Rational Software Corporation®

RATIONAL® CLEARCASE®

OMAKE GUIDE

WINDOWS EDITION

VERSION: 2002.05.00 AND LATER

PART NUMBER: 800-025066-000

Rational

the software development company

OMAKE Guide
Document Number 800-025066-000   October 2001
Rational Software Corporation 20 Maguire Road Lexington, Massachusetts 02421

Contents

Tables

Preface

The ClearCase **omake** software build program provides compatibility with Windows NT-based **make** tools.

About This Manual

This manual is for users who are new to the **omake** tool and who may have other experience with **make** tools. It provides information on setting up **omake**, gives an overview of building programs using **omake**, and describes **omake** usage. The appendices contain information on error messages, macros and rules, compatibility, and regular expressions.

If you are already familiar with the build process and makefiles, the recommended sequence for proceeding through the documentation is:

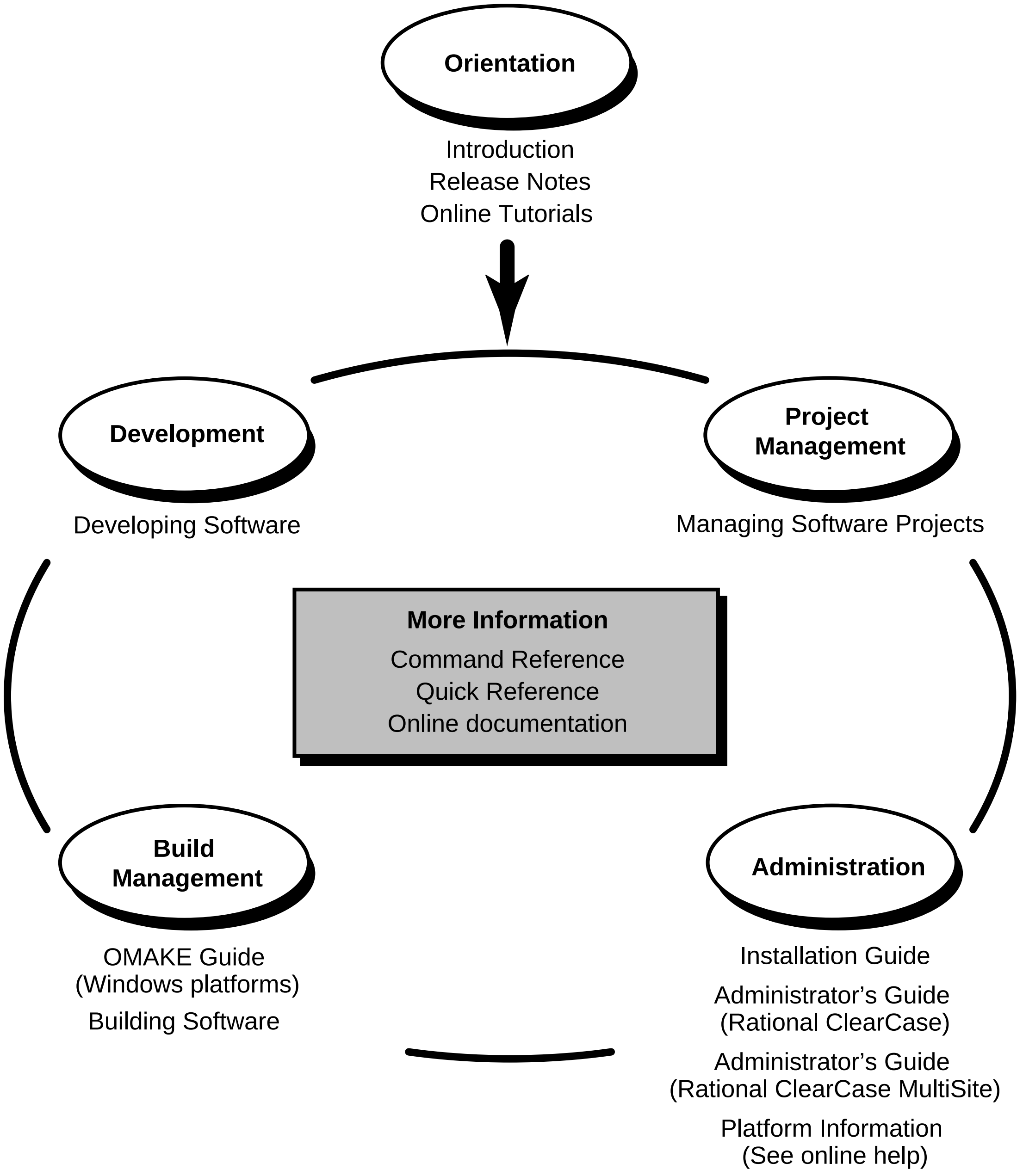
- ➤ Read Chapter 1, *Introduction*, for configuration and usage information
- ➤ Read the first section of Chapter 2, *Overview of Using omake*
- ➤ Refer to Chapter 3, *omake Reference*, for information on **omake**-specific features
- ➤ Read the remaining chapters of this book

If you are not familiar with the build process and makefiles, the recommended sequence for proceeding through the documentation is:

- ➤ Read Chapter 2, *Overview of Using omake*
- ➤ Read Chapter 1, *Introduction*, for configuration and usage information

Building Software provides additional information about using the **clearmake** and **omake** build tools.

ClearCase Documentation Roadmap



Typographical Conventions

This manual uses the following typographical conventions:

➤ *ccase-home-dir* represents the directory into which the ClearCase Product Family has been installed. By default, this directory is **/usr/atria** on UNIX and **C:\Program Files\Rational\ClearCase** on Windows.

➤ *attache-home-dir* represents the directory into which ClearCase Attache has been installed. By default, this directory is **C:\Program Files\Rational\Attache**, except on Windows 3.*x*, where it is **C:\RATIONAL\ATTACHE**.

➤ **Bold** is used for names the user can enter; for example, all command names, file names, and branch names.

➤ *Italic* is used for variables, document titles, glossary terms, and emphasis.

➤ `A monospaced font` is used for examples. Where user input needs to be distinguished from program output, **bold** is used for user input.

➤ Nonprinting characters are in small caps and appear as follows: **<EOF>**, **<NL>**.

➤ Key names and key combinations are capitalized and appear as follows: SHIFT, CTRL+G.

➤ [ ] Brackets enclose optional items in format and syntax descriptions.

➤ { } Braces enclose a list from which you must choose an item in format and syntax descriptions.

➤ | A vertical bar separates items in a list of choices.

➤ ... In a syntax description, an ellipsis indicates you can repeat the preceding item or line one or more times. Otherwise, it can indicate omitted information.

NOTE: In certain contexts, ClearCase recognizes "**...**" within a pathname as a wildcard, similar to "*" or "?". See the **wildcards_ccase** reference page for more information.

➤ If a command or option name has a short form, a "medial dot" (·) character indicates the shortest legal abbreviation. For example:

lsc·heckout

This means that you can truncate the command name to **lsc** or any of its intermediate spellings (**lsch**, **lsche**, **lschec**, and so on).

Online Documentation

The ClearCase graphical interface includes a standard Windows help system.

There are three basic ways to access the online help system: the **Help** menu, the **Help** button, or the F1 key. **Help** > **Help Topics** provides access to the complete set of ClearCase online documentation. For help on a particular context, press F1. Use the **Help** button on various dialog boxes to get information specific to that dialog box.

ClearCase also provides access to full "reference pages" (detailed descriptions of ClearCase commands, utilities, and data structures) with the **cleartool man** subcommand. Without any argument, **cleartool man** displays the **cleartool** overview reference page. Specifying a command name as an argument gives information about using the specified command. For example:

> **cleartool man** *(display the cleartool overview page)*

> **cleartool man man** *(display the cleartool man reference page)*

> **cleartool man checkout** *(display the cleartool checkout reference page)*

ClearCase's **–help** command option or **help** command displays individual subcommand syntax. Without any argument, **cleartool help** displays the syntax for all **cleartool** commands. **help checkout** and **checkout –help** are equivalent.

```
> cleartool uncheckout –help
Usage: uncheckout | unco [-keep | -rm] [-cact | -cwork ] pname ...
```

Additionally, the online *ClearCase Tutorial* provides important information on setting up a user's environment, along with a step-by-step tour through ClearCase's most important features. To start the *ClearCase Tutorial*, choose **Tutorial** in the ClearCase folder off the **Start** menu.

Technical Support

If you have any problems with the software or documentation, please contact Rational Technical Support via telephone, fax, or electronic mail as described below. For information regarding support hours, languages spoken, or other support information, click the **Technical Support** link on the Rational Web site at **www.rational.com**.

Your Location	Telephone	Facsimile	Electronic Mail

North America	800-433-5444 toll free or 408-863-4000 Cupertino, CA	408-863-4194 Cupertino, CA 781-676-2460 Lexington, MA	**support@rational.com**
Europe, Middle East, and Africa	+31-(0)20-4546-200 Netherlands	+31-(0)20-4546-201 Netherlands	**support@europe.rational.com**
Asia Pacific	61-2-9419-0111 Australia	61-2-9419-0123 Australia	**support@apac.rational.com**

1 Introduction

omake is a programming tool that helps you maintain programs, particularly those that are constructed from several component files. **omake** controls the entire process of building your program, whether it involves preprocessing files, compiling, linking, or other steps. It keeps track of the project files, recompiling and relinking them only when required. Moreover, **omake** maintains important information, such as compiler and linker options, in an editable text file called a makefile. If you've ever returned to a project after a long absence, you know how hard is to remember all the parameters that were used the last time the project was built. **omake** remembers everything about the project for you.

Rational ClearCase includes two independent build programs: **clearmake** and **omake**. Both incorporate the major ClearCase build-related features, including *configuration lookup*, *derived object sharing*, and *configuration record* maintenance. The **omake** program’s strength lies primarily in its support for users who require compatibility with other PC-based build programs, including Borland® Make, Microsoft® NMAKE, PolyMake, and Opus Make.

NOTE: **omake** is intended for use in *dynamic views*. You can use **omake** in a *snapshot view*, but none of the features that distinguish it from ordinary make programs — build avoidance, build auditing, derived object sharing, and so on — works in snapshot views. The rest of the information in this manual assumes you are using **omake** in a dynamic view.

1.1 File Manifest

omake includes the following files, which are installed in *ccase-home-dir***bin**:

builtins.*	PVCS Configuration Builder and NMAKE emulation rules and macros
omhelp.*	PVCS Configuration Builder and NMAKE help screens

make.ini	Sample **omake** initialization file
omake.exe	**omake** executable

1.2 Configuring omake

omake reads an initialization file, *ccase-home-dir***bin****make.ini**, which you can configure. The default **make.ini** file gives you a minimal configuration. Installing a new version of ClearCase replaces your **make.ini** file; if you have customized this file and want to maintain your changes after upgrading to a new version of ClearCase, back up the file before installation and restore it afterward from your backup copy.

One of the major features of **omake** is its ability to emulate either PolyMake/PVCS Configuration Builder or Microsoft NMAKE. The emulation is enabled with a command-line option. See *PM/CB Emulation* on page 152 and *NMAKE Emulation* on page 158 for more information. Throughout this document, PM/CB refers to PolyMake/Configuration Builder, and NMAKE refers to Microsoft NMAKE.

1.3 Notes on Using omake

The ClearCase **omake** utility is based on Opus Config Builder 6.10, from OPUS Software, and is compatible with most of its makefiles. It has been extended to include the fundamental ClearCase build features. This section describes the ClearCase extensions and offers some basic usage guidelines. Other sources of help on **omake**:

- **omake** reference page (in the *Command Reference*)
- **omake –h**
- **omake –EN –h** (help on NMAKE compatibility mode)
- **omake –EP –h** (help on PolyMake compatibility mode)

For more information on the ClearCase build model, see *Building Software* and the **clearmake** reference page.

omake’s Configuration Management Features

The features described here constitute the core of ClearCase build configuration management.

Configuration Lookup

ClearCase **omake** uses a build-avoidance scheme based on configuration lookup that is more sophisticated than the conventional **make** and Opus Make schemes, which are based on time stamps of built objects. For example, the **omake** build-avoidance scheme guarantees correct build behavior as C-language header files change, even if the header files are not listed as dependencies in the makefile. By default, **omake** uses configuration lookup when you are working in a *view*, and the target is located in the *MVFS*.

Derived Object Sharing

Developers working in different views can share the MVFS objects created by **omake** builds. See the **clearmake** reference page for details.

Configuration Record Creation

omake audits the build operations, generating software bill-of-materials records, called *configuration records*, that fully document a build. In the process, **omake** also performs automatic *dependency detection*. By default, configurations records are created whenever you run **omake** from a ClearCase view, and derived files are stored in the MVFS. See the **clearmake** and **catcr** reference pages for details.

Express Builds

You can use either **omake** or **clearmake** during an express build, which is a build that creates nonshareable derived objects. Express builds do not write DO information to the VOB, and therefore enhance site-wide performance because they do not block other users' access to the VOB. For more information on express builds, see *Building Software*.

Command-Line Options

➤ The following options have been added to **omake**:

–L	Disables configuration lookup and config record creation. See also **clearmake –F**.
–O	Disables build script checking during configuration lookup.
–v	Increases output verbosity during the build process (particularly with respect to configuration lookup and config record creation). **NOTE:** The Opus Make **–V** option, which prints the version string, has been removed.
–W	Disables shopping for derived objects to wink in. See also **clearmake –V**.
–T	Examines sibling derived objects when determining whether a target object in a VOB can be reused. See also **clearmake –O**.

Location of Temporary Files

The **CCASE_AUDIT_TMPDIR** environment variable controls the location of build audit temporary files. By default, these files are stored in the directory named by the **TMP** environment variable. If you do not set **CCASE_AUDIT_TMPDIR**, make sure that **TMP** is set to a valid temporary storage directory on a FAT, NTFS, or LAN Manager file system. Note that setting **CCASE_AUDIT_TMPDIR** to a location on a remote file system increases build times.

Non-MVFS Dependencies

omake supports local and remote non-MVFS dependencies:

- If the dependency resides on a local drive, you can use the local drive letter. However, to share the applicable derived object, users on other hosts must access the dependency using the same drive-specific pathname.
- If the dependency resides on a remote host, you must use UNC names. (If you use drive letters, only users with identical network drive configurations can share the resultant DOs.)

Differences Between omake and Standard Make Tools

- **$?** means all dependencies, not just out-of date dependencies for targets for which configuration lookup is used.
- Double-colon (**::**) rules. If any instance of a double-colon rule is out of date (based on configuration lookup), all double-colon build scripts for the affected target are executed. Conversely, if an error occurs during a build of the target of a double-colon rule, the target's remaining build scripts are not executed, and no config records or derived objects are constructed. For example:

```
install::foo.exe
   copy foo.exe \inst\bin

install::bar.exe
   copy bar.exe \inst\bin
```

  If **foo.exe** or **bar.exe** is out of date, both build scripts are executed. (Both programs are copied.)

Running omake

Typically, you run **omake** from a view using the following procedure:

1. Set a view context by assigning a drive to a view (with **Tools** > **Map Network Drive** in Windows Explorer or the **net use** command) and changing to that view:

   ```
   c:\> net use f: \\view\myview

   c:\> f:

   f:\>
   ```

2. Change to the appropriate directory and run **omake**:

   ```
   f:\> cd myvob\src                        (`\myvob' is the VOB-tag)

   f:\myvob\src> omake options
   ```

   You are using a view context to prevent VOB pathnames from being dependent on the view the build occurs in. From **f:**, you (and your makefiles) can access versioned objects with non-view-extended, *absolute VOB pathnames* such as **\vob2\src\main.c** in either **cleartool** commands or standard operating system commands.

   If you work directly on **M:**, in view-extended namespace, full pathnames to VOB objects include a view-specific prefix, which can affect configuration lookup so as to prevent DO sharing between views.

Makefiles

There are several rules to follow when constructing, or converting, makefiles for use by **omake** on a Windows NT host.

Supporting Both omake and clearmake

It is possible, but not trivial, to prepare makefiles that can be used with either **omake** or **clearmake**. The general approach is to supply **omake**-specific macro definitions in the makefile, and to supply **clearmake**-specific macro overrides in a build options specification (BOS) file; **clearmake** reads the BOS file, but **omake** does not. When **clearmake** executes, it looks for macro definitions in two locations:

- **%HOME%\.clearmake.options**
- **makefile.options**, in the same directory as **makefile** (substitute the actual name of your makefile, if it is not **makefile**)

BOS files at other locations can be passed to **clearmake** with the **–A** option.

Parallel and Distributed Build Operations

Each **omake** build executes serially on the local Windows NT host. Parallel and distributed build operations are not available.

Build Scripts and the rm Command

It is common for a makefile to include a target whose build script invokes a command such as **rm** to delete files. Some Windows NT installations include **rm** commands that do not actually delete a file, but move it to a deleted directory instead. As a result, build script temporary files become sibling DOs of the targets. To avoid this problem, use a remove command—**del**, for example—that actually deletes files.

Pathnames in CRs

In a config record created on Windows NT, MVFS object pathnames begin with the VOB-tag and do not include view-tag prefixes. For example:

```
...
---------------------------- MVFS objects: ----------------------------
\proj1\include\cmsg.h@@\main\nt3\39          <22-Jul-94.17:49:53>
\proj1\lib\fsutl.h@@\main\12                 <22-Jun-94.12:07:24>
...
```

Pathnames in this format can be cut and copied, and applied elsewhere as is, if you are on a drive assigned to a view (with **Tools** > **Map Network Drive** in Windows Explorer or the **net use** command).

1.4 Auditing 16-bit Tools

Compilers, linkers, and other tools written to run on MS-DOS or Windows (16-bit tools) require special handling when used in audited builds with **omake**.

The program **vdmaudit** allows auditing of 16-bit tools. To use **vdmaudit**, you need to have **omake** run **vdmaudit** and let it call the 16-bit tool. This involves either editing the **makefile** where it calls the 16-bit tool, or if your **makefile** defines a macro for each 16-bit tool, redefining the macros in the **makefile** or on the **omake** command line.

If the makefile defines a macro for each 16-bit tool, you can change the macros to call **vdmaudit**. For example, if your makefile contains macros like

```
CPP=cl.exe
LINK=link.exe
```

change them as follows:

```
CPP=vdmaudit cl.exe
LINK=vdmaudit link.exe
```

You can redefine the macros on the **omake** command line like this:

```
omake  –EN  –f  disptree.mak  CPP="vdmaudit cl.exe"  LINK="vdmaudit link.exe"
```

Call all 16-bit tools from **vdmaudit**; if you do not, **omake** does not audit all tools and the configuration record is incomplete.

An alternative method for auditing 16-bit tools is to use the ClearCase Virtual Device Driver (VDD). To install the VDD during ClearCase installation, select **16-bit build auditing** on the **ClearCase Client Options** or **ClearCase Server Options** page. The VDD runs any time a 16-bit tool is run, whether during an audit or not. However, the VDD can cause 16-bit tools to fail to display all output or to fail to clear the screen when done.

2 Overview of Using omake

This chapter provides an overview of using **omake** to build and maintain programs. It describes the process **omake** uses to build programs, the files you use to control **omake**'s operation, and some selected features of **omake**.

2.1 The omake Process

omake works in two distinct phases:

- *read time*, when it reads the files (called *makefiles*) that provide it with a project description
- *run time*, when it builds the project

A project consists of one or more *targets*. A target is a thing that can be made, usually a file, such as a source file, object file, or executable file. The target is said to exist if the file is present on disk. The *pathname* to a target is its location on disk. The target's *configuration record* (CR) is a software bill of materials record that fully documents the build of the target and supports **omake**'s ability to rebuild. The CR includes the following information:

- Which MVFS file system objects (and which versions of those objects) were used by the build as input data
- The commands used to perform the build
- Which MVFS files were created as output

The target's time stamp is the time the target was last changed. This is usually the creation or modification time of the file.

Reading the makefiles gives **omake** dependency information. A *dependency* is the relationship between a target and the things needed to make it. For example, an executable file that is linked from a set of object files must be relinked if any of the object files change; thus, the executable file depends on the object files. One way to think of this dependency information is as a data tree, which may be several levels deep. For example, an executable **proj.exe** may depend on **main.obj** and **io.obj**, each of which depend on source files **main.c** and **io.c**, respectively.

The things a target depends on are its *dependencies*. In fact, dependencies are also targets because they can be (and are) made.

To make or *build* a target means to bring it up to date. A *build script* (sometimes referred to as a shell line) is a command that **omake** executes to bring a target up to date. For example, a link command updates an executable file. The command returns an exit status, which lets **omake** know whether it succeeded. If the command succeeds, **omake** generates and stores a configuration record. This configuration record is attached to each file modified or created by executing the build script. A file produced by the build and associated with a configuration record is called a *derived object* (DO).

If the target (derived object) is located in the MVFS, **omake** uses configuration lookup to determine whether it is necessary to rebuild a derived object by executing a build script or by reusing an existing instance of the derived object. **omake** traverses the dependency tree and compares the configuration record of each target to the build configuration of the current view (the set of dependency versions, the current build script, and the current build options), rebuilding the target if necessary. For each target, the tree is traversed to the bottom and back up.

If the target is not in a VOB, **omake** compares the time stamp of the target to those of its dependencies. If the target is older than any of its dependencies, **omake** rebuilds it.

Note that if you are working in a *snapshot view,* **omake** does not perform configuration lookup and it does not create derived objects or configuration records. For more information about snapshot views, see *Developing Software*.

NOTE: In this chapter, the examples assume that the files are in a VOB.

2.2 Invoking omake from the Command Line

omake is invoked from the command line with:

omake [*option* | *macro* | *target* | *@file*] . . .

Each *option* is indicated by a **–** (dash) or **/** (slash) followed by a single letter, which is the option name. Options can be grouped; that is, **–k –v** is the same as **–kv**. The case of the option name is important. **omake**'s command-line options are described in *Command-Line Options* on page 41.

Each *macro* is a macro definition of the form *name*=[*value*]. Macros defined on the command line take precedence over macros defined in the makefile. See *Macros* on page 21.

Each *target* is a target you want **omake** to build. If no targets are listed on the command line, **omake** builds the makefile's default target.

Each *@file* directs **omake** to read *file* for additional *options*, *macros*, *targets* and *@files*. The *@file* is treated as if the contents of *file* were on the command line except that each end-of-line character in *file* is treated as a space.

You do not have to group options, macros, or targets on the command line; you can intermix them. However, **omake** reads makefiles in order of their placement on the command line, and builds targets in order as well. If the same macro or parameter is defined more than once on the command line, the last definition is the one **omake** uses.

2.3 Initialization Files and Makefiles

omake reads its instructions from an initialization file, which holds instructions used to customize the general operation of **omake**, and from a *makefile*, which has instructions for a specific project.

The Initialization File

The initialization file contains instructions for all make programs and is used to customize the operation of **omake**. **omake** reads the initialization file, *ccase-home-dir***bin\make.ini**, whenever it starts up.

The initialization file contains targets, inference rules, and macro definitions that you do not want to duplicate in every makefile, and it is used to customize the operation of **omake**. The initialization file is processed like a makefile, with the following exceptions:

- The first target in the first makefile is the default target and is built if no command-line targets are specified. There is no special significance to the first target in the initialization file.

- Targets defined in the initialization file are redefined by targets of the same name that are defined in the makefile. That is, these targets are available to all make programs, but they can be redefined in the makefile.

The Makefile

The *makefile* has instructions for a specific project. The default name of the makefile is literally **makefile**, but you can specify a different name with the **–f** option. The **–f** *file* command-line option names the makefile, and several **–f** options can be given; each makefile is read in the order it appears on the command line. If there are no **–f** command-line options, **omake** tries the file named **makefile**. If it doesn't exist, **omake** tries **makefile.mak**.

Continued Lines

Lines in the makefile can be very long; the total makefile line length is limited only by available memory. For easier reading, a long line can be broken up by typing **\<ENTER>** after part of the line and typing the rest of the logical line on the next physical line of the makefile. For example, the line

```
first_part_of_line second_part_of_line
```

is the same as

```
first_part_of_line \
second_part_of_line
```

To have the backslash (\) as the last character on the line, use **\\<ENTER>** or **\<SPACE>,<ENTER>**.

Comments

The simplest makefile statement is a comment, which begins with the comment character (**#**). Everything on a makefile line after the **#** is ignored. Use **\#** for a literal **#** character. The following large comment may appear in a makefile to describe its contents:

```
#
# Makefile for omake
#
# Compiler: Microsoft Visual C++ 5.0
# Linker: Microsoft(R) 32-Bit Incremental Linker Version 5.00.7022a
#
```

The comment character can also be used at the end of another makefile statement:

```
some makefile statement          # a comment
```

If **\<ENTER>** appears on a commented line, the comment acts until the end of the line and the following line is still continued. The following examples are equivalent:

```
line_one \
line_two # more_line_two \
line_three

line_one line_two line_three
```

Contents of Initialization Files and Makefiles

With a few exceptions, the initialization file holds the same kind of information as a makefile. Both the initialization file and makefiles consist of these components:

- Rules
- Dependency lines
- Build scripts
- Macro definitions
- Inference rules
- Response files
- Directives

These components are described in the following sections.

2.4 Rules

A rule tells **omake** both when and how to make a file. For example, suppose your project involves compiling source files **main.c** and **io.c**, and linking them to produce the executable **project.exe**. The following makefile manages the task of making **project.exe**:

```
project.exe : main.obj io.obj
  link /out:project.exe main.obj io.obj

main.obj : main.c
  cl /c main.c

io.obj : io.c
  cl /c io.c
```

This makefile shows three rules, one each to make **project.exe**, **main.obj**, and **io.obj**. These rules are called explicit rules because they are supplied in the makefile. **omake** also has inference rules that generalize the make process. Inference rules are discussed in *Inference Rules on page 29*.

2.5 Dependency Lines

Lines that include the colon (**:**) are called dependency lines. They specify a relationship between targets and the files on which they depend. This is the general form of a dependency line:

target `[` *target* `... ] [` *attribute* `... ] : [` *dependency* `... ]`

To the left of the colon is the target of the dependency. To the right of the colon are the *dependencies* needed to make the target. The target depends on the dependencies. For example, the following line states that **project.exe** depends on **main.obj** and **io.obj**:

```
project.exe : main.obj io.obj
```

There can be one or more *targets*, optionally a list of target *attributes*, a colon, one or more space or tab characters, and an optional list of *dependencies*. (Other make programs may call the dependency a source, dependent, or prerequisite.) The first target name must start in the first column of the makefile.

Dependency lines specify when to make the target. **omake** uses configuration lookup to determine whether to build the target or reuse an existing derived object.

Explicit and Inferred Dependencies

The dependencies that are listed explicitly on dependency lines are called explicit dependencies. For example, the line

```
test.exe : main.obj sub.obj
```

declares that **test.exe** depends on **main.obj** and **sub.obj**. When **omake** builds **test.exe**, it uses configuration lookup to compare **test.exe** with your current build configuration to see whether it must be updated.

A dependency line can declare that several targets have several dependencies. In the following line, both **main.obj** and **sub.obj** depend on both **system.h** and **io.h**:

```
main.obj sub.obj : system.h io.h
```

omake also has inference rules to infer a dependency of a particular target. Sources determined using inference rules are called inferred dependencies. For a discussion of inference rules, see *Inference Rules on page 29*.

Macros in Dependency Lines

The name of the target (and its root name) can be referenced in a dependency line through the use of macros. For example:

```
main.obj sub.obj : $*.h io.h
```

omake processes the target **main.obj** first; the expression **$*** evaluates to the root of the target name, **main**. The dependency line declares that **main.obj** depends on **main.h** and **io.h**. Next, **sub.obj** is processed, with **$*** evaluating to **sub**. Thus, this line also declares that **sub.obj** depends on **sub.h** and **io.h**.

The Make Process Is Recursive

A basic feature of **omake** is to make a target's dependencies before the configuration lookup is performed for the target. The following line instructs **omake** to make **main.obj** and **io.obj** before performing configuration lookup for **project.exe**:

```
project.exe : main.obj io.obj
```

This line instructs **omake** to make **main.c** before performing configuration lookup for **main.obj**:

```
main.obj : main.c
```

Detected Dependencies

Unlike standard make variants, **omake** does not require you to declare source-file dependencies in the makefile; **omake** detects dependencies automatically. This feature guarantees, for example, correct build behavior as C-language header files change, even if the header files are not listed as dependencies in the makefile. However, the list of dependencies must include build-order dependencies, for example, object modules and libraries that must be built before executables.

