:

  ```
  cd %doc_root%\nt
  ```

- Run `wmake` to create all online help.

- Note that you must have the Microsoft Help Compiler (HCRTF) installed.

For Watcom Help format (for the WHELP command):

- Switch to the appropriate directory:

  ```
  cd %doc_root%\dos
  ```

- Run `wmake` to create all online help.

For OS/2 Help format:

- Switch to the appropriate directory:

```
cd %doc_root%\os2
```

- Run `wmake` to create all OS/2 online help.

- Note that this will only work on an OS/2 system with the IBM IPF Compiler (IPFC) installed.

To format one document at a time, go to the appropriate directory (for instance `docs\os2`) and run wmake with argument `hbook=<book_name>` where `<book_name>` is one of the online books listed above.

8.4 Editing the Documentation

All the documentation is stored in ASCII text files with the file extension "GML". The files are annotated with a combination of Script and GML (Generalized Markup Language) tags.

The Script tags are of the form ".tag" (i.e., they begin with a period and are followed by two or more letters or digits). Script tags will be most familiar to anyone who has ever used Waterloo Script or IBM Script. The tagged format is also similar in idea to other tagged formatting systems like RUNOFF or ROFF.

The GML tags are of the form ":TAG." (i.e., they begin with a colon, followed by some letters and digits and end with a period). GML tags will be most familiar to anyone who has ever used IBM GML or Waterloo GML. This tag set is a variant of SGML. The most familiar SGML tag format is `<TAG>.` In Watcom GML, the "<" and ">" are replaced by the ":" and ".". If you know HTML, you know how tags work — HTML is just another variant of SGML.

The tag set includes a base set of predefined tags. In addition to this base set, you can define an extended tag set using the built-in macro language. The base Script tag set employs two letters (e.g., two, three, four or more letters (e.g. .chapter, .section, .beglevel). For a good example of user-defined Script tags, see `%doc_root%\doc\gml\fmtmacro.gml.` GML tags can also be defined. For a good example of user-defined GML tags, see `%doc_root%\doc\gml\cppextra.gml.`

These tags are described here for you, not so that you can begin defining your own tags, but so that you will recognize them in the ASCII text that comprises the documentation. But of course no-one's stopping you from defining your own tags should you feel so inclined.

Here's a snippet from one of the doc files.

```
    .np
    The recommended options for generating the fastest 16-bit Intel
code
    are:
    .ix 'fastest 16-bit code'
    .begnote
    .note Pentium Pro
    -oneatx -oh -oi+ -ei -zp8 -6 -fpi87 -fp6
    .note Pentium
    -oneatx -oh -oi+ -ei -zp8 -5 -fpi87 -fp5
    .note 486
    -oneatx -oh -oi+ -ei -zp8 -4 -fpi87 -fp3
    .note 386
    -oneatx -oh -oi+ -ei -zp8 -3 -fpi87 -fp3
    .note 286
    -oneatx -oh -oi+ -ei -zp8 -2 -fpi87 -fp2
    .note 186
    -oneatx -oh -oi+ -ei -zp8 -1 -fpi87
    .note 8086
    -oneatx -oh -oi+ -ei -zp8 -0 -fpi87
    .endnote
    .np
    The recommended options for generating the fastest 32-bit Intel
code
    are:
```

The ".np" is a user-defined tag for "start a new paragraph". The ".ix" creates an index entry in the index. It doesn't appear with the text. In on-line help, this index entry becomes a searchable item. The ".begnote", ".note", and ".endnote" user-defined tags are used to create an unordered list. Every piece of text entered into the source file is identified by tags like these.

The best way to understand what the tags do is to look at a printed copy of the document and see what it looks like. Luckily for you, you don't have to look very far:

The recommended options for generating the fastest 16-bit Intel code are:

Pentium Pro -oneatx -oh -oi+ -ei -zp8 -6 -fpi87 -fp6

Pentium -oneatx -oh -oi+ -ei -zp8 -5 -fpi87 -fp5

486 -oneatx -oh -oi+ -ei -zp8 -4 -fpi87 -fp3

386 -oneatx -oh -oi+ -ei -zp8 -3 -fpi87 -fp3

286 -oneatx -oh -oi+ -ei -zp8 -2 -fpi87 -fp2

186 -oneatx -oh -oi+ -ei -zp8 -1 -fpi87

8086 -oneatx -oh -oi+ -ei -zp8 -0 -fpi87

The recommended options for generating the fastest 32-bit Intel code are:

The WATCOM GML program (WGML) is a compiler/interpreter that reads the document's source files to produce an output file. In our case, we want PostScript for printing and we want another form for generation of online help. This second form is a non-printable form that is suitable for post-processing to

turn it into IPF for the OS/2 IPF compiler, RTF for the WinHelp tools, special Watcom-defined format for use with a DOS-based help tool (WHELP) or the ever-popular HTML.

If you are a programmer, and that is likely, you'll be somewhat comfortable with the whole process of turning ASCII text into documentation. WGML is a text processor (compiler) that reads a source file (GML) which, in turn, imbeds other source files, and produces an output file (the object file). WGML is very fast. It was very fast in the old 20MHz 386 days and is, of course, much faster with today's processors. The C Library Reference comprising 1,241 pages takes one minute to format into PostScript on a 600 MHz Pentium-III.

If you ever used TeX or LaTeX you will be comfortable with the concept of nonvisual content-driven formatting. If you only know so-called WYSIWYG word processors heavily relying on visual formatting, you might be surprised to find that it is possible to let the computer do lot of the hard work. Just give up the idea of controlling every pixel — it never works right anyway. Instead of saying "this is Arial 10pt Bold" you will say "this is a keyword" or "this is a code example" and let the machine worry about formatting.

8.5 Diagnostic Messages

If you see `***WARNING***` messages issued by WGML, you can ignore them. Of course it is better if you don't and correct whatever is causing the warnings. If you see `***ERROR***` messages, you cannot ignore them and have to fix them before any output is produced.

#

B

C

D

E

F

H

I

M

N

P

S

T

V

W