NOTE: You may want to include source-file dependencies in your makefile to ensure portability to another group or company that is not using **omake**.

Wildcards in Dependency Lines

The dependency side of a dependency line can use a wildcard specification such as this:

```
main.exe : *.obj
```

omake provides this feature, but using it can cause problems if a required file is missing. For example, the line

```
main.exe : *.obj
  link /out:$(.TARGET) $(.SOURCES)
```

works fine until a required **.obj** file is accidentally deleted. **omake** works until the next update of **main.exe**, when the link command isn't called with all the required object names.

The Dependency Line Separator

The colon (**:**), which separates targets and dependencies on a dependency line, is also the character used as the drive separator in Windows NT. To distinguish this colon from the drive separator, you must put white space in front of it or put a space (*target***:<SPACE>**), a tab (*target***:<TAB>**), a semicolon (*target***:;**), another colon (*target***::**), or nothing (*target***:<ENTER>**) after it. We suggest putting at least one space before and after it.

A Dependency Example

Assume the program **test.exe** is linked from **main.obj** and **sub.obj**. These object modules are compiled from source files **main.c** and **sub.c**, respectively. The makefile looks like this:

```
D-1    test.exe : main.obj sub.obj
         link /out:test.exe main.obj sub.obj
D-2    main.obj : main.c
         cl /c main.c
D-3    sub.obj : sub.c
         cl /c sub.c
```

Line D-1 declares that **test.exe** depends on **main.obj** and **sub.obj**. Lines D-2 and D-3 declare that **main.obj** depends on **main.c** and **sub.obj** depends on **sub.c**.

Assume that since the last build of **test.exe**, **main.c** has been changed and **sub.c** has not. Running **omake** causes the first target in the makefile, **test.exe**, to be evaluated. **test.exe** depends on **main.obj**, which depends on **main.c**. **main.c** doesn't depend on anything. Comparing the configuration record of **main.obj** with the current view's build configuration shows that **main.obj** must be rebuilt using the selected version of **main.c**. This is done with the build script:

```
cl /c main.c
```

Next, **sub.obj**, which depends on **sub.c**, is evaluated. Configuration lookup shows that **sub.obj** does not need updating.

Comparing the configuration record of **test.exe** with the current build configuration shows that **test.exe** must be rebuilt with the selected version of **main.obj**. The following build script updates **test.exe**:

```
link /out:test.exe main.obj sub.obj
```

Running the command **omake** again causes **omake** to display this message:

```
omake: 'test.exe' is up to date.
```

2.6 Build Scripts

All lines that immediately follow a dependency line and begin with white space are the target's build script. Each build script is a command or list of commands, such as **compile** or **link**, that **omake** executes to update the target. The explicit rule for a target consists of the dependency line plus the build scripts.

Build scripts appear in the following form:

target `[` *target* `... ] [` *attribute* `... ] : [` *dependency* `... ]`
    build script
.
.
.

For example, the following makefile specifies that making **project.exe** requires running the program **link** to link **main.obj** and **io.obj**. This build script is run only if **omake** determines that **project.exe** needs to be rebuilt.

```
project.exe : main.obj io.obj
  link /out:project.exe main.obj io.obj
```

A build script can appear on the dependency line by separating it from the last dependency by a semicolon. For example, the following lines are equivalent:

```
test.exe : test.c ; cl /c test.c

test.exe : test.c
  cl /c test.c
```

A target can have a build script with multiple lines, listed one after the other. For example:

```
project.exe : main.obj io.obj
  echo Linking project.exe
  link /out:project.exe main.obj io.obj >link.out
```

The first line shows that Windows NT command processor commands can be executed by **omake**. The second line shows redirection of output, where the output of the **link** program is redirected to the **link.out** file.

Build Script Execution

omake executes build scripts in one of two ways:

- Direct execution of the shell-line command as an **.exe** file.
- By passing the build script to a shell program such as **cmd.exe**. This method is slower and takes additional memory, but is necessary if the build script requires a feature that only the shell program can provide.

Each build script runs in its own child process or subshell of **omake**, starting in the current directory. This behavior has special consequences for commands that are supposed to have effect between build scripts. For example, the build scripts

```
copy_to_tmp :
  # Comment and blank lines are allowed
  chdir some_directory
  copy *.* c:\temp
```

copy the files that match ***.*** in the current directory to **c:\temp**. This occurs because the **copy** command starts in the current directory, irrespective of the **chdir** command executed before it. See *Using Multiple-Command Build-Script Lines* on page 73 for the correct way to do this.

Auto-Detection Mode

By default, **omake** detects when to use the shell program (defined by the **.SHELL** directive; see Table 15 on page 92) and when to use direct execution. The shell program is used when the command is or does one of the following:

- A Windows internal command

  break, **call**, **cd**, **chdir**, **cls**, **copy**, **ctty**, **date**, **del**, **delete**, **dir**, **echo**, **erase**, **for**, **if**, **md**, **mkdir**, **path**, **pause**, **prompt**, **rd**, **rem**, **ren**, **rename**, **rmdir**, **set**, **time**, **type**, **ver**, **verify**, **vol**

  To change the list of internal commands, use the **SHELLCOMMANDS** macro. See Table 5 on page 67.

- A batch file (a file with a **.bat** or $(**SHELLSUFFIX**) extension
- A multiple-command shell line
- Uses redirection of input or output

Standard Execution Mode

The standard execution mode is to use the shell program for every build script. For Windows NT, the default shell program is named in the **COMSPEC** environment variable. If this EV is not defined, **cmd.exe** is used.

omake can execute the build script without using the shell program if the command being run is executable and the build script does not use I/O redirection. Use the colon (**:**) build-script prefix to have **omake** execute the build-script line without using the shell program (see *Select the Shell Program* on page 71).

Build-Script Line Exit Status

After it executes each build-script line, **omake** checks the command exit status. The exit status is a number the program returns and is tested by **omake**. At the Windows NT command line, you can check the exit status of the last executed program by using the **if errorlevel** command.

By convention, programs return a zero exit status when they finish without error and a nonzero status when an error occurs. The first build-script line that returns a nonzero exit status causes **omake** to display this message:

```
omake: shell line exit status exit_status. Stop.
```

This usually means that the program being executed failed. Immediately after displaying this message **omake** does any necessary deinitialization and exits. You can control this behavior with the **–i** or **–k** command-line options (see *Command-Line Options* on page 41), with build-script line prefixes (see *Build-Script Line Prefixes* on page 69), or with target attributes (see *Target Attributes* on page 101).

Some programs return a nonzero line exit status inappropriately; you can have **omake** ignore the exit status by using a build-script prefix. Prefixes are characters that appear before the program name and modify the way **omake** handles the command script. For example:

```
project.exe : main.obj io.obj
  – link /out main.obj io.obj
```

The dash (**–**) prefix causes **omake** to ignore the exit status of that build-script line. If the exit status is nonzero, **omake** displays the following message:

```
omake: Shell line exit status exit_status (ignored)
```

See *Build-Script Line Prefixes* on page 69 for more information.

2.7 Macros

A macro is a makefile line that consists of a macro *name*, an equal sign (**=**), and a macro *value*. In the makefile, a macro reference is an expression of the form **$(***name***)** or **${***name***}**. The reference is macro-expanded to produce *value*. When *name* is a single character, the **( )** or **{ }** around *name* are optional; for example, **$X**, **$(X)**, and **${X}** all specify the value of macro X. The macro character ($), always instructs **omake** to expand the macro that follows. When you need a dollar sign in your makefile, you must use **$$**.

In the following makefile, the text `main.obj io.obj` occurs more than once. To reduce the amount of repeated text, you can use a macro definition to assign a symbol to the text.

```
project.exe : main.obj io.obj
  link /out main.obj io.obj

main.obj : main.c
  cl /c main.c

io.obj : io.c
  cl /c io.c
```

Here is the same makefile written with the introduction of four macros:

```
OBJS          = main.obj io.obj
CC            = cl
CFLAGS        = /c

project.exe : $(OBJS)
  link $(OBJS) /out : project.exe

main.obj : main.c
  $(CC) $(CFLAGS) main.c

io.obj : io.c
  $(CC) $(CFLAGS) io.c
```

The value of the **OBJS** macro is the list of object files to be compiled. The macro definitions for **CC** and **CFLAGS** make it easier to change the name of the C compiler and its options.

omake imports environment variables as macros, so you can refer to things like **$(COMSPEC)** or **$(PATH)** in your makefile without having to define them in the makefile.

Macro Precedence

Macros can be defined in makefiles, defined on the command line, or predefined by **omake**. **omake** also accesses the values of environment variables as if they were macros.

Where a macro is defined determines its precedence. To redefine an existing macro, the new definition must have at least as high a precedence. This is the order of precedence:

1. Macros predefined by **omake**
2. Command-line definition
3. Environment definition (with **–e** command-line option)
4. **makefile** (and **make.ini**) definition
5. Environment definition (default)

By default, the lowest precedence for a macro is the environment definition. The **–e** command-line option gives the environment definition a higher precedence than the makefile definition. With this option, a macro definition in a makefile does not redefine a macro from the environment.

Defining Macros in the Makefile

Macros are defined at read time in a makefile with macro definition lines of the form

name	=	[*text*]	*standard definition*
name	?=	[*text*]	*conditional definition*
name	:=	[*text*]	*expanded definition*
name	+=	[*text*]	*appended definition*

where *name* is the macro name starting in the first column of the makefile. The name can include any characters except the equal sign (**=**), the colon (**:**), and white space. By convention, the *name* is composed of uppercase letters, periods (**.**), and underscores (**_**). Any macro references in *name* are expanded; if you want a literal dollar sign in *name* you must use **$$**.

The *text* is arbitrary text and can reference the value of other macros with expressions of the form **$(***other_macro***)**. There are four forms of macro definition:

Read-Time Expansion of Macros

At read time, some parts of a makefile are macro-expanded; other parts are not expanded until later.

For macro definitions, expansion depends on the separator between the *macro name* and the *value*:

=	*macro name* is expanded; *value* is not expanded until referenced (In NMAKE compatibility mode, *value* is expanded before assignments)
?=	*macro name* is expanded; *value* is not expanded until referenced
+=	*macro name* is expanded; *value* is not expanded until referenced
:=	*macro name* is expanded; *value* is expanded

The makefile can also contain conditional directives such as these:

```
%if condition
%elif condition
```

The *condition* is macro-expanded and evaluated only if the previous enclosing condition is true. For example:

%if condition_1	*(expanded & evaluated)*
% if condition_2	*(expanded & evaluated only if condition_1 is true)*
% endif	
%else	
% if condition_3	*(expanded & evaluated only if condition_1 is false)*
% endif	
%endif	

Standard Macro Definition: name = [text]

This definition defines macro *name* and sets its value to *text*. If *text* is not given, the value of the macro is the null string, “”. White space before or after the = is ignored.

Conditional Macro Definition: name ?= [text]

This definition is like the standard macro definition, but the macro is defined only if it isn't already defined. White space between *name* and **?=** and between **?=** and text is ignored.

Expanded Macro Definition: name := [text]

This definition defines macro *name* and sets its value to the macro-expansion of *text*. The text is arbitrary text and can refer to the value of other macros with expressions of the form **$(***other_macro***)**, which are expanded. If *text* is not given, the value of the macro is the null string (""). White space between *name* and **:=** and between **:=** and *text* is ignored. Expanded macro definitions are useful when the *text* is expensive to calculate (it may require reading a file, for example).

Appended Macro Definition: name += [text]

This definition appends *text* to the current value of macro *name*. White space between *name* and **+=** is ignored. If there is no white space between **+=** and *text*, the new value is OLDVALUE*text*. Otherwise, the new value is OLDVALUE **<SPACE>** *text* (with one intervening space). If *name* is an undefined macro, this definition is the same as a standard macro definition.

Case-Sensitivity of Macro Names

By default, for Windows NT **omake** macro names are case-insensitive. The **.CASE_MACRO** and **.NOCASE_MACRO** directives turn case-sensitivity on and off.

The Location of Macro Definitions

By convention, macro definitions appear at the beginning of the makefile. Macros must be defined before they are expanded; if they are not, their expanded value is the null string.

Indenting Macro Definitions

Usually *name* starts in the first column of the makefile line, but macro definitions can be indented (for example, inside an **%if**...**%endif** conditional directive). Indenting is allowed only before the first target in the makefile, or after a nonindented macro definition. This restriction is a consequence of **omake**'s use of indenting to indicate a target's build scripts.

Undefining Macros

Macros can be undefined with the **%undef** directive at read time:

```
%undef CFLAGS
```

Some macros that **omake** predefines cannot be undefined with this directive.

Example Macro Definitions

➤ This defines the macro **OBJS** as **main.obj sub.obj**. **$(OBJS)** is **main.obj sub.obj**.

```
OBJS = main.obj sub.obj
```

➤ This defines **PROJECT** as **$$/Make**. **$$** is a literal dollar sign. This means that **$(PROJECT)** is **$/Make**.

```
PROJECT= $$/Make
```

➤ These define the macro **DEBUG** as **7** and the macro **CFLAGS** as **–Z$(DEBUG)**. **$(DEBUG)** is **7** and **$(CFLAGS)** is **–Z7**.

```
DEBUG        = 7                          (defined as 7)
DEBUG        ?= i                         (not redefined)
CFLAGS       = –Z$(DEBUG)
```

➤ This shows an appended macro definition inside a conditional directive. If the **OPT** macro is defined, the **CFLAGS** macro's value has <SPACE>**–Od** appended to it. The new value of **CFLAGS** is **–Z$(DEBUG) –Od** and **$(CFLAGS)** is **–Z7 –Od**.

```
%if defined(OPT)
 CFLAGS   += –Od
%endif
```

➤ This defines macro **OBJSLIST** as **project.lst** and then defines **OBJS**. The **:=** causes the right side to be expanded. **OBJSLIST** is expanded, producing **project.lst** and the **@** macro modifier (see *Macro Modifiers* on page 52) reads the file **project.lst**. The contents of this file is the value of **OBJS**. The expression **:=** is useful here because reading the file is a relatively slow process that you want done only once. If **OBJS** is defined with a standard macro definition, each occurrence of **$(OBJS)** causes **omake** to read **project.lst**. By using **:=**, **project.lst** is read only once, when **OBJS** is defined.

```
OBJSLIST   = project.lst
OBJS       := $(OBJSLIST,@)
```

Defining Macros on the Command Line

Macros can be defined on the command line, and the value of the command-line macro overrides a makefile macro or environment definition with the same name. Only standard macro definitions are allowed on the command line. A command-line macro that contains spaces must be enclosed in double quotes. (To include a literal double quote, use **\"**.) For example, the command line

omake `BSCFLAGS=/n "CFLAGS=-Zi -Od"`

runs **omake** with **BSCFLAGS** defined with the value **–n** and **CFLAGS** defined with the value **–Zi –Od**.

Dynamic Macros

omake defines some special macros whose values are dynamic. These run-time macros return information about the current target being built. For example, the **.TARGET** macro is name of the current target, the **.SOURCE** macro is the name of the inferred dependency (from an inference rule) or the first of the explicit dependencies, and the **.SOURCES** macro is the list of all dependencies.

Using run-time macros, the example can be written as follows:

```
OBJS          = main.obj io.obj
CC            = cl
CFLAGS        =

project.exe : $(OBJS)
  link /out:$(.TARGET) $(OBJS)

main.obj : main.c
  $(CC) $(CFLAGS) –c $(.SOURCE)

io.obj : io.c
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

As you can see, the build scripts that update **main.obj** and **io.obj** are identical when dynamic macros are used. Dynamic macros are important for generalizing the build process with inference rules, as shown in *Inference Rules on page 29*.

Macro Modifiers

Macros can be used to reduce the amount of repeated text. They are also used at run time to generalize the build process with inference rules. You often want to start with the value of a macro and modify it in some manner. For example, to get the list of source files from the **OBJS** macro you can define this macro:

```
SRCS         = $(OBJS,.obj=.c)
```

This definition uses the *from*=*to* macro modifier to replace the *from* text in the expansion of **OBJS** with the *to* text. The result is that **$(SRCS)** is **main.c io.c**. In general, to modify a macro, expand it with

$(*name***,***modifier***[,***modifier* ...**])**

Each *modifier* is applied in succession to the expanded value of *name*. Separate modifier with comma.

File Name Components

There is a complete set of macro modifiers for accessing parts of file names. For example, with this macro definition:

```
SRCS         = \src\main.c parse.l
```

Table 1 lists some of the modifiers.

Table 1 Macro Modifiers

Modifier and description	**Example**	**Value**
D, the directory	**$(SRCS,D)**	**\src .**
E, the extension (or suffix)	**$(SRCS,E)**	**.c.l**
F, the file name	**$(SRCS,F)**	**main.c parse.l**

Tokenize

The **W***str* modifier replaces white space between elements of the macro with *str*, a string. The *str* can be a mix of regular characters and special sequences, the most important sequence being **\n**, which represents a newline character (like pressing the **<ENTER>** key). For example:

```
$(OBJS,W +\n) is     main.obj +
  io.obj
```

Other Modifiers

Other modifiers include **@** (include file contents), **LC** (lowercase), **UC** (uppercase), **M** (member), and **N** (nonmember). The **M** and **N** modifiers and the **S** (substitute) modifier use regular expressions for powerful and flexible pattern-matching. See *Macro Modifiers* on page 52 for more information.

Environment Variables

Environment variables are placed into the environment with this command:

```
set name=value
```

NOTE: This example applies to **cmd.exe**; the command varies depending on the shell you use.

By default, **omake** preloads all environment variables as macros. The **.NOENVMACROS** directive prevents this loading and hence prevents **omake** from accessing the value of any environment variables.

Macro Expansion or Macro Referencing

Expanding a macro also expands any macro references recursively. For example:

```
CDEFS  = –DDEBUG –DNT
COPTS  = –Ot
CFLAGS = –Zi $(CDEFS) $(COPTS)
```

The expression **$(CFLAGS)** evaluates to **–Zi –DDEBUG –DNT –Ot**.

In a macro reference **$(***name***)**, *name* can reference other macros. For example, given the definitions

```
SYS      = NT
NTFLAGS  = –DNT –UNT
```

the expression **$($(SYS)FLAGS)** evaluates to **–DNT –UNT**.

Run-Time Expansion of Macros

Macros in build scripts are expanded immediately before the build scripts are executed. There are several macros whose values change dynamically at run time. They are discussed in *Predefined Macros: Run-Time Macros* on page 60.

Recursive Macro Definitions

NOTE: NMAKE supports recursive macro definitions as shown in this section. When **omake** is emulating NMAKE, recursive macro definitions are supported.

A macro value may reference other macros. If the value circularly references itself, **omake** displays a warning message when the macro is expanded. For example, the values

```
A = A $B
B = B1 $A B2
%echo $A
```

generate this message:

```
omake: file (line num): Recursive macro 'A = A $B' (warning).
A B1  B2
```

The expression **$A** expands to **A** *expansion_of_B*. In turn, the expansion of B is **B1** *expansion_of_A* **B2**. When **omake** tries to expand a macro that is already being expanded, it displays the warning message and ignores the recursive expansion. The *file* and *num* depend on where the example lines were defined.

You can make this kind of recursive reference by using the expanded macro definition:

```
B := B1 $A B2
```

This type of definition expands any references in the right side before the macro definition of **B** occurs.

2.8 Inference Rules

Inference rules generalize the build process, which eliminates the need to give **omake** an explicit rule for each target. For example, compiling C source (**.c** files) into object files (**.obj** files) is a common occurrence. Rather than require a statement that each **.obj** file depends on a like-named

.c file, **omake** uses an inference rule to infer that dependency. The dependency determined by an inference rule is called the inferred dependency.

Inference rules also provide build scripts to update the target from the inferred dependency. The target inherits these build scripts if it does not have its own.

omake predefines several inference rules, and you can change their definitions or define your own rules.

Defining Inference Rules

Inference rules (also called metarules) are identified by the use of the rule character (%) in the dependency line. This character is a wildcard, matching zero or more characters. For example, here is an inference rule for building **.obj** files from **.c** files:

```
%.obj : %.c
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

This rule states that a **.obj** file can be built from a corresponding **.c** file with the build-script line **$(CC) $(CFLAGS) –c $(.SOURCE)**. The **.c** and **.ob**j files share the same root of the file name.

When the dependency and target have the same file name except for their extensions, this rule can be specified in an alternative way:

```
.c.obj :
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

The alternative form is compatible with other **make** utilities.

omake predefines the **%.obj : %.c** inference rule as listed above so the example now becomes much simpler:

```
OBJS          = main.obj io.obj
CC            = cc
CFLAGS        = /Zi

project.exe : $(OBJS)
  link /out:$(.TARGET) $(OBJS)
```

The General Inference Rule Definition

```
[ Tp ] % [ Ts ] [ attribute ... ] : [ Sp ] % [ Ss ]
    build script
  .
  .
  .
```

On the target side of the dependency line is a pattern: a target prefix *Tp*, a **%**, and a target suffix *Ts*. Any target attributes (see *Target Attributes* on page 101) appear next. On the dependency side of the line is a dependency prefix *Sp*, a **%**, and a dependency suffix *Ss*. The prefixes can contain any character except **%** and, in particular, can be a directory. The suffixes can contain any character (including **%**, which is taken literally).

In order for a rule to match a target, *Tp* and *Ts* must match the first and last parts of the target name, with **%** matching everything in between. The inferred dependency name is *Sp* followed by the characters matched by **%,** followed by *Ss*.

The Target Inherits Build Scripts and Attributes

Following the dependency line are the build-script lines that update the target from the inferred dependency. If a target doesn't have its own build scripts, it *inherits* the build scripts and attributes of the inference rule. Target attributes take precedence over rule attributes.

Alternative (Suffix-Only) Form

Inference rules can also be defined in this form:

```
.source_extension.target_extension :
  build script
  .
  .
  .
```

The *source_extension* is the extension of the dependency. The *target_extension* is the extension of the target. This alternative form is compatible with other make utilities and is discussed in *Compatibility with Suffix Rules (.SUFFIXES)* on page 37. The suffix-only form is converted by **omake** to the equivalent metarule form:

```
%.target_extension : %.source_extension
  build script
  .
  .
  .
```

Automatic Use of Inference Rules

omake uses inference rules when building a target that has no build scripts. Even when targets have build scripts, you can cause **omake** to use inference rules by giving the target the **.INFER** target attribute.

In either case, **omake** first builds the target's explicit dependencies (those listed on dependency lines), and then uses its inference rules to search for an inferred dependency. The search proceeds by finding all rules that match the target, building each possible inferred dependency name in turn, and checking whether the inferred dependency exists as a file. If it exists, **omake** proceeds as follows:

1. If the target has no build scripts, the target inherits the inference-rule build scripts and attributes. When this is a conflict between the attributes, the target's attributes take precedence.
2. The inferred dependency is added to the target as a dependency.
3. The inferred dependency is built.

Inference Rules and Target Groups

Inference rules can build several targets (a target group) from a single dependency. To do this put the targets on the target side of the rule, with a plus sign (**+**) between them. (There must be white space between the **+** and the target names.) When **omake** executes the rule's build scripts to update any target in the group, all targets are updated.

For example, here is a rule for building both **.c** and **.h** files with a **yacc** program, which takes a **.y** file as input:

```
%.h + %.c : %.y
  $(YACC) –d $(YFLAGS) $(.SOURCE)
  copy y.tab.h $(.TARGET,1)
  copy y.tab.c $(.TARGET,2)
  del ytab.*
```

Note the use of **$(.TARGET,***num*), which evaluates to the names of both the **.h** and **.c** files. The *num* macro modifier selects the *num*th element of the macro. So, for example, **$(.TARGET,1)** is **parse.h** when the inferred dependency is **parse.y**.

Multiple-Step Inference Rules

When **omake** tries to find the inferred dependency of a target, it first tries all rules that directly produce the target. If the inferred dependency cannot be found with one rule, **omake** chains rules into twos, threes, and so on. **omake** always chooses the smallest number of rules (the shortest path) between a target and its inferred dependency.

If a multiple-rule path is found, **omake** creates the targets between the inferred dependency and the target, and chains them. Chained targets have special properties:

- **omake** deletes chained targets as part of the build; as soon as it finishes running the build scripts, it deletes any intermediate chained targets. A chained target is not considered a sibling derived object.
- For the purposes of time stamp comparisons, a chained target is the same age as the target that depends on it. This allows **omake** to compare the time stamp of the target at one end of the chain with the time stamp of the inferred dependency at the other end and correctly determine whether the target needs updating.

NOTE: Although the characters matched by % are the same within a single inference rule, the % can match different characters for the different rules of a multiple-rule path.

Chained Targets Are Deleted Automatically

To illustrate how the deletion of chained targets works, here are rules for extracting a **.obj** file from a **.c** file and for compiling the **.obj** file into a **.exe** file:

```
%.obj : %.c
  $(CC) $(CFLAGS) -c $(.SOURCE)
```

```
%.exe : %.obj
  $(LINK) /out:$(.TARGET) $(LINKFLAGS) $(.SOURCE)
```

Assume that **omake** is trying to build **main.exe** and that **main.c** exists, but **main.obj** does not. **omake** finds the two-rule inference that uses the **%.obj : %.c** rule to produce **main.obj**, and the **%.exe : %.obj** rule to produce **main.exe**. **omake** retrieves **main.c** and compiles it to produce **main.obj**. It then links **main.obj** to produce **main.exe**. After running the **%.exe** and **%.obj** build-script lines, **omake** deletes **main.obj** because it was marked as being chained.

To prevent this deletion, write a rule that combines the rules in the multiple-rule path but that has a shorter path. A one-step rule between **main.exe** and **main.c** that leaves behind **main.obj** is

```
%.exe : %.c
  $(CC) $(CFLAGS) -c $(.SOURCE)
  $(LINK) /out:$(.TARGET) $(LINKFLAGS) $(.SOURCE)
```

A second way to prevent this deletion is to use the **.PRECIOUS** attribute. **omake** does not delete targets with this attribute. Modify the **%.obj : %.c** rule as follows:

```
%.obj .PRECIOUS : %.c
  $(CC) $(CFLAGS) -c $(.SOURCE)
```

Preventing Multiple-Step Rules

A *target* with the **.NOCHAIN** attribute instructs **omake** to try only the one-step paths for finding the inferred dependency.

A rule with the **.NOCHAIN** attribute cannot be used in the middle of a multiple-step rule. That is, it means this rule is terminal.

Inference Rule Search Order

To determine which inference rules to use, **omake** gathers all rules that can build the current target and sorts them from best score to worst score. For each rule in this sorted list, **omake** constructs the dependency name and tests whether it exists as a file. If so, **omake** chooses this rule as the inference rule, and this file as the inferred dependency.

The score is defined as the number of characters in the target name that **%** matches; the best score is zero. Often, many rules may have the same score (for example, all rules that build **%.obj** targets), and **omake** puts these rules in creation order. For identical scoring rules, the rule created first (usually built in to **omake** or in **make.ini**) is first on the list.

Overriding the Rule Ordering

Sometimes the rule ordering is inappropriate. For example, you may have both **video.c** and **video.cpp** that can be used to build **video.obj**, and you want **omake** to use **video.c**. By making **video.obj** depend on **video.c**, **omake** chooses the **%.obj : %.c** rule to update **video.obj**:

```
video.obj : video.c
```

If you find yourself doing this, you can prevent **omake** from doing the inference rule search by supplying your own build scripts. For example:

```
video.obj : video.c
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

One problem with supplying the build scripts explicitly is that they must change if the **%.obj : %.c** rule changes. To avoid this problem, use the **%do** directive to execute the build scripts of the **%.obj : %.c** rule:

```
video.obj : video.c
  %do "%.obj : %.c"
```

Rule Finding for Target Names with a Directory Component

When the target name has a directory component (for example, **objs\video.obj**), **omake** uses the following methods to find a rule:

1. The rules with directory components in their targets are matched against the complete target name.
2. If no rules match in Step #1, the rules without directory components in their targets are matched against the file name of the target, by default. The **.UNIXPATHS** directive changes this behavior. When you use **.UNIXPATHS**, the rules without directory components are matched against the full target name.

Common Operations on Inference Rules

These operations on inference rules are performed most often:

➤ Redefining an inference rule

  Inference rules can be redefined any number of times. To redefine an inference rule, provide a new definition in one of two places:

- ➢ In **make.ini**, to give every makefile access to the rule
- ➢ In your makefile, if you want a local variant

- Changing an inference rule's attributes

  To change an inference rule's attributes, use a dependency line with the new attributes and don't specify a build script. These attributes are combined with the current attributes.

- Disabling an inference rule

  To disable an inference rule, use a dependency line with no attributes and no build script:

  ```
  %.obj : %.c
  ```
  ...or...
  ```
  .c.obj :
  ```

- Disabling the search for an inferred dependency

  omake searches for an inferred dependency for targets that have no build script. You can omit the inference search with the **.NOINFER** attribute. For example:

  ```
  all .NOINFER : import test export
  ```

  The **all** target only drives the build of its dependencies, so no inference search is needed.

- Using % in nonrules

  A **%** on the target side of a dependency line indicates that this line is an inference rule. To use a literal **%** without indicating a rule, give the target the **.NORULE** attribute:

  ```
  percent%.obj .NORULE : percent%.c
  ```

- Changing the rule character

  The **.RULE_CHAR** directive sets the rule character. For example, to change it to an asterisk:

  ```
  .RULE_CHAR : *
  ```

Built-In Inference Rules

omake predefines several rules. You can reject them by using the **–r** (reject rules) command-line option or the **.REJECT_RULES** directive. The built-in rules are listed in *Inference Rules* on page 142.

Compatibility with Suffix Rules (.SUFFIXES)

Many make utilities, such as PM/CB, NMAKE, and Borland Make, have a simple style of inference rule, called suffix rules. These rules use only the suffix (extension) of the file name. Here is a suffix rule:

```
.c.obj :
  $(CC) $(CFLAGS) –c $<
```

Some make utilities also use a **.SUFFIXES** directive to control the search order of the suffix rules. The **.SUFFIXES** directive lists the suffixes in the order that inference rules are to be attempted. Each appearance of **.SUFFIXES** prepends to the current list of suffixes, but a **.SUFFIXES** directive with nothing on the dependency side clears the list:

`.SUFFIXES :`	*(clears list)*
`.SUFFIXES : .lib`	*(list is: .for .lib)*
`.SUFFIXES : .exe .obj`	*(list is: .exe .obj .lib)*

When looking for an inference rule to build a target, the list is traversed from left to right with each extension being combined with the target extension to form the name of a suffix rule. If that suffix rule exists, **omake** forms the name of the corresponding inferred dependency and, if it exists, the inference search is complete.

Handling of Suffix Rules and .SUFFIXES

omake converts suffix rules into equivalent **omake** inference rules. If **.SUFFIXES** is used, **omake** uses the suffixes order to sequence *only* its suffix rules. Nonsuffix rules are left in creation order. By default, all inference rules are in creation order. Note that a **.SUFFIXES** directive with no suffixes on the dependency side effectively disables all suffix rules, but nonsuffix rules remain enabled.

2.9 Response Files

omake has two kinds of support for response files: automatic response files, where **omake** determines when to build a response file, or inline response files, where statements that create response file creating statements are written in the makefile.

Inline Response Files

Response files can also be coded inline in your makefile. For example:

```
project.dll:$(OBJS)
     link @<<
$(OBJS)
/out:$(.TARGET)
$(.TARGET)
<<
```

The **link** program is invoked as **link** @*response_file*, where *response_file* is a name generated by **omake**. The *response_file* contains:

```
main.obj io.obj /out:project.exe
```

2.10 Directives

Makefile directives control the makefile lines **omake** reads at read time. The following makefile uses conditional directives (**%if**, **%elif**, **%else**, and **%endif**) to support both Borland and Microsoft compilers. Comments have been added for documentation.

```
# This makefile compiles for the configuration specified
# in the CFG macro.

CFG                         = Release
OBJS                        = main.obj io.obj

# Configuration-dependent section
%if $(CFG)                  == Release
  CFLAGS                    = /O1
  LINK_FLAGS                =
%elif $(CFG)                == Debug
  CFLAGS                    = /Od /Zi
  LINK_FLAGS                = /debug
%else
%abort Unsupported configuration $(CFG)
%endif
```

```
project.exe: $(OBJS)
        link /out:$(.TARGET) $(LINK_FLAGS) $(OBJS)

%.obj: %.c
        cl $(CFLAGS) /c $(.SOURCE)
```

The layout of this makefile is fairly traditional; macros are defined first, and the primary target follows the macros.

This example also uses the **%abort** directive to abort **omake** if the makefile does not support a particular compiler. Directives can also be used at run time, to control the build scripts **omake** executes. For more information, see *Makefile Directives* on page 74.

2.11 The Keep-Working Mode

omake usually stops at the first build-script line that returns an error. This is the right behavior for interactive work, but not for a long, unattended build such as an overnight build of your entire system. The solution is to use the keep-working mode. This mode allows **omake** to do the maximum amount of work consistent with any errors that occur during the build.

In the keep-working mode, a build script error while **omake** is updating a target causes **omake** to stop working on the current target. The make process continues, with **omake** noting that the current target was incompletely built. Future targets are updated only if they do not depend on any incomplete targets.

Because **omake** does as much work as possible, you only have to fix the problem targets and start **omake** again. The keep-working mode does the maximum amount of safe and correct work.

This mode is enabled with the **.KEEPWORKING** makefile directive (see *Dot Directives* on page 91) or **–k** command-line option (see *Command-Line Options* on page 41).

3 omake Reference

This chapter contains reference information for **omake**.

3.1 Command-Line Options

Command-line options or flags are indicated by a dash (**–**) or slash (**/**) followed by a one-letter option name. Some options take an argument and that argument follows the option name with or without intervening space. Several options can be grouped together after a single dash, but options that take an argument can only be at the end of the group. To signal the end of options, a double dash (**--**) can be used in case a command-line target or macro has a dash as the first character. Here are some examples:

omake –n –d –f project.mak –p
omake –ndpf project.mak
omake –f ../makefile -- –readme–

The first and second example are identical and show how options can be grouped. The third example shows the use of the double dash to indicate the end of options. This is required to prevent **omake** from interpreting **–readme–** as more options.

Initial Command-Line Parameters: OMAKEOPTS

Before parsing the command line, **omake** looks for the **OMAKEOPTS** environment variable and, if it is found, parses its value. After reading the initialization file, **omake** also parses the value of the **MFLAGS** macro for additional command-line parameters. **MFLAGS** can be defined on the command line, in **make.ini** or in the environment (but an **MFLAGS** definition in the makefile is ineffective). Using the **–z** command-line option causes **omake** to ignore **MFLAGS**. Setting **MFLAGS** in **make.ini** is strongly discouraged, because directives can be used instead. After the **OMAKEOPTS** environment variable is parsed, the command line itself is parsed.

The Command-Line Options

The options in Table 2 are parameters for **omake**'s native mode. Many parameters are different for NMAKE and PolyMake emulation modes.

Table 2 omake Command-Line Options (Part 1 of 4)

Option	Meaning
–a	Updates all targets whether or not they are out of date. If no targets are specified on the command line, the default target is updated.
–A	Uses automatic dependencies. This option is enabled only if you are not using configuration lookup (because you are processing non-MVFS files or using the **–W** option).
–b *file*	Names *file* as the initialization (built-ins) file (The default is **make.ini**.) If *file* is the empty string, no initialization file is read. You can specify this on the command line with "**–b**<SPACE>", **–b**"<SPACE>" or **–b**<SPACE>"". The value of the **BUILTINS** macro is set to the full pathname of *file*.
–d	Run-time debugging mode. A run-time trace of **omake** operation displays targets being made and rules being tried. See also the **–p** command-line option.
–D	Keep-directory mode. The first access of the current directory or any search directory to look for a file reads the directory into memory. Subsequent accesses to the directory use the in-memory version and are much quicker.
–e	Environment macro precedence. Macros defined from environment variables have a higher precedence than makefile macros.
–EN	Sets **omake**'s emulation mode to NMAKE. Selecting this emulation causes the rest of the command line to be interpreted as an NMAKE command line. See *Microsoft NMAKE Compatibility* on page 158 for more information.
–EO	Default emulation mode; that is, no emulation.

Table 2 omake Command-Line Options (Part 2 of 4)

Option	Meaning
–EP	Sets **omake**'s emulation mode to PM/CB. Selecting this mode causes the rest of the command line to be interpreted as a PM/CB command line. See *PM/CB (Intersolv Configuration Builder and PolyMake)* on page 145 for more information.
–E2	Sets **omake**'s emulation to Opus Make v5.2x. See *Opus Make v5.2x Compatibility and Emulation* on page 162 for more information.
–f *file*	Specifies *file* as the makefile. More than one **–f** option may appear, and the files are read in order. **omake** first tries to read *file*, but if *file* does not exist, **omake** tries *file***.mak**. If *file* is **con** or **–** the console is read for the makefile and **<CTRL+Z>**,**<ENTER>** (**<CTRL+D>** for UNIX) as the last input finishes the input. If no **–f** options are specified, **omake** tries **makefile** and then **makefile.mak**. The value of the **MAKEFILE** macro is the *file* of the first **–f** option.
–G	Restricts dependency checking to *makefile dependencies* only—those dependencies declared explicitly in the makefile or inferred from an inference rule. All *detected dependencies* are ignored. For safety, this option disables winkin of DOs from other views; it is quite likely that other views select different versions of detected dependencies. For example, a derived object in your view may be reused even if it was built with a different version of a header file than your view currently selects. This option is mutually exclusive with **–W**.
–h	Displays a help screen showing the command-line syntax and command-line options. The current emulation mode determines the help that is displayed.
–i	Ignores exit status, which causes the nonzero exit status from any build scripts to be ignored. This option can be used to collect all errors into a single file without stopping **omake**. See also the **–k** option. Many compilers send their error messages to standard error. This command-line option redirects standard error either into a file or to the standard output. With it, the command **omake –i –x – > errs** collects all messages into the **errs** file. See the **–x** command-line option.

Table 2 omake Command-Line Options (Part 3 of 4)

Option	Meaning
–k	Keep-working mode. Any errors when updating a target cause work on that target to be stopped, but the make process continues. Because the target was incompletely made, any other targets that depend on it are prevented from being updated. This mode is handy for long, unattended builds because it maximizes the amount of making without completely ignoring the exit status as the **–i** command-option does.
–L	Disables configuration lookup and config record creation. See also **clearmake –F**.
–M	Makes makefile. **omake** makes the makefile before reading it. See the section *Makefile Directives* on page 74.
–n	No execute. Displays, but does not execute, the build scripts that update the targets. This option is useful for debugging makefiles and for showing the work that will be done. To override this option for a recursive make, use the **.MAKE** target attribute (see *Target Attributes* on page 101) or **&** shell-line prefix (see *Build-Script Line Prefixes* on page 69).
–O	Disables build script checking during configuration lookup.
–p	Prints debugging information to screen. Macros are listed with their location of definition and value; search directories are indicated; inference rules are displayed and targets, build scripts and attributes are printed. The interpretation of the output is discussed in Chapter 4, *Debugging Makefiles*.
–r	Rejects inference rules. **omake** ignores inference rules that are built in and that are defined in the initialization file. Only inference rules defined in the makefile are used.
–s	Silent mode. Build scripts are usually displayed before they are executed. The **–s** option prevents this display.
–T	Examines sibling derived objects when determining whether a target object in a VOB can be reused (is up to date). By default, when determining whether a target can be reused, **omake** ignores modifications to objects created by the same build rule that created the target (sibling derived objects). **–T** directs **omake** to consider a target out of date if its siblings have been modified or deleted. **–T** is equivalent to **clearmake –R**.

Table 2 omake Command-Line Options (Part 4 of 4)

Option	**Meaning**	
–v	Increases output verbosity during the build process (particularly with respect to configuration lookup and config record creation).	
–W	Disables shopping for derived objects to wink in. **–W** is equivalent to **clearmake –V**.	
–x *file*	Redirects error messages into file. As a special case, if *file* is **–**, the error messages are redirected to the standard output. For example:	
	omake –x make.err **omake –x – >> make.err**	*(errors go into make.err)* *(errors are appended to make.err)*
–z	Ignores **MFLAGS**. The **MFLAGS** macro is not examined for additional options.	
–#1	Debugging option—read-time debugging mode. A read-time trace of **omake** operation displays makefiles being read and conditional directives being interpreted. See Chapter 4, *Debugging Makefiles* for an interpretation of the **–#1** output.	
–#2	Debugging option—warns about undefined macros. When **omake** tries to expand the value of a undefined macro, it displays a warning message. The default action is to silently ignore undefined macros.	
–#4	Debugging option—warns about unrecognized makefile lines. When **omake** reads a line in a makefile it can't understand, it displays a warning message. The default action is to silently ignore unrecognizable makefile lines.	
–#8	Debugging option—leaves behind generated response files and batch files. **omake** uses several temporary files, which are usually deleted before **omake** quits. This option causes **omake** to leave them behind so they can be examined.	

Debugging features can be specified as a sum or as a comma-separated list. For example, both **–# 7** and **–# 1,2,4** display makefile lines, warn about undefined macros, and warn about unrecognized makefile lines). See also the description of the **.DEBUG** directive in the section *Dot Directives* on page 91.

The Current Options

After reading its initialization file, **omake** exports all command-line options except **–b** and **–f** into the **MFLAGS** macro (unless you are using **omake** in an emulation mode). Also, the state of each command-line option is kept in the macros listed in the section *Predefined Macros: State Macros* on page 63 and in Appendix C, *Built-In Macros and Rules*.

3.2 Locating the Initialization File

omake locates the initialization file in the following manner:

1. If the **–b** *file* command-line option is used, **omake** uses *file*.
2. If the **OMAKECFG** environment variable exists, **omake** uses its value.
3. Otherwise, **omake** searches for **make.ini** in the following locations:
   a. The current directory
   b. The directory of the **omake.exe** file
   c. Directories specified in the **INIT** environment variable

Here, we assume the initialization file is named **make.ini**.

When the initialization file is found, **omake** searches its directory for its ancillary files (**omhelp.***
and **builtins.***).

Disabling the Initialization File

omake does not look for the initialization file if you use the command-line option **–b**"**<SPACE>**" or if you set the **OMAKECFG** environment variable to the empty string. To set the environment variable:

```
set OMAKECFG=<SPACE><ENTER>
```

3.3 The Make Process

This section describes the events that occur when you invoke **omake**.

Read Time

At read time, **omake** parses the command line, reads the initialization file, and reads one or more makefiles. Steps in the start-up process:

1. If the **OMAKEOPTS** environment variable exists, it is parsed for command-line parameters.
2. The command line is parsed.
3. The initialization file is found and read.
4. If the **MFLAGS** macro is defined and the **–z** command-line option was not specified, the macro is expanded and parsed as a command line.
5. Each makefile is read in the order it appears on the command line.

omake reads the initialization file and makefiles, collecting macro definitions, dependency lines, rules, and build scripts. Dependency lines are built into a set of dependency trees, with each target node connected by a dependency arc to each of its dependency nodes. Because each dependency is itself a target, it is connected to its own dependencies. At the leaf nodes of the trees are targets without any dependencies.

Run Time

At run time, **omake** makes the command-line targets or, if none are specified, the first normal target in the first makefile. In this context, "normal" means not a directive, special target, attribute, or inference rule. This first normal target is called the default target. Briefly, these are the steps involved in making a target:

1. Locate the target.

   Look for the target as a file on disk and, if found, record its location.

2. Make its dependencies.

Make the target's explicit dependencies.

3. Locate its inferred dependency.

   If the target has no build scripts, use inference rules to look for an inferred dependency. If the inferred dependency is found, add it as a dependency of the target and add the rule's build scripts to the target. Then, make the inferred dependency.

4. Compare and update the target.

   omake's *build avoidance* scheme includes automatic dependency detection. By default, **omake** uses *configuration lookup* to compare the configuration record of the target to your build configuration and determine whether a build is required. If the targets are not in a VOB, **omake** compares time stamps to determine whether a build is necessary. If the time stamp of any target dependency is more recent than that of the target itself, the target is updated by executing its build script. After the build scripts have been executed, the target is made.

If the target was found in Step #1, **omake** assumes its location stays the same. If the target was not found, **omake** makes a second attempt to locate it on disk. In either case, **omake** also updates its internal copy of the target's time stamp.

The consequence of Step #2 and Step #3 is that **omake** makes a target's dependencies recursively. **omake** goes as far as it can down the dependency tree and works back up, using configuration record matching and build avoidance, as described in Step #4. If the target is not in a VOB, **omake** checks the time stamp of the target against its dependencies. For targets that have at least one newer dependency, the target's build scripts are executed to update the target.

The utility of Step #3 is that it generalizes the make process. Inference rules are akin to statements such as "all **.obj** files can depend on like-named **.c** or **.cpp** files."

An extensive example that shows the read-time steps in the make process is in Chapter 4, *Debugging Makefiles*.

Run-Time Initialization and Deinitialization

Before making any targets, **omake** executes the build scripts of the **.BEFORE** special target if it is defined.

Immediately before exiting, **omake** executes the build scripts of the **.AFTER** special target if it is defined. **omake** then removes any temporary files it has created and exits.

Updating the Time Stamp

If the target is not in a VOB, **omake** updates its internal copy of a target's time stamp after the target has been made. The value the time stamp is set to is determined as follows:

- If the target has build scripts, the time stamp is set to the current system time after all build scripts have been executed.
- If the target has no dependencies, the time stamp is set to the current system time.
- Otherwise, the time stamp is set to the most recent of the target's current time stamp and those of all its dependencies.

3.4 Targets

This section provides general information about targets.

Case-Sensitivity of Target Names

By default, for Windows NT **omake** target names are case-insensitive. The **.CASE_TARGET** and **.NOCASE_TARGET** directives turn case-sensitivity on and off.

A Target Has a Name and a Pathname

It is an important fact that a target has both a name (from the makefile) and a *pathname* (its location on disk). Each name in the makefile specifies a unique target. The *pathname* is determined when **omake** searches for the target on disk.

If *name* does not have a path component in it, **omake** may search multiple directories to locate the target and the name and pathname may differ. If *name* has a path component, or if **omake** is not instructed to search multiple directories, the name and pathname are the same. See *Search Directories* on page 106.

Using the Path Separator in Names

On Windows NT, you can use either a slash (**/**) or backslash (\) as the path separator inside of **omake** makefiles. Windows NT accepts either character as the path separator when Make looks in the file system directory.

We suggest you use the slash in your makefiles because the backslash can be used to modify the meaning of the character that follows it. For example, **\<ENTER>** continues makefile lines and **\#** is a literal **#**. If you use the backslash as the path separator, you must be aware of how **omake** interprets this expression:

\*the-following-character*

The First Target in the Makefile Is the Default Target

If no targets are specified on the **omake** command line, the first normal target in the makefile is made. In this context, a normal target is not a directive, special target, attribute, or inference rule. It is a common practice to have the default target depend on other targets to be built. Consider this makefile:

```
all : a.exe b.exe

a.exe : dependencies
  build scripts

b.exe : dependencies
  build scripts
```

Running **omake** without a command-line target causes **all** (**a.exe** and **b.exe)** to be made. The target **all** is sometimes called a dummy target because it is used to drive the build of other (real) targets.

Targets May Appear on Several Dependency Lines

A target may appear on the target side of more than one dependency line. The target depends on all dependencies on all dependency lines that the target appears on, but the build script that updates the target can be listed after only one of the dependency lines.

Double-Colon Dependency Lines

When a double-colon (**::**) separates the target and its dependencies, build scripts can be specified after each double-colon dependency line. If in the MVFS, all build scripts are executed if any dependency on any dependency line is newer than the target. Double-colon dependencies are rarely used. Here is a simple example:

```
backup :: a.exe
  copy $(.SOURCE) c:\backup

backup :: b.exe
  copy $(.SOURCE) c:\backup
```

If you were to execute the **omake backup** command, **a.exe** is built and then **b.exe** . Each target is copied to **c:\backup** after it is built.

Mixing Single-Colon and Double-Colon Dependency Lines

A target cannot appear on the target side of both double- and single-colon dependency lines. If it does, **omake** displays a warning and ignores the offending dependency line.

Targets Without Dependencies

A target that has no dependencies (neither explicit nor inferred) is up to date if it exists as a file; if it doesn't exist, it is out of date and must be updated.

Target Groups

Several targets may be made simultaneously from a single dependency. This is called a target group. Updating any target of the group (by running the target's build scripts) updates all the targets.

To indicate a target group, put a plus sign (**+**) between each target on the dependency line. For example, here is a rule to build **parse.h** and **parse.c** simultaneously from **parse.y** using the program called **yacc**:

```
parse.h + parse.c : parse.y
  yacc –d parse.y           # produce ytab.h & ytab.c
  copy ytab.h parse.h       # copy ytab.h to parse.h
  copy ytab.c parse.c       # copy ytab.c to parse.c
  del ytab.*                # remove the ytab files
```

Assume that **parse.y** was changed recently and that some other target depended on **parse.h**. Making this other target causes **parse.h** to be made, which executes the build scripts. If **parse.c** is evaluated later in the make process, **omake** recognizes that it is a member of the same target group as **parse.h**, which has already been updated, and does not rerun the build scripts.

3.5 Macros

A macro definition associates a name and a value. The macro expansion of a macro name returns the value. Macros are used at read time to organize names of files, compiler options, and so on. At run time, macros provide information about the current target being built.

Macro Modifiers

When a macro is referenced, the expanded value can be modified through the use of macro modifiers. To modify a macro, expand it with:

```
$(name,modifier[,modifier ...])
```

Everything between **$(** and **)** is expanded, the value of *name* is expanded, and then each *modifier* is applied in succession to the expanded value. The separator between *name* and *modifier* can be a comma or colon, but subsequent modifiers must follow a comma. A literal comma is available by using **\,**. Some modifiers use regular expressions. The treatment of the backslash, can make it awkward to use a modifier when a backslash also terminates its argument. You can work around this by using two backslashes before a comma separator(\\,) or
if possible, move the awkward modifier to the end. If this not possible, use two modifications. For example,

```
OBJS  =  $(SRCS,<obj\,R,>.obj)
```

does not work, but this pair of macro definitions does:

```
OBJS  =  $(SRCS,<obj\)
OBJS  :=  $(OBJS,R,>.obj)
```

Considering the macro value as a list of macro elements separated by white space, the first modifier is applied to all elements, then the next modifier is applied, and the next, and so on. The following list of modifiers is organized into functional groups. Examples of each modifier use the following macro definition:

```
SRCS = \src\main.c \sys\sub.cpp io.cpp
```

Filename Components

B The base name part of the element. This is the file name without an extension.
`$(SRCS,B) is main sub io`

D The directory part of the element. For **X:\***file*, **X:***file*, **\***file*, or *dir\file*, the directory part is **X:\**, **X:**, \, or *dir,* respectively. If there are no directory separators the directory part is the current directory (**.**).
`$(SRCS,D) is \src \sys .`

E The extension (or suffix) of the element. This is “” if there is no extension.
`$(SRCS,E) is .c .cpp .cpp`

F The file name part of the element. The part after the last directory separator.
`$(SRCS,F) is main.c sub.cpp io.cpp`

P The path part of the element. This is similar to the **D** modifier, but includes the last directory separator. So for **X:\***file*, **X:***file*, **\***file*, *dir\file*, or *file* the path part is **X:\**, **X:**, \, *dir***\** or “”, respectively.
`$(SRCS,P) is \src \sys`

R The root part of the element. The same as its full name minus its extension.
`$(SRCS,R) is \src\main \sys\sub io`

Z The drive letter part of the element (or the current drive), terminated by a colon
`$(SRCS,Z) is V: V: V: (if “V” is the current drive)`

Absolute Pathname

A[*sep*] Convert the element to an absolute pathname. If the element is not already an absolute path, this modifier prepends the current drive and working directory. Then, the *sep* separator replaces the slash and backslash. Finally, **..** and **.** directory parts are removed. For example, if the current working directory is **v:\\stage**, then:

```
$(SRCS,A) is v:\src\main.c v:\sys\sub.cpp v:\stage\io.cpp

$(SRCS,A/) is v:/src/main.c v:/sys/sub.cpp v:/stage/io.cpp
```

If *sep* is missing, the backslash is used.

Append and Prepend Strings

>*string* Appends *string* to each element of the macro.

```
$(SRCS,B,>.obj) is main.obj sub.obj io.obj
```

<*string* Prepends *string* to each element of the macro.

Change Case

LC Lowercase. Change the letters of the element to lowercase.

```
$(SRCS,Z,LC) is d: c: c:
```

UC Uppercase. Change the letters of the element to uppercase.

```
$(SRCS,B,>.obj,UC) is MAIN.OBJ SUB.OBJ IO.OBJ
```

Expand Pathnames

X This modifier expands elements into their pathnames. As an example:

```
.PATH.obj   = objs
OBJS           = 1.obj 2.obj
project.exe : $(OBJS)
  link $(.TARGET), $(OBJS,X);
```

The **.PATH.ob**j macro defines a search directory where **.obj** files are located. The **X** modifier causes **omake** to search for each element in its search directories. Assuming **1.obj** and **2.obj** are in the **objs** directory:

```
$(OBJS) is      1.obj 2.obj
$(OBJS,X) is  objs\1.obj objs\2.obj
```

Include File

@ If the macro's value names a file, the value of the macro modification is the contents of the file with spaces, tabs, and newlines collapsed to single spaces.
This modifier is best used with the **:=** expanded macro definition so the file or project is read only once, when the macro definition occurs. For example:

```
FILE =   link.rsp                          (response file)
OBJS :=  $(FILE,@)                         (read contents of link.rsp)
```

Include File with Regular Expression Matching

@d regex d subst d

Include file, with regular-expression matching. [A regular expression **.** matches any single character; a regular expression **\.** is a literal period (or dot). Several regular expression characters that **omake** uses are poor choices when the regular expression is used to match file names. **omake** allows you to redefine these characters with the **.REGEX_CHAR** and **.REGEX_WILD** directives.]

The macro's value is the name of a file to be read. Each line in the file is examined with the regular expression *regex*. If *regex* matches the line, the matched part of the line is replaced with the substitution string *subst*. If *regex* does not match, the line is skipped.

The *d* is a single-character delimiter that cannot appear in **regex** or *subst* and cannot be a comma (usually a slash (**/**) or single quote (**'**) is used).

Member and Nonmember

M*regex* The member selector. This selects elements matching regular expression *regex*.

```
$(SRCS,M\.c$$) is    \src\main.c
```

After macro expansion, **\.c$$** is **\.c$**, a regular expression that matches **.c** at the end of the element. This matches **.c** but not **.cpp**.

M"*spec*" The member selector, where *spec* is enclosed in double quotes and is a Windows NT file specification, not a regular expression:

```
$(SRCS,M"\src\main.c") is    \src\main.c
$(SRCS,M"*.cpp") is    io.cpp
```

One use of this modifier is to determine whether the current target belongs to some special list of names. For example:

```
SPEC_OBJS    = objs\spawn.obj objs\sync.obj
%.obj : %.c
  if '$(SPEC_OBJS,M"$(.TARGET)")'if $(.TARGET) is in SPEC_OBJS
special commands to handle special objects
endif
```

N*regex* The nonmember selector. It selects elements that do not match *regex*.

```
$(SRCS,N\.c) is    io.cpp
```

N"*spec*" The nonmember selector, where *spec* is a Windows NT file specification, not a regular expression:

```
$(SRCS,N"\src\main.c") is    \sys\sub.cpp io.cpp
$(SRCS,F,N"*.c") is    \sys\sub.cpp io.cpp
```

Select a Particular Element

number Selects the *number*th element of this macro value. If the macro value doesn't have this *number*th element, the modified value is the empty string. This modifier is particularly useful for inference rules that build multiple targets from a single dependency.

```
$(SRCS,2) is    \sub.cpp
```

String Substitution

Sd *regex d subst d*

The substitution modifier. The *d* is a single-character delimiter that is neither in regular expression *regex* nor in substitution string *subst* and isn't a comma. If *regex* matches some part of a macro element, the matched part of the element is replaced with *subst*. The element is not changed if *regex* does not match it.

If *regex* is omitted, the last regular expression is reused. This is useful in combination with the member modifier, **M**. For example, given that **CFLAGS** has the value **–AX –Ifoo –Ibar –DX=–IT**:

```
$(CFLAGS,M^-I,S'^-I'') is     foo bar
$(CFLAGS,M^-I,S''') is     foo bar
```

The **M** modifier selects elements that start with **–I**, and the **S** operator substitutes the **–I** with nothing. This is quite different from

```
$(CFLAGS,S'^-I'') is     -AX foo bar -DX=-IT
```

Notice how the elements unmatched by *regex* **^–I** are not changed.

*from***=***to*

This modifier replaces occurrences of *from* string with *to* string. If *from* does not appear in a macro element, the element is not changed.

from and *to* can be simple strings or can use the percent sign (%) as a wildcard. If **%** is used, the search for *from* is anchored to the end of the element.

```
$(SRCS,%.c=%.obj) is \src\main.obj \sys\sub.cpp io.cpp
$(SRCS,.c=.obj) is \src\main.\sys\obj sub.cpp io.cpp
```

The use of **%** anchors the search for **%.c** so it matches **main.c** but not **sub.cpp**. The simple string substitution **.c=.obj** replaces any occurrence of **.c** with **.obj** so matches and changes **sub.cpp** to **sub.objpp**.

The delimiter between strings is the equal sign, but it is treated literally when prefixed with a backslash (**\=**). If *fromString* ends with a backslash, you must use two (**\\=)** to mean literal backslash followed by a delimiter. (Under PM/CB or NMAKE emulation, the delimiter is not quotable. This means that **\=** is taken literally, and there is no need to double the backslash).

For historical reasons, when *fromString* and *toString* are simple strings, this modifier operates on the entire macro value rather than on its elements, and can be used to replace the white space between elements. However, the **W** modifier is better suited for this task.

Tokenize

W*str*

This modifier replaces the white space that separates macro elements with *str*, where *str* is treated literally except for the following special sequences:

\n	A newline character
\r	A return character
\t	A tab character
\\	A backslash
\ddd	An octal character ddd (1 to 3 digits)
\xdd	A hexadecimal character dd (1 or 2 digits)

This example can be used to prepare an inline response file with the **OBJS** definition of the **X** modifier listed above:

```
$(OBJS,X,W+\n) is     objs\1.obj+
     objs\2.obj
     project.exe;
```

Wildcard Expand File and Directory Names

*** or *F** The macro's value is a wildcard file specification used to match **file** names. The value of the macro modification is the list of files that matches the specification. This list includes any directory components. For example:

```
SPEC         =   obj\*.obj        (set up file spec)
OBJFILES     :=  $(SPEC,*F)       (get list of obj\*.obj files)
```

***D** Like the ***F** modifier but matches directory names. For example:

```
SPEC         =   *                (set up directory spec)
SUBDIRS      :=  $(SPEC,*D)       (get list of subdirectories)
```

Regular Expressions

The **M**, **N**, and **S** modifiers use pattern-matching strings known as regular expressions. The regular expression (or *regex*) matches characters in strings, both literally and with wildcards, and the match is case-sensitive.

Configuring Regular Expressions

Two directives control the special characters that appear in regular expressions:

➤ **.REGEX_CHAR** sets the character used to indicate special character sequences. The default character is a backslash (\).

➤ **.REGEX_WILD** sets the wildcard character that matches any single character. The default character is a period (**.**).

The backslash and period are standard characters in UNIX-styled regular expressions, but are awkward to use in Windows NT because the backslash is the directory separator and the period is the file extension separator. The **.REGEX_CHAR** and **.REGEX_WILD** directives can be used to change these characters to some other characters.

The default characters are most problematic for the **M** and **N** modifiers when you are trying to match a file name. For this case, use the **M**"*spec*" and **N**"*spec*" modifiers because the *spec* is a file specification rather than a regular expression.

Regular Expressions for the M Modifier

Assume the following macro definitions:

```
SRCS   = main.c sub.cpp io.cpp
CFLAGS = –AX –Ifoo –Ibar /Ibaz –DX=–IT xI.c yi.c
```

To select files whose names include **.c**:

`$(SRCS,M\.c)` is `main.c sub.cpp`

To select files that end in **.c**, the search can be anchored to the end with the regex character **$**. To get **$** to the regular expression, you need to use **$$** in the makefile:

`$(SRCS,M\.c$$)` is `main.c`

You can also select **.c** files with the **M**"*spec*" modifier:

`$(SRCS,M"*.c")` is `main.c`

Analogous to the **$** anchor, the **^** regex character anchors the search to the front of the macro element:

`$(CFLAGS,M-I)` is `-Ifoo -Ibar -DX=-IT`
`$(CFLAGS,M^-I)` is `-Ifoo -Ibar`

The [*set*] regex characters indicate a *set* of characters, where the *set* can be single characters (**[aA@]** matches **a**, **A** or @), a range of characters (**[a-z]** matches **a** through **z**), or a mixture. For example:

`$(CFLAGS,M^[-/]I)` is `-Ifoo -Ibar /Ibaz`

Regular Expressions for the S Modifier

One powerful feature of regular expressions is that when used in substitutions, they can access the matched parts of the string. The down side of the more powerful regular expressions is that the expressions can be hard to read. For example, when `DIR = NT_L`, the expression `$(DIR,S/\(.*\)_.*/\1/)` is `NT`.

The **\(**, **\)** pair surround a part of the regular expression that is referenced later. Inside the pair is **.*,** which matches any character (**.**) repeated zero or more times (*). Taken together they instruct **omake** to match any character, zero or more times, and attach a tag to it. The rest of the regular expression is _ , which matches _, and **.***, which matches any character repeated zero or more times.

The expression **\1** is the stuff matched in the first pair of \(\), so the substitution evaluates to NT. Simple regular expressions are easy to read, but if you write more complicated expressions, be sure to provide thorough comments.

With Configuring .REGEX_CHAR and .REGEX_WILD

If **omake** is configured for alternative regular expression characters, the following directives appear in your **make.ini**:

```
.REGEX_CHAR  : ~
.REGEX_WILD  : ?
```

These examples are then easier to read:

```
$(SRCS,M.c) is main.c sub.cpp
$(SRCS,M.c$$) is main.c
$(CFLAGS,M–I) is –Ifoo –Ibar –DX=–IT
$(CFLAGS,M^–I) is –Ifoo –Ibar
$(CFLAGS,M^[–/]I) is –Ifoo –Ibar /Ibaz
$(DIR,S/~(?*~)_?*/~1/) is NT
```

Predefined and Built-In Macros

omake defines several macros before it reads the initialization file and makefiles. These macros come in two general types:

- Predefined macros cannot be changed or undefined. Important subsets of the predefined macros are run-time macros, which change with the target being built, and state macros, which hold the state of the command-line options and read-time directives.
- Built-in macros have the same precedence as normal makefile macros, and so can be redefined in your makefiles or on the command line.

Predefined Macros: Run-Time Macros

The run-time macros are changed dynamically by **omake** according to the current target being built. A key feature of the run-time macros is that their values are *pathnames*. A target's pathname is the location the target was found on disk. The run-time macros evaluate to pathnames, which is important for use on build scripts where the executed command needs the location of the targets.

All run-time macros have names of the form **.***name*, and several have one- or two-letter aliases that you are likely to see in makefiles of other make utilities.

Run-Time Macros

.NEWSOURCES (alias is **?**)

This run-time macro evaluates to the pathnames of all dependencies newer than the current target. When configuration lookup is enabled, it evaluates to the pathnames of all dependencies for the current target, unless that behavior is modified with the **.INCREMENTAL_TARGET** directive (see the description in the section *Dot Directives* on page 91), in which case it evaluates to the pathnames of all dependencies different from the previously recorded versions.

.SOURCE (alias is **<**)

This run-time macro evaluates to the *pathname* of the inferred dependency that satisfied the inference rule, or the pathname of the first explicit dependency if no inferred dependency was found. Here is a sample inference rule for updating an object file with the Borland C compiler:

```
%.obj : %.c
  cl –c $(.SOURCE)
```

Using an alias, the second line becomes

```
  cl –c $<
```

.SOURCES (aliases are **^** or **)

This run-time macro evaluates to the *pathnames* of all dependencies of the current target. Here is an example that updates an executable with all object files that are its dependencies:

```
project.exe : main.obj io.obj
  link /out :$(.TARGET), $(.SOURCES);
```

Using an alias, the second line becomes

```
link /out : $@, $**;
```

NOTE: The ** macro is the exception to the rule that multicharacter macros need parentheses or braces around them. The expression **$**** is the same as **$(**)**.

.TARGET (alias is **@**)

This run-time macro evaluates to the *pathname* of the target currently being made. For example:

```
%.obj : %.c
cl –o$(.TARGET) –c $(.SOURCE)
```

Using an alias, the second line becomes

```
cl –o$@ –c $<
```

This macro can also be used at read time on a dependency line and its value is the name of the target currently getting dependencies. For example:

```
main.obj io.obj : $(.TARGET,R).c io.h
```

Using an alias, the line becomes

```
main.obj io.obj : $(@,R).c io.h
```

The **.TARGET** macro value is set to the name of each target, in turn. The dependencies are then macro-expanded and added to the target's list of dependencies. The **R** macro modifier selects the root of the target name. The above dependency line is equivalent to the two lines:

```
main.obj : main.c io.h
io.obj : io.c io.h
```

For compatibility with other Make utilities. **omake** treats **$$@** on the dependency side of a dependency line just like **$(.TARGET)**.

.TARGETROOT (alias is *)

This run-time macro evaluates to the *pathname* of the target currently being made, minus its extension. This macro can also be used at read time on the dependency side of a dependency line. Here is the read-time example from the **.TARGET** macro rewritten using this macro:

```
main.obj io.obj : $(.TARGETROOT).c io.h
```

Using an alias, the line becomes

```
main.obj io.obj : $*.c io.h
```

Predefined Macros: General Macros

The predefined macros in Table 3 give you information about the current make process. One of the most important predefined macros is the status macro at the bottom of the table.

Table 3 General Macros (Part 1 of 2)

General Macro	Value
$	When you need a literal dollar sign, you must use **$$**.
BUILTINS	The name of the initialization (built-ins) file.

Table 3 General Macros (Part 2 of 2)

General Macro	Value
CWD	The current working directory (same as **MAKEDIR**).
FIRSTTARGET	The first command-line target or, if one isn't given, the default makefile target.
INPUTFILE	The name of the current makefile. **omake** can read multiple makefiles, either with multiple **–f** command-line options or with **%include** directives. The **INPUTFILE** macro's value is the current makefile being read.
MAKEARGS	The command line with which **omake** was started, including command-line options, macros, and targets.
MAKEDIR	The current working directory (same as **CWD**).
MAKEMACROS	The command-line macros with which **omake** was started. Any macros that have spaces are enclosed in double quotes so they can be used on a command line again.
MAKESTATUS	The exit status of **omake**. This can be used in the **.AFTER** special target to determine whether **omake** is exiting with an error.
MAKETARGETS	The list of command-line targets passed to **omake**.
MAKEVERSION	The version number for **omake**. Its format is *X***.***Y*, where *X* and *Y* are the major and minor release numbers.
OPUS	Defined to the value 1. This can be used to test if you are running **omake** or some other Make.
status	The exit status of the last build script. It is used with conditional directives to execute other build scripts. For historical reasons, the name is lowercase.

Predefined Macros: State Macros

The state macros provide information about the state of command-line options and read-time directives. Most command-line options have an equivalent directive, and the value of the state macro is the same regardless of whether the option or directive was used.

In the list of macros in Table 4, the macro values that are the state of a directive or command-line option mean the value is 1 if the directive or command-line option was used; otherwise the value is 0.

NOTE: In terms of implementation, **.ALWAYS**, **.IGNORE**, and **.SILENT** are not directives; they are target attributes. They look like directives when they appear in a makefile on the target side of a dependency line when there are no dependencies. For example,

```
.IGNORE :
```

sets the **.IGNORE** attribute for every target created after the appearance of this line. The **.ALWAYS**, **.IGNORE**, and **.SILENT** macros have the correct value, as if these attributes were directives.

NOTE: Exactly one of **$(.MS_NMAKE)**, **$(.omake)** or **$(.POLY_MAKE)** is 1.

Table 4 State Macros (Part 1 of 3)

State Macro	Value
.ALWAYS	The state of the **.ALWAYS** directive and **–a** command-line option. [In terms of implementation, **.ALWAYS**, **.IGNORE**, and **.SILENT** are not directives but target attributes. They look like directives when they appear in a makefile on the target side of a dependency line when there are no dependencies. For example, `.ALWAYS :` sets the **.ALWAYS** attribute for every target created after the appearance of this line. The **.ALWAYS**, **.IGNORE**, and **.SILENT** macros have the correct value, as if these attributes were directives.]
.CASE_MACRO	The state of the **.CASE_MACRO** directive.
.CASE_TARGET	The state of the **.CASE_TARGET** directive.
.DEBUG	The debug options as set by the **.DEBUG** directive and **–#** command-line option.
.DEBUG_PRINT	The state of the **.DEBUG_PRINT** directive and **–p** command-line option.
.DEBUG_RUN	The state of the **.DEBUG_RUN** directive and **–d** command-line option.
.ENVMACROS	The state of the **.ENVMACROS** directive.

Table 4 State Macros (Part 2 of 3)

State Macro	Value
.ENV_OVERRIDE	The state of the **.ENV_OVERRIDE** directive and **–e** command-line option.
.IGNORE	The state of the **.IGNORE** directive and **–i** command-line option.
.IGNORE_MFLAGS	The state of the **–z** command-line option.
.KEEPDIR	The state of the **.KEEPDIR** directive and **–D** command-line option.
.KEEPWORKING	The state of the **.KEEPWORKING** directive and **–k** command-line option.
.MAKE_MAKEFILE	The state of the **.MAKE_MAKEFILE** directive and **–M** command-line option.
.MS_NMAKE	The state of the **.MS_NMAKE** directive (exactly one of **$(.MS_NMAKE)**, **$(.omake)** or **$(.POLY_MAKE)** is 1) and **–EN** command-line option.
.OPUS_52X	The list of Opus Make v5.2x compatibility features, as set by the **–E2** command-line option.
.omake	The state of the **.omake** directive and **–EO** command-line option.
.POLY_MAKE	The state of the **.POLY_MAKE** directive and **–EP** command-line option.
.QUERY	The state of the **–q** command-line option.
.REGEX_BACK	The regular expression literal backslash (\). This is provided for writing regular expressions that are independent of the value of **.REGEX_CHAR**. If **.REGEX_CHAR** is **\\,**its value is \\\\; otherwise, its value is \\.
.REGEX_CHAR	The regular expression escape character set by the **.REGEX_CHAR** directive.
.REGEX_DOT	The regular expression literal dot (or period). If **.REGEX_WILD** is **.** its value is **${REGEX_CHAR}**.; otherwise its value is **.**
.REGEX_WILD	The regular-expression wildcard character set by the **.REGEX_WILD** directive.

Table 4 State Macros (Part 3 of 3)

State Macro	Value
.REJECT_RULES	The state of the **.REJECT_RULES** directive and **–r** command-line option.
.REREAD	The state of the **.REREAD** directive.
.RULE_CHAR	The regular expression character as set by the **.RULE_CHAR** directive.
.SHELL	The command to execute the shell program as set by the **.SHELL** directive.
.SILENT	The state of the **.SILENT** directive or **–s** command-line option.
.SUFFIXES	The list of suffixes as set by the **.SUFFIXES** directive.
.UNIXPATHS	The state of the **.UNIXPATHS** directive.

An Example Use of the State Macros

Here is an example of how these macros can be used. Suppose you want to debug the contents of your makefile with the **–#1** command-line option. However, this also produces output that comes from your **make.ini** file, which probably doesn't need debugging. Modify your **make.ini** file with the following:

```
_OLD_DEBUG := $(.DEBUG)                 (get current state)
.NODEBUG : 1                            (turn off "-#1" if set)
[Original make.ini goes here]
.DEBUG : $(_OLD_DEBUG)                  (restore state)
```

Built-in Macros

omake defines the built-in macros with a default value, but they have the same precedence as normal makefile macros, so you can be redefine them.

Table 5 Built-in Macros (Part 1 of 2)

Built-in Macro	**Definition**
CC	Used in the **%.obj : %.c** rule, the name of the C compiler. The default value is **cl**, the name of the Microsoft C Compiler.
FC	Used in the **%.obj : %.for** rule, the name of the FORTRAN compiler. The default value is **f77l**, the name of the Lahey FORTRAN Compiler.
LIBEXE	Used in the **%.lib : %.obj** rule, the name of the object librarian. The default value is **lib**, the name of the Microsoft Librarian.
LINK	Used in the **%.exe : %.obj** rule, the name of the object linker. The default value is **link**, the name of the Microsoft Linker.
MAKE	The value is the full pathname of the **omake** executable. The **MAKE** macro is special in that its appearance on a build script overrides the **–n** (no execute) command-line option for that line. This can be used in a recursive make, to have **omake** do the recursion.
MAKE_TMP	The name of the directory **omake** uses for temporary files (response files under Windows NT). Initially **MAKE_TMP** is undefined and **omake** uses the current directory for temporary files. If you define **MAKE_TMP** its value must be an absolute directory (such as **D:\** or **D:\\tmp**, preferably on a RAM disk. As an example, the environment variable TMP names the temporary directory some compiler vendors use. You can put the following in **make.ini**: `MAKE_TMP = $(TMP)`

Table 5 Built-in Macros (Part 2 of 2)

Built-in Macro	Definition
MFLAGS	After all makefiles have been read, **omake** defines **MFLAGS** with all the command-line options. **MFLAGS** is useful for invoking **omake** recursively, as shown in the definition of the MAKE macro. NOTE: **MFLAGS** is a built-in macro only if you are using native **omake** mode (that is, if you are not using an emulation mode). In the emulation modes, it is not predefined by **omake**; however, you can redefine it. A second usage of **MFLAGS** is to pass initial options to **omake**. If **MFLAGS** is defined in the environment or in **make.ini** its value specifies additional command-line options. For example, the following macro definition placed in **make.ini** turns on the keep-directory mode (the **–D** command-line option): `MFLAGS = –D` Because there are directives for all command-line options, this second usage of **MFLAGS** is discouraged, and you should use the directives.
OS	The operating system. Its value is **NT**.
RC	Used in the **%.res : %.rc** rule, the name of the resource compiler. The default value is **rc**, the name of the Microsoft Resource Compiler.
SHELLCOMMANDS	Alphabetical list of commands known to need execution by the shell program. If you define this macro, **omake** uses it to detect when to use the shell program. This macro is not defined initially; **omake** uses an internal list of commands. See *Auto-Detection Mode* on page 19.
SHELLSUFFIX	Suffix used by **omake** for the batch files it generates. The default value is **.bat**.

Compatibility with Other Make Utilities

Through emulation, **omake** supports all PM/CB and NMAKE macros.

3.6 Build-Script Line Prefixes

Build-script line prefixes control some aspects of a build script's execution. The prefixes occur on a build script before the program name. The first nonprefix character ends the prefixes. The bar (|) prefix can also be used to mark the end explicitly. White space is allowed between prefixes, and between the prefixes and the program name. Build-script lines are macro-expanded before prefixes are detected, so you can use macros to define prefixes.

Do Not Echo the Build-Script Line (Silent Operation)

@

Usually **omake** echoes (displays) the build-script line to the screen before executing it. The @ prefix prevents this display except when running **omake –n**.

Here is an example of its use:

```
ERRS = make.out

$(OBJS) : $(.TARGETROOT).c
  %do CCquiet

CCquiet :
  @ %echo –n Compiling $(.SOURCE) ...
  @ $(CC) $(CFLAGS) –c $(.SOURCE) >>$(ERRS) 2>&1
  @ %echo done.
```

Redirection by **>>$(ERRS) 2>&1** means error messages from the compiler go into the **$(ERRS)** file. The net effect is that a series of messages of the form:

```
Compiling file ... done.
```

are displayed and all output from the compiler is redirected into **make.out**.

See also the **.SILENT** target attribute to turn on silent mode for this target and the **–s** command-line option to enable silent mode for all targets.

@@

The @@ prefix prevents display of the build script before execution, even when running **omake –n**.

Ignore the Build-Script Line Exit Status

–[*num*]

Normally a build-script line that returns a nonzero exit status causes **omake** to terminate with this message:

omake`: Shell line exit status exit_status. Stop`

The dash (**–**) prefix causes **omake** to ignore the exit status returned from the build-script line and to continue. The message now becomes

omake`: Shell line exit status exit_status (ignored)`

If *num* is given, **omake** ignores the exit status if it is no greater than *num*. For example, the prefix **–4** tells **omake** to ignore an exit status of 1, 2, 3, or 4. There must be at least one space after *num*.

Importantly, the *status* macro is set to the exit status of the build-script line so it can be tested later. See the **--** prefix for an example.

See also the **.IGNORE** target attribute to turn on ignore mode for this target and the **–i** command-line option to turn on ignore mode for all targets.

--[*num*]

Ignores the exit status and does not display the warning message.

If *num* is given, **omake** ignores the exit status if it is less than or equal to *num*.

For example, the **CCquiet** target of the previous example can be changed to this:

```
CCquiet :
  @ %echo –n Compiling $(.SOURCE) ...
  @– – $(CC) $(CFLAGS) –c $(.SOURCE) >>$(ERRS) 2>&1
%if $(status) != 0
  % abort FAILED! See $(ERRS) for errors.
  %endif
  @ %echo done.
```

Now a failed compile generates this message:

```
Compiling file ... FAILED! See make.out for errors.
```

~[*num*]

Ignores the exit status and displays the warning, but the **status** macro retains its current value and is not set to the exit status of this build script.

~~[*num*]

Ignores the exit status, does not display the warning, and retains **status**.

Override the –n Command-Line Option

&

Use **&** to override the **–n** command-line option for this build script. This prefix is useful if you are using **omake** to call itself recursively.

It is usually better to use the **.MAKE** target attribute instead because it overrides the **–q** and **–t** command-line options as well as **–n**.

Select the Shell Program

omake executes all build-script lines with the shell program. This default can be changed with the **.SHELL** directive, which selects use of the shell program, and the **.NOSHELL** directive, which selects direct execution.

The following prefixes override the general execution mode for the current build script:

:	Executes the build script directly.
+	Executes build scripts by the shell program.

Iterate the Build Script

!

Iterates the build-script line for each element of **$?** (i.e. **$(.NEWSOURCES**)) or **$**** (that is, **$(.SOURCES**)), on the basis of which appears first on the build-script line. The build-script line is executed once for each element of the original **$?** (**$****) with the value of **$?** (**$****) taking on successive elements of the original **$?** (**$****).

If neither **$?** nor **$**** appears on the build-script line, the build-script line is iterated once for each element of **$****.

omake performs iteration explicitly over any number of build scripts with its **%foreach** directive, and the use of the **!** prefix is discouraged.

Miscellaneous Prefixes

|

The bar (|) prefix can be used to mark the end of the prefixes. This is useful when the command to be executed starts with one of the prefixes.

>

The right-angle bracket (>) prefix inserts an extra carriage return/linefeed after the build script is executed. This prefix is supplied for PM/CB compatibility.

Build Script Compatibility with Other Make Utilities

For PM/CB compatibility, **omake** supports named prefixes such as (Silent) and (Ignore).

3.7 Build-Script Problems: The cd and set commands

Each build-script line is executed in a separate shell and each build-script line starts in **omake**'s current directory. This causes special problems for some commands. For example, suppose you want to change directory to **subdir** and run **omake** recursively. If you use this line

```
recursion :
  chdir subdir
  omake $(MFLAGS)
```

omake $(MFLAGS) is executed from the current directory.

Similarly, setting environment variables does not work as expected because the environment variable is set in one shell, the shell exits back to **omake**, and the next command is executed in a new shell that doesn't have the variable set. For example, this line

```
setenv :
  set USER=dgk                       (for UNIX: setenv USER dgk)
  echo %USER%                        (for UNIX: echo $$USER)
```

does not echo `dgk`.

NOTE: In **NMAKE** emulation mode, **omake** handles internally any commands that start by setting an environment variable, rather than calling the shell.

Both of these problems can be avoided by using a multiple-command build-script line.

Using Multiple-Command Build-Script Lines

The syntax for executing more than one command in the same shell is

```
( command [ & command ] ... )
```

The build-script line is enclosed in parentheses, and the commands are separated by a semicolon (**;**). (Use **\;** to specify a literal semicolon). In terms of implementation, **omake** writes each command to a batch file and uses the shell program named by the **.SHELL** directive to execute the batch file.

The recursion examples now become

```
recursion :
  ( chdir subdir & omake $(MFLAGS) )
```

Simultaneous support of all operating systems is possible with a conditional macro definition such as this:

```
%if $(OS) == NT
; = &$; is ″&″
%else
; = ;$; is “;″
%endif
```

The multiple-command build-script line is now written as

```
( chdir subdir $; omake $(MFLAGS) )
```
(for all OSes)

After the macros are expanded, the build-script line becomes

```
( chdir subdir ; omake
command_line_flags )
```
(if OS is not Windows NT)

```
( chdir subdir & omake
command_line_flags )
```
(for Windows NT)

Using Directives: %chdir and %setenv

Rather than use a multiple-command build script to work around the change-directory problem, you can use a directive. The **%chdir** *directory* directive causes **omake** to change to *directory*, where it stays until the next **%chdir** directive or until **omake** exits. The recursion example becomes

```
recursion :
  %chdir subdir                        (change to the subdirectory)
  omake $(MFLAGS)                      (do the recursive Make)
  %chdir $(MAKEDIR)                    (change the directory back)
```

Likewise, to work around the environment-variable problem you can use a directive. The **%setenv name val** directive sets environment variable **NAME** to the value *val*. Like **%chdir**, **%setenv** has effect until the next **%setenv** or until **omake** exits. The **setenv** example becomes

```
setenv :
  %setenv USER dgk
  echo %USER%                          (for unix: echo $$USER)
```

For more information and recommendations on using the **%setenv** directive, see the **%setenv** entry in Table 14 on page 90

3.8 Makefile Directives

Makefile directives provide additional control over **omake**. There are two directive types:

- ➤ Percent directives, which use a percent sign (%) as the first non-white-space character on the makefile line.
- ➤ Dot directives, which look like dependency lines but use special target names.

Percent Directives

Percent directives (usually called directives) have names of the form %*name*, where the directive character is the first non-white-space character on the makefile line. White space is allowed between % and *name*.

These directives work as read-time directives or run-time directives with indentation determining whether they are interpreted at read time or at run time. If % is in the leftmost column of the line, the directive is interpreted at read time. If white space precedes the %, the directive is interpreted at run time.

Conditional Directives

The directives in Table 6 control the flow of the make process at read time and at run time. Conditional and iteration directives can be nested up to 31 levels deep.

Table 6 Conditional Directives

Conditional Directives	Action	Applicable Time
% **if** *condition*	Start a conditional block	read/run
% **ifdef** *macro*	Start a conditional block	read/run
% **ifndef** *macro*	Start a conditional block	read/run
% **elif** / **elseif** *condition*	Continue the conditional block	read/run
% **else**	Start the default conditional block	read/run
% **endif**	End the conditional block	read/run

Iteration Directives

The directives in Table 7 provide iteration control. Iteration and conditional directives can be nested up to 31 levels deep.

Table 7 Iteration Directives

Iteration Directives	Action	Applicable Time
% **foreach** *name* [in] *list*	Loop for each *name* in *list*	read/run
% **while** *condition*	Loop while *condition* is true	run
% **end**	End the loop	read/run
% **break**	Interrupt and quit innermost loop	run
% **continue**	Interrupt and restart innermost loop	run

Other Percent Directives

Other directives behave like commands built into **omake**. At run time, these directives can be preceded with build script prefixes, which must appear before the %. See Table 8.

Table 8 Other Percent Directives

Other Directives	**Action**	**Applicable Time**
% **abort** [*status*] [*message*]	Display *message*, exit with *status*	read/run
% **chdir** *directory*	Change current directory	run
% **do** [*target*] [*macros*]	Execute build scripts of *target*	run
% **echo** [–n] *message*	Display *message*	read/run
% **error** [*status*] [*message*]	Display *message*, return exit *status*	read/run
% **exec** *command*	Execute *command* line	read
% **include** [<"] *file* [>"]	Include contents of *makefile*	read
% **restart** [*flags*]	Start **omake** again	read/run
% **set** *name* [+]= *value*	Set macro *name* with *value*	run
% **setenv** *name* [=] *value*	Set environment variable **NAME**	read/run
% **undef** *name*	Undefine macro **name**	read/run

Conditional Directives

Conditional directives control the makefile lines that **omake** reads at read time and control the build scripts that are executed at run time. These directives take this form:

```
%if condition
  [ makefile line ]
  .
  .
  .
[ %elif condition ]
  [ makefile line ]
  .
  .
  .
[ %else ]
  [ makefile line ]
  .
  .
  .
%endif
```

Each **%if** must be matched with an **%endif**. There can be several **%elif** or **%elseif** clauses and at most one **%else**. Lines between **%if** or **%elif** and the next **%elif**, **%else**, or **%endif** are a block. If *condition* is true, the makefile lines are read or the build-script lines are executed. If none of the conditions is true and there is a **%else** clause, the block between **%else** and **%endif** is read or executed.

If the % of these directives is in the leftmost column of the makefile, the directive is evaluated at read time; otherwise, the directive is evaluated at run time. There can be white space between % and the name of the directive. No shell-line prefixes are allowed before the %.

There are two specialized version of **%if**:

%ifdef *name*
%ifndef *name*

%ifdef is true if macro *name* is defined. **%ifndef** is true if macro *name* is not defined.

Conditional Directives and Continued Lines

Directives can be used to conditionally select the continued makefile text. For example:

```
OBJS = main.obj parse.obj \               (the continuing line)
%ifdef Debugging
  version.obj \                           (continues here if Debugging  defined)
  mymalloc.obj
```

```
%endif
a blank line                                  (continues here if Debugging undefined)
```

If the Debugging macro is not defined, the **OBJS** macro value is **main.obj parse.obj**. If the macro is defined, the value is **main.obj parse.obj version.obj mymalloc.obj**.

The way this example is written makes the blank line after the **%endif** necessary because the continuing line is continued on the first line after the **%endif** if **Debugging** is undefined. Using an **%else** clause instead is done like this:

```
OBJS = main.obj parse.obj \                   (the continuing line)
%ifdef Debugging
  version.obj \                               (continues here if Debugging defined)
  mymalloc.obj
%else
                                              (continues here if Debugging undefined)
%endif
```

Conditional Expressions

The **%if**, **%elif**, and **%while** directives use the same conditional expressions. These expressions compare strings and numbers and combine the comparisons with logical operations. As well, the status of macros, command-line targets, and files can be determined. All macro references in the conditional expression are expanded before the expression is evaluated.

The conditional expressions here are organized by type. Some examples follow each type. For the examples, the following macros are assumed to be defined:

```
DEBUG    = 0
MODEL    = S
NONE     =
OBJS     = 1.obj 2.obj
```

Simple Expressions

You can use any of the following simple expressions:

value
'*value***'**
"*value***"**

If *value* is zero, the condition is false; all other values indicate true. Single or double quotes must be placed around *value* if it contains spaces or is null.

Some examples of simple expressions:

```
%if ASTRING    value is true
%if 12         value is true
%if 0          value is false
%if $(MODEL)   value is true
%if $(DEBUG)   value is false
%if $(NONE)    ERROR! $(NONE) is null
%if "$(NONE)"  value is false
```

Comparison Operators

The operators in Table 9 make a numerical or alphabetical comparison of two values, returning 1 if the comparison is true and 0 if false.

Table 9 Comparison Operators

Comparison Operators	**Value**
value1 = = *value2*	True if *value1* is equal to *value2*.
value1 != *value2*	True if *value1* is not equal to *value2*.
value1 < *value2*	True if *value1* is less than *value2*.
value1 <= *value2*	True if *value1* is less than or equal to *value2*.
value1 > *value2*	True if *value1* is greater than *value2*.
value1 >= *value2*	True if *value1* is greater than or equal to *value2*.

If both values start with a digit, the comparison is done numerically; otherwise, it is done alphabetically. If either value contains spaces or is null, it must be enclosed in quotes. The case-sensitivity of the string comparison is the same as for target names and can be set with the **.CASE_TARGET** and **.NOCASE_TARGET** directives.

Some examples of comparison operators:

```
%if $(MODEL) == S   true
%if ABC > DEF       false
%if $(DEBUG) != 0   false
%if $(XYZ) == 1     error! $(XYZ) is null
%if $(XYZ)x == 1x   false
%if $(OBJS)x != x   error! $(OBJS) has spaces
%if "$(OBJS)" != "" true
```

Functional Operators (Also Called Built-In Functions)

All function operators take one or more arguments and return a value that is false (0), true (1) or some other number. See Table 10.

Table 10 Functional Operators

Functional Operators	**Value**
%**defined**(*name*)	True if macro *name* is defined.
%**dir**(*name*)	True if *name* is a directory.
%**exists**(*name*)	True if *name* is a file or directory.
%**file**(*name*)	True if *name* is a file.
%**length**(*name*)	The number of characters in $(*name*).
%**make**(*name*)	True if *name* is a command-line target.
%**member**(*name*, *list*)	True if *name* appears exactly as an element of *list*.
%**null**(*name*)	True if macro *name* is undefined or if its expansion is null.
%**time**(*name*)	The on-disk time stamp of file *name*.
%**writable**(*name*)	True if file *name* is writable

File-Test Operators

The file-test operators return the state of files and directories. Most of these operators have been made obsolete by the equivalent functional operators. See Table 11.

Table 11 File-Test Operators

File-Test Operators	**Value**
–d *name*	Same as %**dir**(*name*).
–e *name*	Same as %**exists**(*name*).
–f *name*	Same as %**file**(*name*).
–r *name*	Same as %**file**(*name*).
–w *name*	Same as %**writable**(*name*).
–z *name*	True if *name* is a zero-length file.

One example of the file-test operators:

```
%if –e builtins.mak                         (true if builtins.mak exists)
```

Command-Execution Operator

The command-execution operator executes a command using the shell program. See Table 12.

Table 12 Command-Execution Operator

Execution Operator	**Value**
[*command*] where the brackets are literal.	The exit status of the *command*. By convention, commands return 0 if they succeed.

Here is an example that runs a hypothetical **mkproto** program that freshens the prototype files for its source files:

```
%if [ mkproto *.c ]
% abort MKPROTO could not freshen the prototypes.
%endif
```

You can ignore the exit status of the executed command by using an **%if ... %endif** pair that doesn't encapsulate anything. The following idiom is seen in VC++ makefiles:

```
!if [ if exist MSVC.BND del MSVC.BND ]
!endif
```

Logical Operators

The logical operators are used to combine other expressions. Each expression is evaluated from left to right, but parentheses can be used to order the evaluation. See Table 13.

Table 13 Logical Operators

Logical Operators	**Value**
exp1 && *exp2*	True if both *exp1* and *exp2* are true.
exp1 || *exp2*	True if either *exp1* or *exp2* are true.
! *exp*	True if *exp* is false. False if *exp* is true.
(*exp*)	The same state as *exp*.

Unlike the C programming language, the logical expressions do not short-circuit. That is, *exp1* and *exp2* are always evaluated. For example:

```
%if %defined(NONE) && %null(NONE)       (true)
%if %defined(XYZ) && %null(XYZ)         (false)
%if ! ( ! -f file || -z file )          (false if file is absent or is zero length)
```

Iteration Directives

The **%while** and **%foreach** directives provide iteration capability. At run time, they allow build scripts to be executed multiple times. **%foreach** can also be used at read time to replicate makefile lines.

The %foreach Directive

The form of the **%foreach** directive:

%foreach *name* [**in**] *list_of_names*
  [*makefile lines*]
  .
  .
  .
%end

The *list_of_names* is a set of names separated by white space. The value of macro *name* is set to the first name in the *list_of_names*, and the makefile lines after this **%foreach** and before **%end** are read (at read time) or executed (at run time).

If the % of these directives is in the leftmost column of the makefile, the directive is evaluated at read time; otherwise, the directive is evaluated at run time. No shell-line prefixes are allowed before **%foreach** or **%end**.

When **%end** is reached, *name* is set to the next name in *list_of_names*, and the loop is restarted. When there are no more names, the loop is done and *name* returns to its previous value. For PM/CB compatibility, the optional keyword **in** is supported and **%endfor** can be used in place of **%end**.

If **%foreach** is used at read time, the makefile lines between **%foreach** and **%end** are parsed multiple times with *name* taking on successive values from the *list_of_names*. As the lines are parsed, all macro references to *name* are expanded.

The following example illustrates the use of **%foreach** at read time:

```
%foreach var in main sub io
macro_$(var) = $(value_$(var)) $(other_macro)
$(var).obj : $(var).c
  cl –c $(.SOURCE)
%endfor
```

All **$(***var***)** references are replaced with the current value of *var*. **omake** treats the previous **%foreach** loop as if it has read the following makefile lines:

```
macro_main = $(value_main) $(other_macro)
main.obj : main.c
  cl –c $(.SOURCE)

macro_sub = $(value_sub) $(other_macro)
sub.obj : sub.c
  cl –c $(.SOURCE)
```

```
macro_io = $(value_io) $(other_macro)
io.obj : io.c
  cl –c $(.SOURCE)
```

Here is an example run-time use of **%foreach** to build a linker response file using the inline response file syntax (see *Inline Response Files* on page 115):

```
project.exe : $(OBJS)
  link @<<                                (link @response_file)
%foreach x in $(.SOURCES)
$x                                        (adds each .obj +)
%end
                                          (adds a blank line)
$(.TARGET);                               (add the executable name)
<<                                        (end the response file)
```

The %while Directive

The form of the **%while** directive:

%while *condition*
  [*build script and directives*]
  .
  .
  .
%end

While *condition* is true, the build script and directives are executed. When **%end** is reached the loop is restarted at the top, *condition* is reexpanded (because it may contain macros) and retested. This is repeated until *condition* is false. No shell-line prefixes are allowed before **%while** or **%end**.

Effects of %foreach and %while when Using omake in a VOB

When you use **omake** in a VOB, there is a side effect with the **%foreach** and **%while** directives: the build script in the CR does not expand loop macros. For a makefile like this one,

```
all:
   %foreach platform in i386,alpha
      $(MAKE) -f makefile.$(platform)
   %endfor
```

you may expect the configuration record to include the commands that were executed. For example:

```
omake -f makefile.i386
omake -f makefile.alpha
```

Instead, you get a copy of the build script:

```
%foreach platform in i386,alpha
omake.exe -f makefile.
%endfor
```

Note also that the platform macro has a value only while inside the loop, so the macro in the configuration record is expanded to nothing.

Interrupting the Iteration

%break and **%continue** interrupt the iteration. **%break** stops the iteration immediately. **%continue** restarts the iteration at the top. For **%while**, the *condition* is reexpanded and retested. For **%foreach** the *name* is advanced to the next in the *list_of_names*.

Another Method of Iteration

The **!** shell-line prefix iterates the build script for elements of either the **.SOURCES** (or **) or **.NEWSOURCES** (or **?**) macro, depending on which appears first on the build script. During iteration, **${.SOURCES}** (or **$****) or **${.NEWSOURCES}** (or **$?**) evaluates to each element, in turn, of the macro value. We discourage the use of **!** for iteration, preferring the explicit **%foreach** directive.

A Sample Makefile

Here is a sample makefile that uses conditional and iteration directives:

```
%if defined(CV)                           (Read-time conditional)
  CFLAGS = –Od –Zi
%else
  CFLAGS = –Ox
%endif


io.obj : io.c
  %while 1                                (Loop forever)
   ––$(CC) $(CFLAGS) –c io.c >io.err      (Compile and gather errors)
2>&1
   %if $(status) == 0                     (Run-time conditional)
  %break                                  (Quit if no compiler errors)
   %endif
```

```
 notepad –mnext_error io.c            (Else edit the file)
 %if %time(io.c) < %time(io.err)      (Was io.c written?)
%error Compile of $(.SOURCE) failed   (No, exit with an error)
 %endif
%end
erase io.err                          (Remove the error file)
```

This example shows a compile-edit loop. The file **io.c** is compiled with all compiler messages redirected into file **io.err**. The **2>&1** redirection is compatible with Windows NT and UNIX. It means that standard error is redirected to standard output. The previous **> io.err** means standard output is redirected into **io.err**. Together, they mean that standard error and standard output both are redirected to **io.err**.

If the compile is successful, the **%break** directive breaks out of the **%while 1** loop and erase the **io.err** file.

Otherwise, **notepad** (the Notepad editor) is started and told to jump to the first error. After Notepad finishes, check the time of the **io.c** file to see whether it is more recent than **io.err**, indicating it has been changed. If it hasn't, **omake** breaks out of the compile-edit loop by calling the **%error** directive. Otherwise, the loop restarts and the file is recompiled.

Other Percent Directives

Table 14 lists other percent directives, the times at which they are applicable, and their descriptions.

Table 14 Other Percent Directives (Part 1 of 6)

Directive	Appl. Time	Description
%**abort** [*status*] [*message*]	read, run	At read time and run time, terminates **omake** with the user-supplied exit *status* (1 if *status* is not supplied). If *message* is supplied, **omake** prints it on the error output (standard error) before terminating; otherwise, the termination is silent. When you use %**abort** while running **omake** in a VOB, no configuration records are written. Any output files created by the build script before it executes %**abort** will not be derived objects.

Table 14 Other Percent Directives (Part 2 of 6)

Directive	Appl. Time	Description
%**chdir** *directory*	run	Changes the current directory to *directory*. This is the directory in which each subsequent build script starts. For example: `copyit:` `    %chdir subdir` *(change into subdir)* `  omake $(MFLAGS)` *(do recursive Make)* `%chdir ..` *(change back to parent)* The **MAKEDIR** macro is not affected by **%chdir** and its value is always the directory **omake** started in.

%**do** [*target*] [*macro*[+]=*value* ...]	run	Executes the build scripts of the named *target*. The exit status of the **%do** is the exit status of the last build script of *target*. If nothing appears after the **%do,** this directive does nothing. If *target* does not exist, **omake** issues a warning unless **%do** is preceded by the @ shell-line prefix. During the **%do**, the attributes of *target* are the combination of the attributes of the current target and of *target*, with the current target having precedence.
		macro=value is a macro definition that is in effect during this **%do**. *macro*+=*value* appends *value* to the current value of *macro*. For either form of redefinition, the old value is restored after the **%do** is finished.
		Spaces in *target* or between the start of *macro* and the end of *value* must be enclosed in double quotes. Up to 10 macro definitions are allowed per **%do**, separating the definitions with spaces. For example, the directive
		`sub.obj : sub.c`
		`  %do COMPILE.c CFLAGS="-Od -Zi" OUT= -Fo$(.TARGET)`
		`COMPILE.c :`
		`%echo Compiling    ==== $(.SOURCE) =====`
		`cl $(CFLAGS) $(OUT) -c $(.SOURCE)`
		`  %echo Done        ==== $(.SOURCE) =====`
		produces the following build scripts when **sub.obj** is updated:
		`Compiling      ==== sub.c =====`
		`cl -Od -Zi -Fosub.obj -c sub.c`
		`Done           ==== sub.c =====`

Table 14 Other Percent Directives (Part 3 of 6)

Directive	Appl. Time	Description

		When you use **%do** while running **omake** in a VOB, the commands from the other build script are not included in the configuration record. For the following makefile, any changes to the rules to build **MakeCFile** do not cause derived objects built previously to be out of date even though the executed commands have changed:
		`.c.obj: $<`
		`   %do MakeCFile`
		
		`MakeCFile:`
		`   @echo Building an object from a .C source file`
		`   $(CC) /c $<`

%echo [**–n**] *message* [>[>] *file*]	read, run	Prints *message* either to standard output, or to *file* if redirected. The *message* is followed by a line feed unless **–n** is specified.
%error [*status*] [*message*]	read, run	At read time, **omake** terminates with the user-supplied exit *status* (1 if *status* is not supplied). At run time, this returns the exit *status* which is treated like any other build script exit status. If *message* is supplied, **omake** prints this message to the error output (standard error) before terminating or returning the exit status.
%exec *command_line*	read	Executes the *command_line* and sets the **status** macro to the exit status.

Table 14 Other Percent Directives (Part 4 of 6)

Directive	Appl. Time	Description	
%**include** [**<** *or* **"**]*name*[**>** *or* **"**]	read	Reads the contents of *name* into this makefile. **omake** looks in different places depending on whether the file is specified as *name*, "*name*", or <*name*>.	
		If *name* is an absolute pathname, it is only looked for there. Otherwise, "*name*" is looked for in the directory of the including file, then in the directory of **make.ini**. For Windows NT, **omake** looks in the directory of **omake.exe,** then in directories of the **INIT** environment variable, and finally in directories of the **INCLUDE** environment variable.	
		<*name*> is looked for in the same manner as is "*name*", except the directory of the including file is not used.	
		For example:	
		`%include c:\home\make.ini`	*(use the absolute path)*
		`%include <c:\home\make.ini>`	*(use the absolute path)*
		`%include make.sub`	*(use dir. of including file)*
		`%include "make.tpl"`	*(use the search scheme)*
%**restart** [*command_line*]	read, run	Restarts the make process. If *command_line* is not supplied , **omake** is restarted with the original command line. This calls **omake** recursively; it can be done at read time, and you don't have to keep track of the command line.	
		When you use %**restart** while running **omake** in a VOB, %**restart** writes a configuration record before restarting. Note that if a target with %**restart** in the build script is ever winked in, the %**restart** doesn't occur. Therefore, if your build script has a rule to rebuild the makefile and then has %**restart** to re-read the updated makefile and continue building, this won't have the desired effect if the makefile is winked in. **omake** winks in the makefile, but no restart occurs, and the old makefile is used for the rest of the build. In this case, use **–M** or **.MAKE_MAKEFILES** to build the makefile before reading it. In all other cases, mark any target with %**restart** in it as **.NOWINK_IN**.	

Table 14 Other Percent Directives (Part 5 of 6)

Directive	**Appl. Time**	**Description**

%set *name* [+]= *value*	read, run	Sets macro *name* to *value* at run time, although it can be used at read time. Use **+=** to append to the current definition. As an example, the build scripts of the following **set_debug_flags** target sets the **LDFLAGS** value to **/CO** and appends **–Od –Zi** to the **CFLAGS** value:
		`set_debug_flags :`
		`  %set LDFLAGS = /CO`
		`  %set CFLAGS += –Od –Zi`
		Any white space between **=** and *value* is ignored. Any white space between **+=** and *value* is condensed to a single space.

%setenv *name* [=] *value*	read, run	Sets environment variable **NAME** (converted to uppercase on Windows NT) to *value*. The variable is available to **omake** and to the build scripts executed by **omake**. If **NAME** is also a macro, the macro value is also updated to *value*. After **omake** terminates, the variable reverts to its previous value.
		The main use of this directive is in the **.BEFORE** special target to set up the environment for the makefile. For example:
		`.BEFORE .MAKE :`
		`  %setenv INIT=$(MAKEDIR);$(INIT)`
		We recommend that you not use this command with configuration records at run time. It can cause evaluation of environment variables to differ from run to run and prevents shopping from finding an appropriate derived object to wink in. For example, in one build, target A is built and changes the value of an EV; target B is then built and references the value of that EV. The next time the build is run, B may be built first and because A hasn't modified the environment variable yet, it runs differently than it did the first time.

Table 14 Other Percent Directives (Part 6 of 6)

Directive	Appl. Time	Description
%**undef** *name*	read, run	Undefines the macro named *name*. When you use %**undef** while running **omake** in a VOB, the configuration record is stored as if the %**undef** were not executed. The %**undef** runs when the build script is executed, but if the macro is evaluated after the %**undef**, the macro is expanded in the configuration record with the value of the macro when the target build began, even though the build script ran with the macro expanding to nothing after the %**undef**.

Compatibility with Other Make Utilities

For PM/CB compatibility, **omake** supports the **%exit** directive and **%status** functional operators. For Microsoft NMAKE compatibility, **omake** supports the **!CMDSWITCHES** and **!message** directives. See Appendix D, *Compatibility and Emulation*, for details.

Dot Directives

Dot directives (usually called directives, also) appear on a dependency line where the target name is a period followed by uppercase letters. Dot directives only work at read time.

At read time, dot directives modify the operation of **omake** from the point the directive is encountered in the makefile. Most command-line options have an equivalent dot directive.

The current state of each dot directive is kept by **omake** in a state macro with the same name as the directive. For example, the **.DEBUG** directive sets the debugging mode and the **$(.DEBUG)** is the value of the debugging mode. (The state macros are described in *Predefined Macros: State Macros* on page 63 and listed in condensed form in Table 22 on page 139.)

Directives that are effective on only some operating systems are noted.

Some directives accept lists of patterns as their dependents. Pattern lists may contain the pattern character %. When evaluating whether a name matches a pattern, the tail of the prefix of the name (subtracting directory names as appropriate) must match the part of the pattern before the %; the suffix of the name must match the part of the pattern after the %. For example,

/dir/subdir/x.o matches the patterns **%.o**, **x.o**, **subdir/%.o**, and **subdir/x.o**, but does not match **/dir/subdir/otherdir/x.o**.

The following **omake** directives accept pattern lists:

.DEPENDENCY_IGNORED_FOR_REUSE:
.DO_FOR_SIBLING:
.INCREMENTAL_REPOSITORY_SIBLING:
.INCREMENTAL_TARGET:
.SIBLING_IGNORED_FOR_REUSE:
.WINK_IN:

Table 15 lists **omake** directives, their equivalent command-line options (if applicable), and the actions that the directives perform.

Table 15 omake Directives (Part 1 of 10)

Directive	Flag	Action

.CASE_MACRO :		By default, **omake** treats macro names in a case-insensitive fashion. For this reason, the makefile lines
		`Case     = mixed`
		`CASE     = upper`
		define only one macro. That is, both **$(Case)** and **$(CASE)** evaluate to `upper`. The **.CASE_MACRO** directive makes macro names case-sensitive. Now the lines above define two macros, Case and CASE.
		The **.NOCASE_MACRO** directive makes macro names case-insensitive.

.CASE_TARGET :		By default, **omake** treats target names as case-insensitive, so **main.obj** and **MAIN.OBJ** are the same target. The **.CASE_TARGET** directive causes the case of target names to be considered. The **.NOCASE_TARGET** directive makes target names case-insensitive.

Table 15 omake Directives (Part 2 of 10)

Directive	Flag	Action
.DEBUG : *value*	**–#**	Selects debugging actions, as listed below: 0 Turn off all warnings 1 Display makefile lines as they are read 2 Warn about undefined macros when they get expanded 4 Warn about unrecognized lines in the makefile 8 Leave behind automatic response and batch files The values can be summed to combine message types. Also, each **.DEBUG** directive adds its value to the current value, which is initially zero. The **.NODEBUG** directive turns off listed *value*s. Here are some makefile examples: `.DEBUG : 4` *(warn about unknown lines)* `.DEBUG : 0` *(no warnings)* `.DEBUG : 6` *(warn about unknown lines and undef'd macros)* `.DEBUG : 1` *(and display makefile lines)* `.NODEBUG : 1` *(now don't display makefile lines)*
.DEBUG_GRAPHICS :		Run-time debugging uses line-drawing characters.
.DEBUG_PRINT :	**–p**	Selects the print debugging information mode.
.DEBUG_RUN :	**–d**	Selects the run-time debugging mode.

Table 15 omake Directives (Part 3 of 10)

Directive	Flag	Action
.DEPENDENCY_IGNORED_FOR_REUSE : *file …*		Ignores the *files* when **omake** determines whether a target object in a VOB can be reused (is up to date). By default, **omake** considers that a target cannot be reused if its dependencies have been modified or deleted since it was built. This directive applies only to reuse, not to winkin. Also, it applies only to detected dependencies, which are dependencies that do not appear in the makefile. You can specify the list of files with a tail-matching pattern; for example, **subdir/%.module**. Unlike the files listed in most directives, the *files* on this list refer to the names of dependencies, not to the names of targets. As such, the directive may apply to the dependencies of many targets at once. This directive is most useful when identifying a class of dependencies found in a particular toolset for which common behavior is desired across all targets that have that dependency.
.DO_FOR_SIBLING : *file …*		This directive is intended to be used in its negative form (**.NODO_FOR_SIBLING**). **.NODO_FOR_SIBLING** disables the creation of a derived object for any file listed if that file is created as a sibling derived object (an object created by the same build rule that created the target). These sibling derived objects are left as view-private files. You can specify the list of files with a tail-matching pattern, for example, **%.lock**. Unlike the files listed in most directives, the *files* on this list refer to the names of sibling objects, not to the names of targets. As such, the directive may apply to the siblings of many targets at once. This directive is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.

Table 15 omake Directives (Part 4 of 10)

Directive	Flag	Action
.ENV_OVERRIDE :	**–e**	Causes macro definitions from the environment to take precedence over macros defined in the makefile. The negation is **.NOENV_OVERRIDE**.
.ENVMACROS :		Causes macro definitions to be made of each environment variable. The negation is **.NOENVMACROS**.
.INCLUDE : *file ...*		The *files* are included at this point in the makefile. Each *file* is treated as if **%include** *file* happened at this point.
.INCREMENTAL_REPOSITORY_SIBLING : *file ...*		The sibling files listed are incremental repository files created as siblings of a primary target. They may contain incomplete configuration information, and should prevent **omake** from winking in the primary target. This is a very special-purpose directive, useful when a toolset creates an incremental sibling object, and the user wants more manual control over that object. You can specify the list of files with a tail-matching pattern, for example, **%.pdb**. Unlike the files listed in most directives, the *files* on this list refer to the names of sibling objects and not the names of targets. As such, the directive may apply to the siblings of many targets at once. This directive is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.
.INCREMENTAL_TARGET : *tgt ...*		Merges configuration record incrementally for the listed targets. In other words, this directive combines information from instances of this target generated previously with the current build of this target. This directive is most useful when building library archives, because typically only some of the objects going into a library are read each time the library is updated. You can specify the list of files with a tail-matching pattern, for example, **%.a**.

Table 15 omake Directives (Part 5 of 10)

Directive	Flag	Action
.KEEPDIR :	**–D**	Enables keep-directory mode. The first access of the current directory or any search directory (see *Search Directories* on page 106) to look for a file results in the directory being read into memory and kept. Subsequent accesses to the directory use the in-memory version and occur much quicker. The negation is **.NOKEEPDIR**.
.KEEPWORKING :	**–k**	Enables the keep-working mode. Any errors when updating a target cause work on that target to be stopped, but the make process continues. Because the target was incompletely made, any other targets that depend on it are prevented from being updated. This mode is handy for long, unattended builds because it maximizes the amount of making without ignoring the exit status as the **–i** command-option does. The negation is **.NOKEEPWORKING**.
.MACRO_CHAR : *char*		Selects *char* as the new macro character. It is best to put this directive in **make.ini**. For example, to change from the default **$** to **+** use the following: `.MACRO_CHAR : +` These characters cannot be the macro character: **( ) { } , : = # ' @ * < &**

Table 15 omake Directives (Part 6 of 10)

Directive	Flag	Action

.MAKE_MAKEFILE :	**–M**	Tells **omake** to make each makefile before trying to read it.
		In terms of implementation of this feature, **omake** reads **make.ini** and checks whether **.MAKE_MAKEFILE** is selected. If it is, **omake** makes the first makefile and reads it. It then makes the next makefile (if any) and reads it.
		The **make.ini** file must supply the rule for making the makefile. The **.MAKE_MAKEFILE** directive turns on makefile making. You can put this rule in **make.ini**:
		`.MAKE_MAKEFILE`
		`makefile:`
		`  imake -I \im\rules`
		Doing an **omake** *target* uses imake to create the makefile. You can also define different search directories for different extensions with the **.PATH.***ext* macro. See *Search Directories* on page 106 for more information.
		NOTE: Be aware that the makefile may contain information (such as the imake directory) needed to extract the makefile itself. You must provide some means of determining the imake directory from **make.ini** or you must give the imake directory to **omake** with a command-line macro.

.MS_NMAKE :	**–EN**	Turns on fullest Microsoft NMAKE emulation. A discussion of the extensive NMAKE emulation capability is in the section *Microsoft NMAKE Compatibility* on page 158. The **.MS_NMAKE**, .**omake**, and **.POLY_MAKE** are exclusive modes and only one of them is true. That is, only one of **$(.MS_NMAKE)**, **$(.omake)** or **$(.POLY_MAKE)** is 1 at any one instance.

Table 15 omake Directives (Part 7 of 10)

Directive	Flag	Action
.NOCMP_SCRIPT : *tgt* ...		Builds the specified targets as if the **–O** option were specified; build scripts are not compared during configuration lookup. This is useful when different makefiles (and, hence, different build scripts) are regularly used to build the same target. The list of targets may be specified with a tail-matching pattern, for example, **%.obj**.
.omake :	**–EO**	**omake** is set to its native emulation mode.
.POLY_MAKE :	**–EP**	Selects PM/CB emulation. A discussion of the extensive PM/CB emulation capability is in the section *PM/CB (Intersolv Configuration Builder and PolyMake)* on page 145.
.REGEX_CHAR : *char*		Selects *char* as the character that indicates special regular expression character sequences. It is best to put this directive in **make.ini**. For example, to change from \ (the default) to **~** use this directive: `.REGEX_CHAR : ~`
.REGEX_WILD : *char*		Selects *char* as the regular expression wildcard character that matches any single character. For example, to change from **.** (the default) to **?** use this directive: `.REGEX_WILD : ?`
.REJECT_RULES :	**–r**	Rejects all rules defined prior to this directive's appearance. Use this directive at the top of **make.ini** to reject **omake**'s predefined rules. Use it at the top of your makefile to reject all rules defined prior to the makefile. The **–r** command-line option is equivalent to a **.REJECT_RULES** directive at the end of **make.ini**.
.RESPONSE.*XXX* **:** [*response definition*]		Controls automatic response files (see *Response Files* on page 110).

Table 15 omake Directives (Part 8 of 10)

Directive	Flag	Action
.RULE_CHAR : *char*		Selects *char* as the new inference rule character. It is best to put this directive in **make.ini**. To change from the default of % to * use this directive: `.RULE_CHAR : *`

.SHELL : [**.AUTO** | **.NOMULTI** | **.NOREDIR**] ... [*program flags*]		Names both the shell *program* and its *flags* and specifies that the shell program be used to execute every build script. The **.NOSHELL** directive specifies that all build scripts are executed directly, without using the shell program. However, the **+** (use shell) and **:** (suppress shell) shell-line prefixes override any **.SHELL** and **.NOSHELL** directives.
		The **.AUTO** keyword tells **omake** to determine when to use the shell program. Without this keyword, the **.SHELL** directive specifies that the shell program is used for every shell line.
		The **.NOMULTI** keyword tells **omake** not to do special processing for multiple-command shell lines. By default, **omake** turns the multiple-command shell line into a batch file, which is executed.
		The **.NOREDIR** keyword tells **omake** not to handle redirection of I/O on the shell line. By default, **omake** handles redirection except |. With this keyword, only the shell program handles redirection. If **.NOREDIR** is used with **.AUTO**, any redirection on the shell line causes the command to be executed by the shell program.
		A **.SHELL** directive that is not given a *program* reverts to the remembered shell program:
		`.SHELL :`
		to execute every build script. The initial value of **.SHELL**:
		`.SHELL : $(COMSPEC) /C`
		To use the MKS shell, use the following syntax:
		`.SHELL : “$(ROOTDIR)\mksnt\sh”`
		or, alternatively for the MKS shell, you could use:
		`.SHELL : “$(ROOTDIR)\mksnt\sh -c”`

Table 15 omake Directives (Part 9 of 10)

Directive	Flag	Action
.SIBLING_IGNORED_FOR_REUSE : *file* ...		Ignores *files* when **omake** determines whether a target object in a VOB can be reused (is up to date). This is the default behavior, but this directive can be useful in conjunction with the **.SIBLINGS_AFFECT_REUSE** directive or **–t** command-line option. This directive applies only to reuse, not to winkin. You can specify the list of files with a tail-matching pattern, for example, **%.lock**. Unlike the files listed in most directives, the *files* on this list refer to the names of sibling objects, not to the names of targets. As such, the directive may apply to the siblings of many targets at once. This directive is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.
.SIBLINGS_AFFECT_REUSE :	**–t**	Examines sibling derived objects when determining whether a target object in a VOB can be reused (is up to date). By default, **omake** ignores modifications to objects created by the same build rule that created the target (sibling derived objects). This directive tells **omake** to consider a target out of date if its siblings have been modified or deleted.
.SUFFIXES : [*extension* ...]		Limits and orders the inference rules. **.SUFFIXES** is provided for compatibility with other make utilities and its use is discussed in the section *Compatibility with Suffix Rules (.SUFFIXES)* on page 37.

Table 15 omake Directives (Part 10 of 10)

Directive	Flag	Action
.UNIXPATHS :		Determines where **omake** looks for the inferred dependency when the current target name has a directory component. **omake** first tries matching the pathed target name against pathed inference rules. When the **.UNIXPATHS** directive is used, the second step is to match the full target name against unpathed inference rules, effectively causing **omake** to look for the inferred dependency in the same directory as the target. The default behavior, and the behavior when the **.NOUNIXPATHS** directive is used, is that the second step matches the *file name* of the target name against unpathed inference rules, causing **omake** to look for the inferred dependency in the current directory and in the search directories.
.WINK_IN : *tgt* ...	**–W**	This directive is intended to be used in its negative form (**.NOWINK_IN**). **.NOWINK_IN** specifies that the configuration lookup is restricted to the current view for the listed targets. The list of targets may be specified with a tail-matching pattern; for example, **%.obj**.

Compatibility with Other Make Utilities

omake supports many other directives for NMAKE, PM/CB, and Borland Make compatibility. See Appendix D, *Compatibility and Emulation* for details.

3.9 Target Attributes

Target attributes are properties given to targets. Attributes can be either positive or negative, for example, **.ATTRIBUTE** and **.NOATTRIBUTE**.

Using Attributes

Attributes are given to targets on dependency lines, but there are two forms:

```
target [ ... ] [ attribute ... ] : [ dependencies ... ]
attribute [ ... ] : [ target ... ]
```

Each form assigns the *attributes* to the indicated *targets*. The first form places the attributes after the targets and before the colon, and each target is given all attributes.

The second form has *attributes* only to the left of the colon and *targets* to the right. Each *target* is given all *attributes*. If no targets are listed, the attributes are given to all targets defined in the makefile after this line. This is very useful. To give all targets in a makefile an attribute, put a line of this form before any other dependency lines. This example that gives all targets defined after this line the **.PRECIOUS** attribute:

```
.PRECIOUS :                                  (near the top of the makefile)
```

Attributes and Inference Rules

Inference rules can have attributes, and the target being made with the inference rule inherits the additional attributes of the rule.

A target's attributes have a higher precedence than a rule's attributes. If a target and a rule specify an inconsistent attribute, the target's attribute is accepted.

List of Attributes

Table 16 lists attributes and their definitions.

Table 16 Attributes (Part 1 of 3)

Attribute	Definition
.ALWAYS	Always rebuilds this target, regardless of the results of configuration lookup or the time stamps of its dependencies. The **–a** command-line option is equivalent to an **.ALWAYS** attribute for each target in the makefile.
.CHAIN	Enables chaining of inference rules. (See *Multiple-Step Inference Rules* on page 33.) **.NOCHAIN** disables chaining, which may increase processing speed slightly.

.DEFAULT	When **omake** is run without specifying any targets on the command line, the first target (the default target) is built. The lack of user control of the default target makes it difficult to write generalized makefiles that can be included by other makefiles.
	The **.NODEFAULT** attribute indicates that this target is not the default target. Use **.NODEFAULT** for all non-rule targets in included, generalized makefiles. That way a makefile can include the generalized makefile without having the first target of the generalized makefile be the default target.
	In the generalized makefile you can put **.NODEFAULT** as the attribute of each target, or put
	`.NODEFAULT :`
	at the top of the generalized makefile and
	`.DEFAULT :`
	at the bottom.
	Note that targets defined in **make.ini** or in makefiles included from **make.ini** are never the default target.

Table 16 Attributes (Part 2 of 3)

Attribute	Definition
.IGNORE	When making a target, a build script that returns a nonzero status causes **omake** to terminate unless the target has the **.IGNORE** attribute. The **–i** command-line option is equivalent to a **.IGNORE** attribute for all targets. The status of individual build scripts can be ignored with the dash (**–**) shell-line prefix.
.INFER	**omake** uses inference rules to look for the inferred dependency for targets that have no build scripts. To omit, the inference rule check for targets without build scripts, give them the **.NOINFER** attribute. To force the inference rule check for targets with build scripts, give them the **.INFER** attribute.

.MAKE	Overrides the **–n** and **–q** command-line options. It is useful when executing **omake** recursively. For example:
	`nt .MAKE :`
	`  ( cd msdos.dir ; omake $(MFLAGS) )`
	If you execute **omake –n nt**, **omake** changes directory into **nt.dir** and executes **omake –n**. Without this attribute, the result is to display
	`( cd nt.dir ; omake $(MFLAGS) )`
	Similarly, **omake –q nt** changes directory and execute of **omake –q**.
	The **.MAKE** attribute differs from the **&** shell-line prefix, which overrides only the **–n** command-line option. The appearance of **$(MAKE)** on the build script also overrides only **–n**.

.NOCMP_NON_MF_DEPS : *tgt ...*	Builds the specified targets as if the **–G** option were specified; for each specified target, any dependency not explicitly declared in the makefile is not used in configuration lookup.

Table 16 Attributes (Part 3 of 3)

Attribute	Definition
.NOCONFIG_REC : *tgt* ...	Builds the specified targets as if the **–L** option were specified; modification time is used for build avoidance, and no CRs or derived objects are created.
.PRECIOUS	When a build script returns a nonzero status, **omake** checks whether the current target has been written. If it has, **omake** deletes the target, which prevents corrupted files from being used. This attribute prevents the deletion of the target itself and of chained targets.
.RULE	Is set when the percent sign appears on a dependency line. You can use the **.NORULE** attribute to allow a target with % in its name. On rare occasions, you may want to use **.RULE** to specify an inference rule that does not use %. Because % is a wildcard character, a rule without it matches exactly one target name.
.SILENT	A target's build scripts are displayed before being executed unless the target has the **.SILENT** attribute. The **–s** command-line option is equivalent to a **.SILENT** attribute for every target in the makefile. The @ shell-line prefix also prevents display of the build script.

3.10 Special Targets

omake uses some special targets at special times. The build scripts associated with these special targets are used at specific times as described in Table 17.

Table 17 Special Targets

Special Target	**Usage**
.AFTER : [*source* ...]	After **omake** builds its last target and immediately before it exits, any *sources* to **.AFTER** are built, and the build scripts of .AFTER are executed. To execute the build scripts even when using **omake –n**, give **.AFTER** the **.MAKE** attribute.
.BEFORE : [*source* ...]	Before **omake** builds its first target, any *source*s to **.BEFORE** are built, and then the build scripts of **.BEFORE** are executed. To execute the build scripts even when using **omake –n**, give **.BEFORE** the **.MAKE** attribute.
.DEFAULT[**.***ext*] :	If **omake** needs to update a target that has no build scripts, **omake** looks for an inference rule that matches the target name and uses the build scripts of the matched inference rule. Each **.DEFAULT**[**.***ext*] target is converted into an **%.***ext* : inference rule.

Compatibility with Other Make Utilities

For PM/CB compatibility, we support **.DEINIT**, **.EPILOG**, **.INIT**, and **.PROLOG**. See Appendix D, *Compatibility and Emulation*.

3.11 Search Directories

When **omake** looks for a file that has no path component in its name, **omake**'s default behavior is to search only the current directory. The search can be tailored to include other directories, a useful feature when your project is spread over multiple directories.

With **omake**'s search directory support you write dependency lines, such as

```
main.obj : main.c io.h
```

and have **omake** figure out where **main.c** and **io.h** are actually located.

Implied Location of Missing Files

omake uses the search directories to locate files. It is clear that if the file exists, the location of the file is the directory it was found in. What happens if the file is missing? In this case **omake** assumes that the missing file is located in the first directory of the appropriate search directory, or **.** (the current directory) if none is appropriate.

Search Directory Macros

Search directories are set up with macro definitions. **omake** supports **.PATH** macros and **VPATH** macros, but issues a warning if you use both.

The .PATH Macros

The **.PATH**[**.***ext*] macros define the directories **omake** uses to find files that don't have a path component. The optional *.ext* makes the **.PATH.***ext* macro extension-specific. That is, **.PATH.***ext* defines the search directories only for files with extension *.ext*. The **.PATH** macro (with no extension) controls the search for all files that aren't handled by a specific **.PATH.***ext*.

The value of the **.PATH** macros is a list of directory names separated by semicolons (;). Here are two examples:

```
.PATH      = .;..
.PATH.obj  = ..\obj
```

The first definition tells **omake** that all files can be found in **.** (the current directory) or **..** (the parent directory). The second definition tells **omake** that files with extension **.obj** can be found in directory **..\obj** (and in no other directory). Note that we have defined both a nonspecific **.PATH** and an extension-specific **.PATH.obj**. **omake** uses directories defined in **.PATH.obj** to search for files with the **.obj** extension, and uses **.PATH** for all other files.

Here is an example of a makefile that uses search directories:

```
OBJS     = main.obj sub.obj io.obj
.PATH.c  = ..

project.exe : $(OBJS)
  link $(.SOURCES), $(.TARGET), $(.TARGET,B);
```

The **.c** files are located only in the parent directory (**..**). Because no **.PATH** macro is defined, **omake** searches for all other files only in the current directory.

The VPATH Macros

The **VPATH** macros are similar to the **.PATH** macros with one major exception: the **VPATH** macros specify search directories in addition to the current directory. That is, the **VPATH** macros always have **.** as the first directory.

Search Directories and Run-Time Macros

Macros in **omake** usually serve as a simple text replacement. However, the run-time macros (**.TARGET**, **.SOURCE**, **.SOURCES**, and so on) include the location the target or source was found. For example,

```
.PATH.obj = objs

main.obj : main.c
  %echo $(.TARGET)
```

displays **objs\main.obj** because the **main.obj** target is located in the **objs** directory, and the value of the **.TARGET** macro is the pathname of the target.

Here is a **%.obj : %.c** rule for Borland C that uses two run-time macros:

```
%.obj : %.c
  $(CC) $(CFLAGS) –o$(.TARGET) –c $(.SOURCE)
```

The **–o** option names the output object file. **$(.TARGET)** is the name of the **.obj** file to be created, including its path component, according to where **omake** found the file. If the **.obj** file was not found, **.TARGET** is the implied location, either the first directory in **.PATH.obj**, if defined. Otherwise, it is the first directory in **.PATH**, if defined, or finally, in the current directory.

Search Directories and File Lookup

When **omake** searches for a file that has no directory component, it looks in the appropriate search directories. After the file has been located, **omake** assumes the file's location is permanent. Occasionally, this behavior is in error. For example, suppose a project's current **.c** and **.obj** files reside in a remote directory that **.PATH** references. You want to change a local copy of the project's **main.c** and then compile and link the resulting local **main.obj** with the remote **.obj** files, to produce a local **.exe** file.

Here is your makefile:

```
%.obj : %.c
  $(CC) $(CFLAGS) –c $(.SOURCE)

OBJS    = main.obj io.obj keyboard.obj
.PATH   = .;c:\remote

project.exe : $(OBJS)
  link $(.SOURCES), $(.TARGET);
```

You change the local copy of **main.c**. If there is a remote **main.obj** but no local **main.obj**, when you run **omake**, these commands are executed:

```
cl –c main.c
link c:\remote\main.obj c:\remote\io.obj c:\remote\keyboard.obj, project.exe;
```

The compilation uses the local **main.c** to produce a local **main.obj**, but the link uses **c:\remote\main.obj** rather than the local **main.obj**. This is because there was no **main.obj** in the current directory when **omake** started, and **omake** found **main.obj** in the **c:\remote** directory.

This example illustrates a general problem with **omake**. For reasons of speed, **omake** looks for a file (target) one time only. After that, **omake** assumes the location of the target is constant. As you have seen, this assumption can be wrong.

You can use the **.REREAD** target attribute to change this behavior. After executing the build scripts that update the target, **omake** searches again for any targets with the **.REREAD** attribute on disk. In the example, the make works correctly if the inference rule is given the **.REREAD** attribute:

```
%.obj .REREAD : %.c
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

The inference rule has the **.REREAD** attribute because **main.obj** inherits the build scripts and attributes of the matched inference rule.

Search Directories and Inference Rules

When **omake** tries to find an inferred dependency for an inference rule, it constructs a particular file name. If the file name has no path component, **omake** tries the file name in each search directory in order. Because each search directory is searched for each possible inferred dependency, **omake** runs more slowly with a large number of search directories. The **–D** command-line option (keep directory) speeds up this search.

Debugging the Search Directories

To debug your search directory choices, use the **–p** command-line option to print a section titled `Search directories`.

Compatibility with Other Make Utilities

For PM/CB compatibility, we support the **.SOURCE** directive, which is an additional way to specify search directories. The **.SOURCE** directive can specify the search directory for specific files, rather than for file extensions only.

For NMAKE compatibility, we support their search paths for dependents. See *Microsoft NMAKE Compatibility* on page 158.

3.12 Response Files

Many programs (for example, Microsoft **link** and **lib**, Gimpel **lint**, and **omake**) can receive their input from a response file. A response file is needed when the length of the build script exceeds the Windows NT command-line limit of 1024 bytes.

omake supports:

- Automatic response files and variables, where it generates a response file or places long command lines into an environment variable.
- Inline response files, where you place response file syntax around your build scripts.

The advantage of automatic responses (both files and variables) is that you don't worry about the command-line limit. When the build script exceeds the limit, a response file or variable is generated and the command is executed, using the response file or variable. You set up automatic responses once, for your linker, librarian, and so on. Thereafter, the build script for those programs can be arbitrarily long.

Inline response files are supported by a variety of other make utility vendors and **omake** supports the syntax they use.

Automatic Responses

A response class is a generalization that describes how and when to generate a response file or variable. When a response class has been defined, you tell **omake** which program names accept that class of response. We predefine most popular response classes, so you need only give **omake** the names of programs that accept these predefined responses.

To tell **omake** which programs accept response files or use environment variables, use the **.RESPONSE.***xxx* directive, where *xxx* is a name of a response class. For example, we predefine a LINK response class that describes the response file acceptable by Microsoft Link.

The **.RESPONSE.***xxx* directive is used to define or modify a response class and to add program names to an existing response class:

.RESPONSE.*xxx* `: [` *parameter* `... ] [` *program* `... ]`

The *parameters* describe the response class. Each *program* is either a base name (having neither path nor extension) or a pathname. Normally, you use a base name only, but for special circumstances you can use a pathname. If you use a pathname, you can have different response classes based on the literal name of the shell-line program, the pathname to the shell-line program, or the base name of the shell-line program.

Adding Program Names

To add program names to an existing response class, list them to the right of the directive. For example, adding LINK-style automatic response file support for SLR Systems **optlink** and Borland **tlink** is done as follows:

```
.RESPONSE.LINK : optlink tlink
```

The response classes are searched for program names in order from most recently defined to first defined, so your use of a program name overrides any predefined usage.

Response Class Parameters

Table 18 shows the parameters, the meaning of each parameter and its default value, and whether the parameter is used in response files and/or response variables.

Table 18 Response Class Parameters

Parameter	Meaning and Default	Used in Files?	Used in Variables?
pre=*ppp*	*ppp* is the prefix before the response file. If *ppp* contains white space it must be enclosed in double quotes.	yes	yes
suf=*uuu*	*uuu* is the suffix (extension) of the response file.	yes	
env=*eee* [1]	*eee* is the name of an environment variable.		yes
in=*num*	*num* is the shell-line length at which a response file or variable should be generated. The default is 1024 bytes.	yes	yes
sep=*s*	*s* is the character that separates the program's logical lines on the command line.	yes	
con=*c*	*c* is the character that connects physical lines in the response file into a single logical line.	yes	
out=*num*	*num* is the output line length of lines in the response file. The default value is 76.	yes	

[1] The **env** parameter determines whether a response variable or response file is used. If **env** is defined, an environment variable is used; otherwise, a response file is used.

Defining or Modifying a Response Class

To define a new response class or to modify an existing response class, use a **.RESPONSE.***XXX* directive with only parameters. One of **pre**, **suf,** or **env** is required when defining or modifying a response type.

If **env** is defined, the response uses an environment variable, otherwise a file is used.

Disabling a Response Class

A directive with neither parameters nor program names removes support for this response class. For example, you can turn off all LINK response files with this directive:

```
.RESPONSE.LINK :
```

Response File Example #1

This directive defines response file support for Microsoft **link**:

```
.RESPONSE.LINK : pre=@ suf=.rsp sep=, con=+ link
```

This declares the LINK response class has a prefix of @, a suffix of **.rsp**, logical lines on the command line are separated with commas, and physical lines in the response file are connected into one logical line with **+<ENTER>**. This directive also declares that the program named **link** accepts LINK-style response files. With this definition, the build script

```
link module1.obj module2.obj ... moduleN.obj, test.exe;
```

is executed as

link @*tempfile***.rsp**

The prefix @ tells **link** that what follows is the name of a response file. *tempfile* is a unique name of the form *tempdir*\MAKE*num*, where *num* is a 5-digit number, and the response file suffix **.rsp** appears as the extension of the name. The response file contents:

```
module1.obj module2.obj ... +
... +
... moduleN.obj
test.exe;
```

The first logical line **module1.obj** ... **module***N***.obj** is broken up in to several physical lines in the response file, each line ending with **+<ENTER>**. The next logical line **test.exe;** appears as the next physical line in the response file.

Response File Example #2

This directive adds response variable support for Microsoft CL:

```
.RESPONSE.CL : env=CL e:\c6\bin\cl.exe
```

The CL response class uses environment variable **CL**, and **e:\c6\bin\cl.exe** accepts this kind of response. Because a pathname is used, the command line

cl /DRELEASE="Release 1.0" /DOS=nt /Ic:\local\include ...

becomes the command line

cl

with environment variable CL having the value
/DRELEASE="Release 1.0" /DOS=nt /Ic:\local\include ..., but only if **cl** is found on the PATH as **e:\c6\bin\cl.exe**!

Using Automatic Responses

Write the build script as if the length of the command line were unlimited. When the build script gets too long, the automatic response is generated, a modified build script is executed, and the automatic response is removed.

Generation of Automatic Responses

To generate an automatic response, **omake** determines whether the program name given on the build script matches a **.RESPONSE.***xxx* program name (the response names). To see whether there is a match, **omake** does the following:

- ➤ If any response names have path components, it first compares the literal shell-line program name with all response names. If there are no matches, it looks on disk for the pathname that corresponds to the shell-line program name and compares the pathname to all response names.
- ➤ If no match has been found yet, it compares the base name of the program and with the response names.

The first match found determines the response class.

When the response class specifies a response file, the response file is generated if the build script is longer than the **in**=*num* parameter.

When the response class specifies an environment variable and the build script is longer than the **in**=*num* parameter, **omake** places the contents of the build script in the response environment variable, appending to the value of the environment variable if it exists.

Deletion of Automatic Responses

Automatic response files are deleted after the build script that generated the response file is executed, unless the **–#8** command-line option was used. Each automatic response variable is restored to its previous value after the build script that generated it executes.

Built-In Automatic Responses

omake has built-in automatic response file support for Gimpel **lint** and Microsoft **cl**, **lib** & **link**.

Built-in Responses for Windows NT

```
.RESPONSE.STD: pre=@ suf=.rsp omake link link32 lib lib32 cl cl386

.RESPONSE.LINT : suf=.lnt lint
```

Inline Response Files

In addition to automatic response files, **omake** also supports response files coded inline in the makefile. Here is the syntax for an inline response file:

```
target :
  command [ prolog ] << [ response_file ]
  [ line copied to response file verbatim ]
  .
  .
  .
  << [ epilog ]
```

The first **<<** introduces the response file; the last **<<** terminates it. *response_file* names the file. If the name is omitted, **omake** generates a unique name of the form *tempdir*\**MAKE***num***.rsp**, where *num* is a unique number. Everything between the pair of **<<** is placed in the response file and the command is invoked as

command prolog response_file epilog

The *prolog* and *epilog* are optional text. Usually, *prolog* is used to indicate that the following argument is a response file, and @ is the most common prolog.

The *epilog* can be used for redirection or other text. There are three special words that can appear in the epilog:

- **KEEP** specifies that the response file is not to be deleted
- **NOKEEP** pecifies that it is be deleted.
- **ECHO** specifies that the contents of the response file be displayed. **omake** also shows the contents of the response file when the **–n** command-line option is used.

Other build scripts can appear both before and after the inline response.

Deletion of Inline Response Files

Inline response files are deleted unless the **–#8** command-line option is used, or the KEEP keyword appears in the epilog. Response files named by **omake** are deleted after the build script is executed. User-named response files are deleted immediately before **omake** exits.

Inline Response File Example

Here is an example of an inline response file for Microsoft LINK:

```
program.exe : $(OBJS)
  link @<< $(MAKE_TMP)\link.rsp
  $(OBJS,W+\n)
  $(.TARGET)
  $(.TARGET,B,>.map)
  $(LIBS,W+\n)
  $(.TARGET,B,>.def);
  << KEEP ECHO
```

Here, **$(OBJS)** and **$(LIBS)** are assumed to be strings separated by white space. The **W** macro modifier replaces the white space with **+<ENTER>**, which is the appropriate line continuation for Microsoft **link**. If **OBJS** has the value **1.obj 2.obj**, and **LIBS** has the value **3.lib**, these build scripts evaluate to

```
link @$(MAKE_TMP)\link.rsp
```

where the contents of the response file are

```
1.obj+
2.obj
program.exe
program.map
3.lib
program.def;
```

The **KEEP** keyword has **omake** leave behind the response file. Otherwise, **omake** deletes it after the build script finishes. The **ECHO** keyword tells **omake** to display the contents of the response file after the **link @$(MAKE_TMP)\link.rsp** line is displayed. The default behavior is to display the contents only when doing **omake –n**.

Compatibility with Other Make Utilities

omake supports PM/CB local input scripts, NMAKE inline files and Borland Make's **&&** redirection operator. See Appendix D, *Compatibility and Emulation*, for details.

Debugging Makefiles

4

This chapter describes the process of debugging **omake** makefiles.

4.1 Command-Line Options

The following options are useful for debugging makefiles that do not work correctly:

- The **–#1** option displays each makefile line as it is parsed. Blank and comment lines are removed, and conditional directives are shown in normal, expanded, and evaluated forms.
- The **–p** option displays **omake**'s internal information. With this option, you can examine the following:
  - The values of all macros, to see where they were defined. Make sure they have the correct values.
  - All targets, their attributes, and their build scripts, to see where they were defined. This list also shows which target is the default target.
  - The inference rules, to see the rules known to **omake**.
- The **–d** option prints a run-time trace of **omake**. With this option, you can do the following:
  - Ensure that the correct initialization file and makefile are being read.
  - Watch **omake** use configuration lookup to compare the target to each dependency.

- ➢ Watch **omake** search for inferred dependencies by using inference rules. Ensure that **omake** attempts the inference rules you have defined. Cross-check with the **–p** flag.

For additional help with macros, the **.DEBUG : 2** directive (**–#2** option) causes **omake** to print a warning if a macro is used without being defined.

Finally, the **.DEBUG : 4** directive (**–#4** option) warns about makefile lines that **omake** does not know how to handle. This can catch misspellings of directives that you may have a hard time detecting. This directive is especially useful if you are trying to use a makefile from some other make utility.

4.2 Read-Time Debugging

To illustrate the process of debugging a makefile, the following command was executed:

```
omake –ndf demo –p –#1 CV=
```

This is the content of makefile **demo**:

```
# List of modules, and target name.
#
TARGET = project.exe
OBJS = a.obj b.obj

# If CV is defined, compile for debug
#
%if defined(CV)
 CFLAGS = –Od –Z7                                  # compile for debug
 LINKFLAGS = -debug -debugtype:both
%else
 CFLAGS = –Ox                                      # compile for size
 LINKFLAGS =                                       # no special link flags
%endif

# The default target in the makefile
#
$(TARGET) : $(OBJS)
```

```
# Additional dependency informationyou think they should
#
b.obj : $(TARGET,B).h

PATH.h = .;C:\SRC\H
```

Output Produced by –#1

For the purposes of this example, an empty **make.ini** file was created. Here is the output, with annotations in italics.

First, the initialization file **make.ini** is read:

```
*** Read make.ini ***                   (Start reading the make.ini file)
*** Done make.ini ***                   (Done reading the make.ini file)
```

Next, the makefile **demo** is read:

```
*** Read demo ***                       (Start reading the demo file)
 +                                      ("+" if more than one blank or comment line was
                                        skipped)


3: TARGET = project.exe                 ("num: " identifies current line number)
 4: OBJS = a.obj b.obj


+
 8: %if defined(CV)                     (Conditional directive is shown
 :---> true                              and evaluated)
 9: CFLAGS = –Od –Z7                    (Conditional was true, so this line and
 10: LINKFLAGS = -debug -debugtype:both  this line are processed)
 11: %else                              (End of first true conditional block)
 14: %endif                             (Lines are skipped until the "%endif")

+
 18: $(TARGET) : $(OBJS)                (The default makefile target)

+
 21: b.obj : $(TARGET,B).h              (An additional dependency)
 +
 23: .PATH.h = .;C:\SRC\H

*** Done demo ***                       (Done reading the "demo" file)
```

Output Produced by –p

Following output by the **–#1** option, **–p** prints out information internal to **omake**, including facts about macros, targets, search directories, and inference rules. The output begins with this line:

```
*** Begin print out ***
```
(Starts the -p printout)

The Macro Definitions

The first block of output is the macro definitions including the names and values of macros and the location they were defined. The location is any of the following:

built in	Defined by **omake,** but changeable by you
predefined	Defined by **omake**, but cannot be changed by you
command line	Defined by you on the **omake** command line
file:*number*	Defined by you in makefile *file* on line *number*

For brevity, most **omake** state macros have been omitted from this list:

```
.DEBUG_PRINT   =   1                                     # predefined     (state macro: the –p flag)
.omake         =   1                                     # predefined     (state macro: emulation)
BUILTINS       =   C:\Program Files\Rational\ClearCase\bin\  # predefined (name of the built-ins file)
                   make.ini
CC             =   cl                                    # built in       (name of the C compiler)
CFLAGS         =   –Od –Z7                               # demo:9         (C compiler flags)
CV             =                                         # command line   (command-line macro)
IMPLIB         =   lib                                   # built in
LINK           =   link                                  # built in       (name of linker)
LINKFLAGS      =   -debug -debugtype:both                # demo:10        (linker flags)
MAKE           =   C:\Program Files\Rational\ClearCase\bin\  # built in   (make location)
                   omake.EXE
MAKEARGS       =   -ndf demo -p -#1 CV=                  # predefined     (make command line)
MAKEDIR        =   C:\SRC                                # predefined     (make starting directory)
MAKEFILE       =   demo                                  # command line   (name of the makefile)
MAKEFLAGS      =   dnp -#1                               # predefined
```

```
MAKEMACROS     =   CV=“ “                             # predefined
MAKEVERSION    =   200                                # predefined      (make version)
MFLAGS         =   –dnp -#1                           # predefined      (command-line
                                                                         flags)
OBJS           =   a.obj b.obj                        # demo:4
OPUS           =   1                                  # predefined      (you know this is
                                                                         OPUS)
OS             =    NT                                # built in        (name of OS)
TARGET         =   project.exe                        # demo:3
```

The Search Directories

The search directories output lists both the extension-specific and nonspecific search directories. The extension-specific directories are listed first:

```
*** Search directories ***
for .h           : .\ C:\SRC\H\
all other files  : .\
```

A **.PATH** macro is not defined, so all other files are searched for only in the current directory.

The Automatic Response Definitions

Following the search directories are all automatic response file definitions. Each line of output of this section appears in exactly the form needed as if it were input to **omake**.

```
*** Automatic responses ***
.RESPONSE.WCC386: env=WCC386 pre=-u in=1024 wcc386
.RESPONSE.LINT: suf=.lnt out=76 in=1024 lint
.RESPONSE.STD: pre=@ suf=.rsp out=76 in=1024 wpp386 cl386 cl lib32 lib link32
```

The Inference Rules

Following the automatic response definitions is the list of inference rules:

```
*** Inference rules ***
* Suffix rules *
%.obj : %.c
 defined in: internal
    $(CC) $(CFLAGS) -c $(.SOURCE)

%.obj : %.cpp
 defined in: internal
    $(CPP) $(CPPFLAGS) –c $(.SOURCE)

%.obj : %.asm
 defined in: internal
    $(AS) $(AFLAGS) $(.SOURCE);

%.obj : %.for
 defined in: internal
    $(FC) $(.SOURCE) $(FFLAGS)

%.res : %.rc
 defined in: internal
    $(RC) $(RFLAGS) -r $(.SOURCE);

%.exe : %.obj
 defined in: internal
    %do %.exe

* Meta rules *
%.lib :
 defined in: internal
    %if  ! %null(.NEWSOURCES)
    %if %exists(${.TARGET})
    $(IMPLIB) -OUT:$(.TARGET) $(LIBFLAGS) $(.NEWSOURCES) $(.TARGET)
    %else
    $(IMPLIB) -OUT:$(.TARGET) $(LIBFLAGS) $(NEWSOURCES)
    %endif
    %endif

%.exe :
 defined in: internal
     $(LINK) -OUT:$(.TARGET) $(LINKFLAGS) $(.SOURCES) $(LINKLIBS)
```

The `*Suffix rules*` are rules that can be of the form *.fromExt***.***toExt*. The `*Meta rules*` are all other inference rules.

The Targets and Build Scripts

A list of the targets follows the inference rules. The default target is listed first.

```
*** Targets and commands ***
>>> default target <<<              (The first target in the makefile)
project.exe : a.obj b.obj           (project.exe depends on a.obj & b.obj)
b.obj : project.h                   (b.obj depends on project.h)
```

The Final –p Output

Finally, after all **–p** output has appeared, you see this message:

```
*** Done print out ***
```

A Errors and Warnings

When **omake** encounters a problem, it produces an error, warning, or report:

- Errors are the most severe, causing **omake** to display a message followed by the word `Stop`. **omake** does some deinitialization and quits.
- Warnings are less severe. A message followed by the text string `(`*warning*`)` is displayed and **omake** continues.
- Reports are the least severe. A message is displayed and **omake** continues.

The diagnostic message uses either of these formats:

```
OMAKE: message.
OMAKE: file (line number): message.
```

In the second, *number* indicates the line in makefile *file* that produced the diagnostic. **omake** displays the additional file and line information when possible.

A.1 Reducing Message Severity

When you use the **–i** command-line flag or dash (**-**) build-script prefix to ignore the exit status, some error messages are downgraded to report messages. For example, if a build script returns an exit status of `4`, **omake** displays the following error message and quits:

```
OMAKE: Shell line exit status 4. Stop.
```

However, if the build script is prefixed with the dash, the report message is

```
OMAKE: Shell line exit status 4 (ignored)
```

and **omake** continues.

A second common mode is **omake**'s keep-working mode, specified with the **–k** command-line flag or **.KEEPWORKING** directive. If **omake** is in this mode and encounters a problem when executing a target's build scripts, it stops updating the target immediately. In this mode some error messages are downgraded to report messages. For example, the previous error message is now this:

```
OMAKE: Shell line exit status 4 (keep working)
```

A.2 Error Messages and Explanations

In Table 19, the diagnostic messages that are followed with the text [(ignored) | (keep working)] are error messages that are downgraded to report messages if either the ignore exit status or keep working mode is in effect.

NOTE: All messages that start with `Test:` are errors in the conditional expression tester.

Table 19 Error Messages (Part 1 of 6)

Message	**Severity**	**Explanation**
Bad foreach '*line*'	error	The foreach *line* has an incorrect syntax. The correct syntax is **%foreach** *name* [*in*] *list_of_values*
Bad transformation macro	error	The PM/CB transformation macro had an incorrect specification.
break/continue without while/foreach	error	A **%break** or **%continue** has occurred without a preceding **%while** or **%foreach**.
Can't @include '*name*'	warning	File cannot be read with the @ macro modifier.
Can't create lock in '*directory*'	error	**omake** creates a lock file in the *directory* and bases the name of its response, swap, and batch files on the name of the lock file. The directory is either the value of the **MAKE_TMP** macro (if defined) or the current directory.

Table 19 Error Messages (Part 2 of 6)

Message	Severity	Explanation
Can't %do '*name*'	warning	The **%do** directive specified a nonexistent target *name*. Use @ as a shell-line prefix before the **%do** to inhibit this warning.
Can't find '*program*' on PATH [(ignored) | (keep working)]	error or report	**omake** cannot execute the current build script because the *program* was not found in any of the directories specified in the **PATH** environment variable.
Can't have : and :: dependencies for '*target*'	warning	The *target* was given on the left side of a regular (single colon) dependency and also on the left side of a double-colon dependency. These cannot be mixed.
Can't have both **.PATH.***ext* and **VPATH.***ext*	error	Both **.PATH.***ext* and **VPATH.***ext* macros were specified, but **omake** does not know which macro to use to determine the search directories.
Can't open *file*	report or error	The version control or library *file* can't be opened for reading or writing.
Can't open batch file '*file*'	error	When executing multiple commands (see *Using Multiple-Command Build-Script Lines* on page 73) the commands are written to a batch file and the batch file is then executed. The probable cause of this error is that the **MAKE_TMP** macro has not been set properly.
Can't open inline response file '*name*'	warning	The inline response file *name* can't be opened for writing.
Can't open response file '*name*'	warning	The automatic response file *name* can't be opened for writing.
Can't read makefile '*name*'	warning	The makefile *name* can't be read.
Can't read response file '*name*'	warning	The response file indicated on the command line by @*name* can't be read.
Can't redirect '*file*'	error	The redirection for *file* was specified incorrectly.
Can't undefine macro '*name*'	warning	The macro *name* can't be undefined because it is a predefined macro.

Table 19 Error Messages (Part 3 of 6)

Message	Severity	Explanation
COMSPEC not found	error	The **COMSPEC** environment variable, used to initialize the name and flags of the shell program, was not found in the environment.
Dependency-line wildcard '*spec*' is null	warning	The wildcard specification *spec* didn't expand to any file names.
Device error [(ignored) | (keep working)]	error or report	Execution of the build script caused a device error.
Directive failed [(ignored) | (keep working)]	error or report	The directive on this build script failed.
Directive nesting too deep. Max 31 levels.	error	Only 31 levels of *foreach*, *if*, and *while* nesting are supported, and your makefile has more than this.
%do macro missing '**=**'	error	A **%do** macro was missing the equal sign that is needed for its definition. The most probable cause is that there are spaces around it. If you need spaces, enclose them (or the macro definition) in double quotes.
Don't know how to make '*target*' [(keep working)]	error or report	**omake** has exhausted all means of making *target*. If **omake** is in keep-working mode, it marks this target as unusable and abandons work on it. Use the **–d** (debug) and **–p** (print info) options to determine the cause.
else without if	error	An **%else** directive encountered without a preceding **%if**.
elseif without if	error	An **%elseif** directive encountered without a preceding **%if**.
end without while/foreach	error	An **%end** directive encountered without a preceding **%while** or **%foreach**.
endif without if	error	An **%endif** directive encountered without a preceding **%if**.
Exec '*command*' failed	error	The **%exec** *command* directive cannot be executed.
Inline failed	warning	The current inline response file couldn't be generated correctly.

Table 19 Error Messages (Part 4 of 6)

Message	**Severity**	**Explanation**
Internal error: *number*	error	Report the internal error *number* to Customer Support.
Invalid executable '*program*' [(ignored) | (keep working)]	error or report	The specified *program* is not a valid executable.
Invalid **SHELL** '*program*'	warning	The specified shell *program* was not found.
Macro '*name*' has no value	warning	A run-time macro is used before its value was set.
Macro '*name*' not found	warning	The macro called *name* is undefined. This message appears only when the **.DEBUG** directive or **–#** command line flag has been set to **2**. When debugging your makefiles, this helps catch misspellings.
Maketmp failed	error	The routine that generates temporary file names failed, probably because the directory for the temporary files is full.
Max 8 response files for '*cmd*'	error	The multiple-command build script generated more than eight (the maximum) response files.
Missing *character* in '*name*'	error	**omake** was looking for *character* to mark the end of macro *name* but came to the end of the line instead.
No targets on dependency line	error	The dependency line being processed had no targets on the left side.
Not enough memory to exec '*name*' [(ignored) | (keep working)]	error or report	There is not enough memory to execute the specified program *name*.
Nothing to make	error	No target was specified on the command line, and no default target exists in the makefiles.
Option '*letter*' needs value	error	A required argument with the command-line option –*letter* was not specified.
Out of memory	error	**omake** ran out of memory. Your makefile may be too large, or **omake** may have installed a TSR.
Recursive macro '*name=value*'	warning	The macro called *name* has a recursive *value*; the value of the macro refers to itself. **omake** ignores the recursion.

Table 19 Error Messages (Part 5 of 6)

Message	Severity	Explanation
Regex: *message*	warning	The regular expression cannot be interpreted correctly; *message* describes the reason.
Removing *target*	report	The *target* is removed because it was modified when an error occurred in one of the build scripts that updated it. Removal prevents partially written files (such as object files) from being left behind when a compilation fails. Targets with the **.PRECIOUS** attribute are not removed.
Shell line exit status *number* [(ignored) | (keep working)]	error or report	The most recently executed build script had a nonzero exit status of *number*.
Shell line too long [(ignored) | (keep working)]	error or report	The build script exceeded the command-line limit. Use a response file. See *Response Files* on page 110.
Shell line without target	error	There is no target to which this build script belongs. Build scripts are indented from the left column by a tab or space character.
Test: bad expression "*exp*"	error	The expression *exp* cannot be parsed.
Test: bad first operand "*op*"	error	In a comparison, the first argument *op* cannot be parsed.
Test: bad operator "*op*"	error	The *op* is not one of the acceptable logical operators: **&&**, **||**, and **==**.
Test: bad second operand "*op*"	error	In a comparison, the second argument *op* cannot be parsed.
Test: no operand before "*text*"	error	There was no argument in a previous functional operator.
Test: unexpected "*text*"	error	The expression is finished logically, but there was more *text* on the line.
Test: unmatched quote "*text*"	error	Pairs of single or double quotes delimit strings that contain white space. A quote in *text* was not matched.
Test: unknown function "*name*"	error	An expression of the form *name***()** has an unrecognized *name*.
Test: unknown operator "*op*"	error	The operator *op* is not recognized.

Table 19 Error Messages (Part 6 of 6)

Message	**Severity**	**Explanation**
Test: '**)**' expected; got "*text*"	error	A right parenthesis is expected, but *text* was found instead.
Too many %do macros. Max of 10.	error	More than 10 macro definitions were specified for each **%do** directive.
Too many shell lines for '*target*'	warning	A *target* on a regular (single colon) dependency line can be given build scripts one time only. The additional build scripts are ignored.
Unexpected *what*	error	The *what* is either `end` or `endif`, and there is no previous **%foreach**, **%while**, or **%if** directive.
Unknown option '*letter*'		Only the command-line options listed by **omake –h** are acceptable.
Unknown status [(ignored) | (keep working)]	error or report	The build script returned an unknown status.
Unmade '*target*' due to source errors	warning	The **–k** flag was used to do the maximum useful work. One of *target*'s dependencies was not made because of an error, so the *target* cannot be made.
Unrecognized **.RESPONSE** keyword '*word*'	warning	The **.RESPONSE** directive does not accept the keyword *word*.
Unrecognized line '*line*'	warning	**omake** cannot parse this line.
Unterminated inline from line *number*	error	The inline response file was not terminated before the end of the makefile.
Unterminated *what* from line *number*	error	The *what* of `if`, `elif`, or `else` that started on line *number* was not balanced before the end of the makefile was reached.
User interrupt	error	The user typed **<CTRL-BREAK>** or **<CTRL-C>** while **omake** was executing a build script.

Exit Status Values

B

omake controls the execution of other programs, and the memory that **omake** itself uses is unavailable to these programs while **omake** is controlling them. When **omake** exits, it returns an exit status indicating the termination reason. The exit status can be tested by the command shell or, if you are doing a recursive make, by **omake** itself. The exit status can also be accessed in the **.AFTER** special target with the **MAKESTATUS** macro. For example:

```
.AFTER :
  %if $(MAKESTATUS) == 3
  %echo omake: The final build script exited with status: $(status)
  %endif
```

Table 20 lists exit values and their meanings.

Table 20 Exit Status Values (Part 1 of 2)

Exit Status	**Meaning**
0	Normal exit with no errors.
1	General purpose error if no other explicit error is known.
2	There was an error in the makefile.
3	A build script had a nonzero status.
4	**omake** ran out of memory.
5	The program specified on the build script was not executable.
6	The build script was longer than the command processor allowed.

Table 20 Exit Status Values (Part 2 of 2)

Exit Status	**Meaning**
7	The program specified on the build script cannot be found.
8	There was not enough memory to execute the build script.
9	The build script produced a device error.
10	The program specified on the build script became resident.
11	The build script produced an unknown error.
16	The user typed **<CTRL-C>** or **<CTRL-BREAK>**.

C Built-In Macros and Rules

This chapter describes the macros and rules that **omake** uses.

C.1 Macros

This section describes general macros, state macros, and built-in macros.

Predefined General Macros

Table 21 lists predefined general macros; these macros cannot be redefined.

Table 21 General Macros (Part 1 of 2)

Predefined Macro	Value
.NEWSOURCES	The list of target dependencies newer than the target; all dependencies when configuration lookup is enabled and the target is not marked as **.INCREMENTAL_TARGET**
.SOURCE	The inferred dependency or, if none, the first explicit dependency
.SOURCES	The complete list of dependencies for a target
.TARGET	The name of the target being made
.TARGETROOT	The root name of the target being made

Table 21 General Macros (Part 2 of 2)

Predefined Macro	**Value**
BUILTINS	The pathname of the built-ins file
CWD	The current working directory (the directory in which **omake** starts)
FIRSTTARGET	The first command-line target or the first makefile target
INPUTFILE	The current makefile being processed
MAKEARGS	All command-line arguments
MAKEDIR	The directory in which **omake** starts (same as CWD)
MAKEMACROS	All command-line macros
MAKESTATUS	The exit status with which **omake** exits
MAKETARGETS	All command-line targets
MAKEVERSION	The version of this **omake** executable
status	The exit status of the last build script executed

Predefined State Macros

These predefined macros return the state of **omake**'s command-line flags and directives. In Table 22, when **yes** appears in the **Directive** column, the value of the named macro is the state of the like-named directive. The **Flag** column shows the command-line flag that is the equivalent of the directive (if any).

NOTE: **.ALWAYS**, **.IGNORE**, and **.SILENT** are actually target attributes. They look like directives when they appear on the target side of a dependency line without any dependencies. Nevertheless, the **.ALWAYS**, **.IGNORE**, and **.SILENT** macros have the correct value, as if these attributes were directives.

Table 22 State Macros (Part 1 of 2)

State Macro	Directive	Flag	Macro Value
.ALWAYS	Yes	**–a**	0 or 1
.CASE_MACRO	Yes		0 or 1
.CASE_TARGET	Yes		0 or 1
.DEBUG	Yes	**–#**	The current debug options
.DEBUG_PRINT	Yes	**–p**	0 or 1
.DEBUG_RUN	Yes	**–d**	0 or 1
.ENV_OVERRIDE	Yes	**–e**	0 or 1
.ENVMACROS	Yes		0 or 1
.GLOBAL_PATH	Yes		0 or 1
.IGNORE	Yes	**–i**	0 or 1
.IGNORE_MFLAGS		**–z**	0 or 1
.KEEPDIR	Yes	**–D**	0 or 1
.KEEPWORKING	Yes	**–k**	0 or 1
.MAKE_MAKEFILE	Yes	**–M**	0 or 1
.MS_NMAKE	Yes [1]	**–EN**	0 or 1
.OMAKE	Yes [1]	**–EO**	0 or 1
.OPUS_52X	Yes	**–E2**	The list of compatibility features
.POLY_MAKE	Yes [1]	**–EP**	0 or 1
.REGEX_BACK			The regex literal backslash
.REGEX_CHAR	Yes		The regex escape character
.REGEX_DOT			The regex literal dot
.REGEX_WILD	Yes		The regex “match any character”
.REJECT_RULES	Yes	**–r**	0 or 1

Table 22 State Macros (Part 2 of 2)

State Macro	Directive	Flag	Macro Value
.RULE_CHAR	Yes		The rule character
.SHELL	Yes		The shell program and shell flags
.SILENT	Yes	**–s**	0 or 1
.SUFFIXES	Yes		The list of suffixes
.UNIXPATHS	Yes		0 or 1

1. Exactly one of **$(.MS_NMAKE)**, **$(.OMAKE)**, or **$(.POLY_MAKE)** is **1**.

Built-In Macros

Table 23 lists macros that are defined by **omake**, but can be changed.

Table 23 Built-In Macros

Built-In Macro	Definition	Default Value
AS	Assembler	**masm**
CC	C compiler	**cl**
FC	FORTRAN compiler	**f77l**
IMPLIB	Windows NT object librarian	**lib**
LINK	Object linker	**link**
MAKE	Pathname to command-line name	
MAKEFILE	First makefile read	
OS	The operating system	**NT**
RC	Resource compiler program	**rc**
SHELLCOMMANDS	List of internal shell commands	Not predefined

C.2 Macro Modifiers

When a macro is referenced, the value can be modified through the use of macro modifiers. To modify a macro, reference it as follows:

$(*name***,***modifier***[,***modifier* **. . .])**

name is expanded, and each *modifier* is applied in order to the elements of the value.

Table 24 lists macro modifiers and the actions they perform.

Table 24 Macro Modifiers (Part 1 of 2)

Modifier	Action
number	Selects the *number*th element of the value.
>*string*	Appends *string* to each element.
<*string*	Prepends *string* to each element.
*from***=***to*	Substitutes occurrences of *from* with *to*. If *from* does not appear in an element, the element is not changed.
F** or **	Each element is a wildcard spec evaluating to a list of files.
***D**	Each element is a wildcard spec evaluating to a list of directories.
@	Includes element as a text file.
@/*from***/***to***/**	Includes lines in text file, matching lines with regex *from* and replacing the matched part of the line with regex *to*.
A/	Converts element to an absolute file name using path separator **/**.
A	Converts element to an absolute file name using path separator \.
A	Converts element to an absolute file name using the default path separator \.
B	Selects the base part of the element.
D	Selects the directory part of the element.
E	Selects the extension part of the element.
F	Selects the file part of the element.

Table 24 Macro Modifiers (Part 2 of 2)

Modifier	**Action**
LC	Converts the element to lowercase.
M*regex*	Chooses elements that match regular expression *regex*.
M"*spec***"**	Chooses elements that match file specification *spec*.
N*regex*	Chooses elements that do not match regular expression *regex*.
N"*spec***"**	Chooses elements that do not match file specification *spec*.
P	Selects the path part of the element.
R	Selects the root part of the element
S/*from***/***to***/**	Substitutes *from* (a regular expression) to *to.* If *from* does not match an element, the element is not changed
UC	Converts the element to uppercase.
W*str*	Replaces white space between macro elements with *str*.
X	Replaces element names with pathnames using the search directories.
Z	Selects the drive part of the element.

C.3 Inference Rules

The following list of rules have been predefined in **omake** and are available unless you use the **–r** command-line flag or **.REJECT_RULES** directive.

Notice how the rules make use of macros. For example, the **%.obj : %.c** rule invokes the program **$(CC)** with the flags **$(CFLAGS)**. In your makefile, set **CFLAGS** to the flags to be passed to the compiler.

C source to object file, using the Microsoft C/C++ compiler:

Windows NT

```
CC   = cl
 %.obj : %.c
  $(CC) $(CFLAGS) –c $(.SOURCE)
```

C++ source to object file, using the Microsoft C/C++ compiler:

Windows NT

```
CPP  = cl
%.obj : %.cpp
  $(CPP) $(CPPFLAGS) -c $(.SOURCE)
```

Assembler source to object file, using Microsoft MASM:

```
AS   = masm
.%.obj : %.asm
  $(AS) $(AFLAGS) $(.SOURCE);
```

FORTRAN source to object file, using Lahey FORTRAN:

```
FC   = f77l
%.obj : %.for
  $(FC) $(.SOURCE) $(FFLAGS)
```

Resource compiler script to resource file, using the Microsoft Resource Compiler:

```
RC   = rc
%.res : %.rc
  $(RC) $(RCFLAGS) -r $(.SOURCE)
```

Update an executable, using Microsoft **link32** for Windows NT:

```
LINK = link
%.exe :
   $(LINK) -OUT:$(.TARGET) $(LINKFLAGS) $(.SOURCES) $(LINKLIBS)
```

Object file to executable, using the `%.exe` rule:

```
%.exe : %.obj
  %do %.exe
```

Update a library, using Microsoft **lib** for Windows NT:

```
IMPLIB = lib
%.lib :
  %if ! %null(.NEWSOURCES)
  % if %file(${.TARGET})
  $(IMPLIB) –OUT:$(.TARGET) $(LIBFLAGS) $(.TARGET) $(.NEWSOURCES)
  % else
  $(IMPLIB) –OUT:$(.TARGET) $(LIBFLAGS) $(.NEWSOURCES)
  % endif
  %endif
```

Compatibility with Other Make Utilities

The *ccase-home-dir***builtins.cb** and *ccase-home-dir***builtins.nm** files define the inference rules used by PM/CB and NMAKE, respectively. **omake** reads these files when PM/CB or NMAKE emulation is chosen; it searches for these files in the same manner in which it searches for the **make.ini** file (see *Locating the Initialization File* on page 46).

Compatibility and Emulation

D

omake achieves a great deal of compatibility with other make utilities, in that it understands the makefiles and features of other makes. When a feature or operation of **omake** differs from that of the other utilities, we provide emulation, so that **omake** operates like the other vendor's make utility.

D.1 PM/CB (Intersolv Configuration Builder and PolyMake)

omake is highly compatible with PolyMake up to v4.0PM/CB Compatibility and with Intersolv PVCS Configuration Builder, to v5.1. **omake** supports all PM/CB macros and transformation macros, library object modules, local input scripts, and most directives.

System Macros

omake supports the PolyMake/Configuration Builder system macros listed in Table 25:

Table 25 PM/CB System Macros (Part 1 of 2)

System Macro	Value	omake Macro
_Arguments	The command-line arguments.	**MAKEARGS**
_Cwd	The current working directory.	**MAKEDIR**
_Directory	The current working directory.	**MAKEDIR**

Table 25 PM/CB System Macros (Part 2 of 2)

System Macro	Value	omake Macro
_Exe	The pathname of the **omake** program.	**MAKE**
_FirstTarget	The first command-line target or, if none, the first makefile target.	**FIRSTTARGET**
_Flags	The command-line flags.	**MAKEFLAGS**
_FlagsMacros	The command-line flags and macros.	
_InputFile	The current makefile.	**INPUTFILE**
_Macros	The command-line macros.	**MAKEMACROS**
_PctStatus	The status of last operation line.	status
_Script	The default or first-named makefile.	**MAKEFILE**
_Source	The source for current target.	**.SOURCE**
_SourceRev	Version control version of current target.	**.VERSION**
_Sources	The sources for current target.	**.SOURCES**
_SysVer	Operating system version: *major.minor.*	
_TargRoot	The root part of the current target name.	**.TARGETROOT**
_Version	The PM/CB version number. **omake** reports v5.1.	
M_ARGS	See **_MakeArgs** above.	**MAKEARGS**
MAKEARGS	See **_MakeArgs** above.	**MAKEARGS**
MAKEVER	See **_Version** above.	
PCTSTATUS	See **_PctStatus** above.	status

Transformation Macros

omake supports PM/CB v5.x, both long and one-letter names. When an **omake** equivalent exists, it is listed in the right column of Table 26. Note that PM/CB macros work on text; **omake** modifiers work on a macro, which is expanded into text and modified. To use the **omake** equivalent, you need to define macro *name* with value *text*.

Table 26 PM/CB Transformation Macros (Part 1 of 2)

Trans. Macro	Long name	Result	omake Equiv.
$[@,*text***]**	Include	Include contents of file *text*.	**$(***name***,@)**
$[c,*str***,***begin***,***end***]**	Clip	Clip string *str* between *begin* and *end*.	
$[d,*text***]**	Directory	Directory part of *text*.	**$(***name***,D)**
$[e,*text***]**	Extension	Extension part of *text*.	**$(***name***,E,.=)**
$[f,*path***,***list***,***ext***]**	Filename	Build file name from *path*, *list* and *ext*.	
$[l,*text***]**	Lower	Convert *text* to lowercase.	**$(***name***,LC)**
$[m,*spec***,***text***]**	Match	Elements in *text* that match file *spec*.	**$(***name***,M"***spec***")**
$[n,*text***]**	Normalize	Normalize *text* as absolute file name.	**$(***name***,A)**
$[p,*text***]**	Path	Pathname of *text*.	**$(***name***,P)**
$[r,*text***]**	Base	Base name of *text*.	**$(***name***,B)**
$[s,*separator***,***text***]**	Separators	Replace *text* separators with *separator*.	**$(***name***,W***sep***)** [1]
$[t,*t1***,***t2***,***list***]**	Translate	Translate *list* mapping letters *t1* to *t2*.	
$[u,*text***]**	Upper	Convert *text* to uppercase.	**$(***name***,UC)**
$[v,*text***]**	Drive	Drive label of *text*.	
$[w,*text***]**	FileList	Wildcard expand file spec. *text*.	**$(***name***,*F)**

Table 26 PM/CB Transformation Macros (Part 2 of 2)

Trans. Macro	**Long name**	**Result**	**omake Equiv.**
$[x,*text***]**	DirList	Wildcard expand directory spec. *text*.	**$(***name***,*D)**
none	**$[Root,***text***]**	Root name of *text*.	**$(***name***,R)**

1. The *separator* can be enclosed in double quotes, which means that the *separator* must be parsed for special character sequences. The *sep* is never enclosed in double quotes and is always parsed for special characters.

Built-In Functions

In conditional expressions **omake** accepts:

%status

This is the exit status of last build script. It is the same as **$(status)** in **omake**.

Built-In Operations (Percent Directives)

%exit [*status*]

This directive terminates the make process with exit *status* (0 if *status* isn't given). Before terminating, the **.DEINIT** and **.EPILOG** special targets are run if they are defined. **.DEINIT** is run only if **.INIT** was run.

Directives

omake supports most PM/CB directives. For read-time interpretation, the directive starts in the first column of the makefile. For run-time interpretation, PM/CB emulation must be chosen and the directive must be indented.

Table 27 provides short descriptions of the supported directives.

Table 27 PM/CB Directives (Part 1 of 3)

Directive	**Applicable Time**	**Description**
.ExtraLine	read time	Causes an additional carriage return/linefeed to be output after each build script is executed. The negation is **.NoExtraLine**.
.Emulate [**Builder** | **NMAKE**]	read time	Sets the emulation mode to either Builder (PM/CB) or NMAKE.
.Ignore	read time	Causes nonzero build-script status to be ignored.
.Include [=] *file* ...	read time	Reads each *file*. If PM/CB emulation is chosen, **omake** looks for relative *file* names in the current directory. If PM/CB emulation is not chosen, **omake** treats **.INCLUDE** *file* the same as **%include**(*file*).
.KeepDir	read time run time	Same as the **omake .KEEPDIR** directive.
.KeepIntermediate	read time run time	Prevents intermediate files from being deleted.
.KeepWorking	read time run time	Same as the **omake .KEEPWORKING** directive.
.Keep_Lis	read time run time	Prevents local input scripts (inline response files) from being deleted. The negative of this directive is **.NoKeep_Lis**.
.KeepTemp	read time run time	Same as the **.Keep_Lis** directive.
.Lis_Comments	read time	Causes **#** in local input scripts to be considered literally, rather than as a comment character. You can also use **\#** to mean a literal **#**.
.Lis_File [=] [*filename*]	read time run time	Names the local input script file. Unlike PM/CB, **omake** allows a blank *filename* to reenable automatic generation of the local input script name.
.Logfile *files*	read time	Handled by **omake** as **.PVCS_STORAGE :** *files*

Table 27 PM/CB Directives (Part 2 of 3)

Directive	**Applicable Time**	**Description**
.Ms_Nmake	read time	Same as the **omake .MS_NMAKE** directive.
.NoEnvMacros	read time	Same as the **omake .NOENVMACROS** directive.
.Order	read time	Same as the PM/CB **.Suffixes :** directive.
.Path.*xxx* [**=**] *dir-1*[**;***dir-2*]...	read time	Treated as an **omake .PATH.***xxx* **=** *dir-1*[**;***dir-2*]... macro definition.
.PermitComments	read time	Same as **.Lis_Comments**.
.Poly_Make	read time run time	Same as the **omake .POLY_MAKE** directive.
.Precious	read time	Same as the PM/CB **.KeepIntermediate** directive.
.RejectInitFile	read time	Same as the **omake .REJECT_RULES** directive.
.Remake	read time	Targets are fully made each time they are encountered as sources.
.Shell [*shell_program*]	read time run time	Similar to the **omake .SHELL** directive, but without automatic detection of when to use the shell program.
.Silent	read time	Build scripts are not displayed on the screen before execution.
.Source [**=**] *dir_list file_list*	read time	Gives search directories to files. The *dir_list* is a semicolon-separated list of directories. The *file_list* is a space-separated list of filenames or file extensions. When **omake** looks for a file, it sees whether the file name or its extension is on any *file_list*; if it is, **omake** searches for the file on the *dir_list*.
.Source [*.ext*] **:** [*dir-1 dir-2 ...*]	read time	This PM/CB **.SOURCE** *dependency* is identical to a **.PATH** macro and is treated as an **.PATH.***xxx* = *dir-1*[**;***dir-2*]... macro definition.
.Suffixes [**:**]	read time	Without a colon, this directive acts like **omake**'s **.REJECT_RULES** directive. With a colon, it acts like the **omake .SUFFIXES** directive.

Table 27 PM/CB Directives (Part 3 of 3)

Directive	Applicable Time	Description
.VolatileTargs	read time	Same as the PM/CB **.Remake** directive.

Reserved Targets

omake supports the following PM/CB reserved targets:

.DEINIT [**:**]
If the **.INIT** special target was used, the build scripts of the **.DEINIT** target are executed immediately before the build scripts of **.EPILOG** are executed.

.EPILOG [**:**]
The same as the **omake .AFTER** special target. If emulating PM/CB, **omake** looks only for **.EPILOG**. Otherwise, it looks for **.AFTER**, and then **.EPILOG**.

.INIT [**:**]
This target's build scripts are executed immediately before any other build scripts.

.PROLOG [**:**]
The same as the **omake .BEFORE** special target. If emulating PM/CB, **omake** looks only for **.PROLOG**. Otherwise, it looks for **.BEFORE**, and then **.PROLOG**.

Local Input Scripts

omake accepts the PM/CB response file syntax:

```
target :
  command [ prolog ] <X<[ text ]
build script
.
.
.
< [ epilog ]
```

where *X* is a single character, usually @. If *text* is given, it is copied to the response file. Each build script is then copied to the response file. The command syntax is

command prolog Xtempfile epilog

where *tempfile* is a temporary file with a name of the form **make***num***.rsp**

Operation-Line Modifiers

omake supports all PM/CB operation-line modifiers, which are listed in Table 28.

Table 28 PM/CB Operation-Line Modifiers

PM/CB operation-line modifier	**omake shell-line prefix**
(Always)	**&**
(ExtraLine)	**>**
(Ignore)[*status*]	**-** [*status*]
(Iterate)	**!**
(NoShell)	**:** (* if emulating PM/CB)
(Shell)	**+**
(Silent)	**@**
(TrackErrors)	**~**

PM/CB Emulation

omake is highly compatible with PM/CB, but there are differences in how they read makefiles and in how they run. In PM/CB emulation mode,**omake** operates like PM/CB.

Emulation at Startup Time

If PM/CB emulation mode is selected at startup time, **omake** emulates the PM/CB command line and selection of the built-ins file. Selection of PM/CB emulation at startup time is done with the **–EP** flag, either on the command line or in the **OMAKEOPTS** environment variable.

To determine the startup emulation mode, **omake** examines the **OMAKEOPTS** environment variable for **–E***x* flags. It then examines the command-line for **–E***x* flags. If the last **–E***x* flag is **–EP**, **omake** starts up emulating PM/CB.

Emulation After Startup Time

The **.POLY_MAKE** directive turns on PM/CB emulation mode from the point it appears in the initialization file or any makefile.

The Command Line

First the **OMAKEOPTS** environment variable is parsed for options. Then, if **omake** is emulating PM/CB at startup, the **MAKEOPTS** and **BUILD** environment variables are parsed for options. Then the command line is parsed. Parsing entails the following:

- Handling the case-insensitivity of the PM/CB command line.
- Mapping options into **omake** equivalents. The help screen switches to the options available with our emulation of PolyMake v4.0 and Configuration Builder v5.x. The help screen's contents are stored in the file **omhelp.cb**.
- Handling the long-named Configuration Builder v5.x options.
- Warning about unconvertible command-line options.

Without emulation at startup, the command line is as documented in the section *Command-Line Options* on page 41.

The Emulation File (BUILTINS.CB)

If **omake** is emulating PM/CB at startup, **omake** reads its internal rules and macros. It then looks for **builtins.cb** first in the directory of **make.ini**, in the directory of **omake.exe**, and along directories of the **INIT** environment variable, in that order. If **builtins.cb** is found, it is read for macros and rules that give more complete PM/CB emulation.

The Initialization File (TOOLS.INI)

PM/CB distinguishes between an initialization file and a built-ins file. Both contain initialization information. The initialization file is almost always the file named **tools.ini**. If **omake** is emulating PM/CB at startup, **omake** searches for the initialization file and reads the first one it finds :

1. Named by the **–Init** command-line option
2. Named **tools.ini** in the current directory
3. Named **tools.ini** in a directory named by the **INIT** environment variable

If **omake** finds the initialization file, it reads information in the file starting with the section heading `[PVCS`.

The BUILTINS File

If **omake** is emulating PM/CB at startup, **omake** searches for the built-in file, and reads the first one it finds:

1. Named by the **–b** command-line flag
2. Named **builtins** in the current directory
3. Named **builtins.mak** in the current directory
4. Named by the **BUILTINS** environment variable

If **omake** is not emulating PM/CB at startup, it uses the method documented in the section *Locating the Initialization File* on page 46 to locate the initialization (built-ins) file.

The Makefile

If **omake** is emulating PM/CB at start up, **omake** looks for the default makefile in this order: **script.bld**, **script**, **makefile**, **makefile.mak**. The first file found is read. When trying to read a makefile file that doesn't have an extension, **omake** tries *file*, *file***.bld**, and *file***.mak**, in that order.

Makefile Contents

If **omake** is emulating PM/CB:

➤ The line continuation character sequence **\<ENTER>** is removed from the input.

- The ^ character is used for quoting and produces the following effects:

Character	**Effect**
^^	literal **^**
^<ENTER>	literal newline
^$	literal **$**
^	literal \\
^#	literal **#**

- The PM/CB-compatible directives listed in the previous section can be used at run time as well as read time.
- The **%end** directive is the same as **%endif**. Without emulation, **%end** terminates a **%foreach** or **%while** directive.
- **.INCLUDE** *file* searches only the current directory.
- Makefile macro names are case-sensitive.
- The **=+** macro (prepend) definition is supported.
- Environment variables override built-in macro definitions.
- The **MFLAGS** macro is the same as **_FlagsMacros**.
- The **TMP**, **TEMP**, and **WORK** environment variables are tried, in order, for the location of the directory where temporary files are created. Without emulation, **omake** uses the value of the **MAKE_TMP** macro.
- Duplicate entries in a target's dependencies are allowed. Without emulation, **omake** removes duplicate dependencies.

Operation Lines (Build Scripts)

If **omake** is emulating PM/CB:

- The shell program is used for executing every build script. The * and (NoShell) prefixes suppress execution of the shell program for this build script.
- The build scripts **cd dir** and **chdir dir** both change to directory **dir** and stay there until changed back explicitly. Otherwise, **omake** starts each build script from the directory that **omake** started in the **$(MAKEDIR)** directory.

- The **:** modifier means swap out of memory and * means suppress shell. Without emulation, * means swap out and **:** means suppress shell.
- The **-** shell-line prefix does not print the `Error code ... (ignored)` message.

Unsupported PM/CB Features

When running PM/CB makefiles with **omake**, be aware of the lack of support for some PM/CB features. The list here provides a workaround, when it is available.

Unimplemented Directives and Reserved Targets

.ArcFile	Tells PM/CB about ARC compression files.
.ExamineCmt (also known as **.Examine_Cmt**)	Checks comments for line continuation. Although this directive is unsupported, **omake** handles line continuation inside comments.
.Error[.*xxx*]	Reserved target; supplies auxiliary instructions for building targets when a build script returns a nonzero exit status. Use conditional directives instead.
.FootPrint	Directive; controls foot printing targets with an internal comment record.
.Ms_Make	Selects Microsoft MAKE emulation.
.NoEnvInherit	Prevents the environment from being passed to child processes.
.Rebuild	Directive; rebuilds previous versions of applications.
.ZipFile	Directive; tells PM/CB about ZIP compression files.

Iteration Groups

An iteration group does an implicit iteration over the **$?** macro. For example:

```
test.lib : test.obj chart.obj input.obj
  {
  lib contract.lib –add $? omake m2 noask
  }
```

The **%foreach** directive can be used in its place:

```
test.lib : test.obj chart.obj input.obj
%foreach file in $?
  lib contract.lib –add $(file) omake m2 noask
%endfor
```

Suffix Dependencies

Suffix dependencies allow the specification of a set of suffixes to be tried when building a particular target. This is not supported by **omake**.

Command-Line Flags

The following PM/CB command-line flags are not supported: **–Batch**, **–C**, **–Compile**, **–NoEnvInherit**, or **–Rebuild**.

Makefile Contents

The ~ suffix on archive extensions to handle like-named files in different directories is not supported. Instead, use a pattern-matching rule , such as this:

```
%.c : c:/apps/archives/%.c
  get –q –w $(_SourceRev) $(_Source)($(_Target))
```

The **_DefaultSuffixes** system macro is not supported.

Shared Definitions

omake does not parse the PVCS Version Manager configuration file automatically.

Operation-Line Modifiers

For PM/CB, the (Always) operation-line modifier (Shell-Line Modifiers) overrides the **–Touch** flag. This is not supported in **omake**. You can use the **.MAKE** attribute instead. For example:

```
# PM/CB
recursive :
  (Always)$(MAKE) $(MAKEFLAGS)

# Omake
recursive .MAKE :
  $(MAKE) $(MAKEFLAGS)
```

Built-In Operations

The built-in operations that handle foot printing are not supported. These operations are **%EAStamp**, **%ExeStamp**, and **%ObjStamp**.

D.2 Microsoft NMAKE Compatibility

omake is highly compatible with Microsoft NMAKE up to version 1.3 (the version supplied with Visual C++). **omake** supports all NMAKE directives, macros, paths, and rules.

NMAKE Directives

omake supports the following NMAKE directives:

!CMDSWITCHES {**+** | **–** } *opt*

This read-time directive turns on (**+**) or off (**–**) one or more options, *opt*.

!message *message*

This read-time directive is the same as the **%echo** *message* directive.

NMAKE Emulation

omake is highly compatible with NMAKE, but there are differences in how they read makefiles and in how they run. **omake**'s NMAKE emulation mode causes **omake** to operate like NMAKE.

Emulation at Startup Time

If NMAKE emulation mode is selected at startup time, **omake** emulates the NMAKE command line and selection of the initialization file. Selection of NMAKE emulation at startup time is done with the **–EN** flag, either on the command line or in the OMAKEOPTS environment variable.

To determine the startup emulation mode, **omake** first examines the OMAKEOPTS environment variable for **–E***x* flags, and then the command-line for **–E***x* flags. If the last **–E***x* flag is **–EN**, **omake** starts up emulating NMAKE.

NOTE: If you want submakes (recursive invocations of **omake**) to inherit NMAKE emulation mode, you must specify the **–EN** flag in the OMAKEOPTS environment variable.

Emulation After Start-Up Time

The **.MS_NMAKE** directive turns on NMAKE emulation mode from the point it appears in the initialization file or any makefile.

The Command Line

The **OMAKEOPTS** environment variable is parsed for options, and then the command line is parsed. Parsing entails the following:

- ➤ Handling the case-insensitivity of the NMAKE command line.
- ➤ Handling all NMAKE command-line options.
- ➤ Switching the help screen to the options available with NMAKE v1.3. The help screen's contents are stored in the file **omhelp.nm**.

Without emulation at startup, the command line is as documented in the section *Command-Line Options* on page 41.

The Emulation File (BUILTINS.NM)

If **omake** is emulating NMAKE at startup, **omake** reads its internal rules and macros, and then looks for **builtins.nm** in the directory of **make.ini**, in the directory of the **omake.exe** file, and along directories of the **INIT** environment variable, in that order. If **builtins.nm** is found, it is read for macros and rules that give more complete NMAKE emulation.

The Initialization File (TOOLS.INI)

If **omake** is emulating NMAKE at startup, **omake** reads the **tools.ini** file found in one of these directories:

- ➤ In the current directory
- ➤ In a directory named by the **INIT** environment variable

If **omake** finds the initialization file, it reads information in the file starting with the section heading `[NMAKE`.

Makefile Contents

If **omake** is emulating NMAKE:

➤ The **^** character is used for quoting and produces the following effects:

Character	**Effect**
^^	literal ^
^<ENTER>	literal <ENTER>
^$	literal $
^\	literal \
^#	literal #

➤ The **!else if**, **!else ifdef**, and **!else ifndef** directives are supported.

➤ Macro definitions of the form

```
VAR = ... $(VAR) ...
```

are supported and cause **$(VAR)** to be expanded before **VAR** is redefined. Other macro references in the macro value are not expanded.

➤ Macro definitions of the form

`ENVVAR =` *value*

where **ENVVAR** is an environment variable, assign to the **ENVVAR** macro the *value,* which is also exported to the environment. If the **–E** command-line flag is used, makefile macros cannot override environment macros, and this macro redefinition is ignored.

➤ Search paths for dependents are supported. These look like this:

```
forward.exe : {\src\alpha;d:\proj}pass.obj ...
```

➤ Inference rule search paths are supported. These look like this:

{*fromdir*}.*fromext*{*todir*}.*toext* **:** *(build script for inference rule)*

➤ Inference rules for targets that have a directory component look for the inferred source in the directory of the target.

➤ The **!include** *file* directive is supported. If *file* is an absolute pathname, it is used; otherwise, **omake** looks for *file* in the current directory, and then in the directory of the including file. If *file* appears inside angle brackets (for example, <*file*>), **omake** then looks along the directories in the **INCLUDE** environment variable.

Macros

If **omake** is emulating NMAKE, the following macro features are supported:

- Command-line macros are exported to the environment, and become available to recursive makes.
- Environment variables override built-in macro definitions (see the section *Macros* on page 52).
- The **MAKEFLAGS** macro is exported to the environment.
- Non-run-time macros are expanded at parse time.

  Most other make programs expand macros when a build script is run, but NMAKE expands macros when it parses the makefile. When the rule is parsed, it is defined with the expanded value of the macro, rather than with a reference to the macro. (The reference would be expanded at run time).

  EXCEPTION: NMAKE evaluates run-time macros at run time, because the values to which they are set are based on the target.

  Therefore, the following makefile is evaluated differently by NMAKE and **clearmake**:

  ```
  FOO=A
  rule:
     @echo $(FOO)
  FOO=B
  ```

  In NMAKE, `$(FOO)` is evaluated at parse time, so building `rule` echoes `A`. In **clearmake**, `$(FOO)` is evaluated at run-time and its value after parsing is `B`, so building `rule` echoes `B`.

Build Scripts

If **omake** is emulating NMAKE, the following build-script features are supported:

- The shell program is used for executing every build script.
- The build scripts **cd** *dir* and **chdir** *dir* both change to directory *dir* and stay there until changed back explicitly.
- A **set** *var*=*val* build script is treated specially and sets the environment variable **VAR** to the value *val*. That is, this build script is treated exactly like the **omake** directive **%setenv** *var*=*val*.
- The *filename-parts* syntax is supported. You can use this to get at the components of the name of the first source file. The complete file name is represented with the **%s** syntax. Parts of the file name are represented with this syntax: **%|[***parts***]F**, where *parts* is zero or more of

the following letters: **none**, the complete name; **d**, drive; **p**, path; **f**, file base name; **e**, file extension.

- Shell-line prefixes are limited to **@**, **-**, and **!**. The **^** prefix stops processing of prefixes.
- The **#** character is not treated as a comment in build scripts.

Inline Response Files

omake supports the NMAKE response file syntax, except that multiple inline response files on a single build script are not support.

NMAKE and **omake** allow the keywords **KEEP** or **NOKEEP** in the *epilog* of the inline response file. **KEEP** causes the response file to be kept (that is, not deleted). The default is **NOKEEP**.

Unsupported NMAKE Features

Multiple inline response files on a single build script are not supported.

Arithmetic operators are not supported in preprocessing expressions.

D.3 Opus Make v5.2x Compatibility and Emulation

omake is not entirely backward compatible with Opus Make 5.2x.

The **–E2** command-line flag makes **omake** emulate Opus Make 5.2x.

omake has the following compatibility and emulation features:

comment	v5.2x treated line continuation before comment detection, so a comment character (**#**) in a continued line causes the rest of the continued line to be ignored. **omake** treats **#** as a comment only until the end of the current physical line.
do	In v5.2x, macro definitions on the **%do** line were separated by a comma (**,**). In **omake**, they are separated by white space and must be enclosed in double quotes if they contain white space.

infer	v5.2x searched for the inferred source unless explicitly told not to with the **.NOINFER** attribute. **omake** searches for the inferred source only for targets without build scripts but the **.INFER** attribute can force it to search.
noiterate	The shell-line prefix **!** means iterate this build script. Note that **!** can also indicate a directive; the rule is that it indicates a directive if it can; otherwise, it is a prefix.

! echo $(.SOURCES)	Does not iterate: **!echo** is a directive
!| echo $(.SOURCES)	Iterates: | indicates the end of prefixes
! add $? to $@	Iterates: **!add** isn't a directive
@! echo $(.SOURCES)	Does not iterate: **!echo** is a directive
!@ echo $(.SOURCES)	Iterates: **!@** is not a directive

The **noiterate** feature turns off any interpretation of **!** as a shell-line prefix.

twopass	Opus Make v5.2x used two passes to macro expand build scripts. The first pass expanded all macros. The second pass tokenized the line and replaced any found targets with the pathname to the target. **omake** does not do the second pass unless you choose this feature. **omake** has a macro modifier, **X**, that allows selective expansion of names into pathnames.

D.4 Borland Make Compatibility

The following sections describe **omake**'s compatibility with Borland Make.

- Automatic dependencies

  omake knows how to read the dependency information stored in object files created from the Borland **bcc** compiler.

- Special targets

  .SWAP : **.SWAP** should only be found in makefiles produced by the Borland **prj2mak** program.

- Inline response files

  omake accepts the inline response file syntax:

```
target :
  command [ prolog ]&&X
  build script
  .
  .
  .
  X [ epilog ]
```

where *prolog* is usually @ and *X* is a single character (Borland uses **!** in its examples). The build scripts are placed in a response file and the command is invoked with the following command, where *tempfile* is a unique filename:

command prolog tempfile epilog

- Borland Make emulation

  There is no specific Borland Make emulation mode.

D.5 UNIX Make Compatibility

omake supports makefiles produced for SunOS (Solaris) Make. The following features are supported:

- makefile syntax

  The makefile expression

  `include` *filename*

  where `include` is in the leftmost column of the makefile includes *filename* at this point in the makefile. The makefile expression

  `$(`*name*`:`*str1*=*str2*`)`

  is a macro modification with string substitution, with strings *str1* and *str2* both capable of using the % character as a wildcard to match zero or more characters in the expansion of *name*.

- Inference rules

  omake supports the suffix rules and metarules of UNIX Make. Suffix rules are converted into **omake** inference rules. The **.SUFFIXES** directive is also supported.

- UNIX make emulation

  There is no specific UNIX make emulation mode.

Regular Expressions

E

Regular expression matching is a scheme for string searching. A regular expression is a string of characters, some normal, some special, that allows the specification of a pattern to be matched.

E.1 Configuration of Regular Expressions

The following directives control the special characters that appear in regular expressions:

Directive	**Use**	**Default Value**
.REGEX_CHAR	Sets the escape character used to indicate special sequences	\ (backslash)
.REGEX_WILD	Sets the wildcard character that matches any single character	**.** (period)

Regular Expression Components

Table 29 lists the forms, or components, of a regular expression.

Table 29 Regular Expression Components

Label	**Form**	**Form Description**
[1]	*character*	A normal character matches itself. Special characters: *wild* **\ [ ] * + ^** $
[2]	*wild*	The wildcard character, *wild*, matches any character.
[3]	\	The escape character makes special characters literal: **\wild** is literal **wild**, \\ is literal \, \[is literal [, and so on. The only exceptions are that **\(** and **\)** are special. See [7] below.
[4]	**[** *set* **]**	Matches one character in set. If the first character in set is **^**, this form matches characters not in set. A range *start–end* means characters from start to end. The characters **]** and **–** aren't special if they appear as the first characters in set. **\t** matches a tab character. For example:

Set	**Matches**
[a-zA-Z@]	Lowercase and uppercase alphabetic, or @
[^]–]	Neither **]** nor **–**
[^A–Z]	Not uppercase alphabetic
[<SPACE> \t]	A space or tab (that is, white space)

Label	**Form**	**Form Description**
[5]	*form**	Any regular expression form labeled [1] to [4] followed by the closure character * matches zero or more of the form.
[6]	*form***+**	**+** is like *, except it matches one or more of the form.
[7]	**\(** *form* **\)**	A regular expression in the form [1] to [10], enclosed as **\(** *form* **\)** matches what *form* matches. The substring matched by *form* can be referenced with a tag (see below).
[8]	**\1** ... **\9**	Matches a previously tagged regular expression (see [7]).
[9]	*form1form2*	A composite regular expression *form1form2*, where *form1* and *form2* are in the form [1] to [9], matches the longest match of *form1* followed by a match of *form2*.
[10]	**^** and **$**	A regex starting with **^** and/or ending with **$** restricts the regex to the beginning of the line and/or the end of line. Elsewhere in the regex, **^** and **$** are ordinary characters.

Referencing the Matched Expression

Once a regular expression has matched, you can reference the matched part:

- **&** refers to the entire matched string (but **\&** is a literal ampersand).
- If **\(\)** is used to delimit a part of the regular expression, the tag **\1** refers to the first delimited part of the matched substring. Successive pairs of **\(\)** are tagged **\2**, **\3**, ..., **\9**.

E.2 Macro Modifiers in OMAKE

The **M**, **N**, and **S** modifiers use regular expressions to match macro elements. The regular expression is matched against each macro element individually. For the following examples, assume the following macro definitions:

```
SRCS                = main.c sub.cpp io.cpp
CFLAGS              = –AX –Ifoo –Ibar /Ibaz –DX=–IT xI.c yi.c
```

Regular Expressions for the M Modifier

To select files whose names include **.c**:

```
$(SRCS,M.c) is       main.c sub.cpp
```

To select files that end in **.c**, anchor the search to the end with the regular expression character **$**. To get **$** to the regular expression, use **$$** in the makefile:

```
$(SRCS,M.c$$) is     main.c
```

The **^** regular expression character anchors the search to the front of the macro element:

```
$(CFLAGS,M–I) is       –Ifoo –Ibar –DX=–IT
$(CFLAGS,M^–I) is      –Ifoo –Ibar
```

The **[***set***]** regular expression characters indicate a *set* of characters, where *set* can be single characters (**[aA@]** matches **a**, **A**, or @), a range of characters (**[a-z]** matches **a** through **z**), or a mixture. For example:

```
$(CFLAGS,M^[-/]I) is   -Ifoo -Ibar /Ibaz
```

Regular Expressions for the S Modifier

One powerful feature of regular expressions is that when they are used in substitutions, they can access the matched parts of the string. The **S/***rfrom***/***rto***/** (substitution) modifier uses regular expression *rfrom* to substitute the *matched* part of an element with the *rto* regular expression. For example, when `DIR = NT_L`, the expression `$(DIR,S/\(wild*\)_wild*/\1/)` is `NT`

The **\(\)** pair surround part of the regular expression that can be referenced later. Inside the pair is *wild**, which matches any character repeated zero or more times. Taken together, they mean instruct **omake** to match any character, zero or more times, and tag it. The rest of the regular expression is _, which matches _ and *wild**, which matches any character repeated zero or more times.

The substitution replaces the matched part of the element with the expression **\1**, which is the stuff matched in the first pair of **\(\)**. The entire element was matched, so the substitution produces `NT`. Table 30 shows other expressions.

Table 30 Examples of Regular Expressions

Expression	**Result**
$(DIR,S/*wild****_\(***wild****\)/\1/)**	`L`
$(DIR,S/*wild****_\(***wild****\)/&/)**	`NT_L`
$(DIR,S/\(*wild****\)_ /\1/)**	`NTL`
$(DIR,S/_ //)	`NTL`
$(DIR,S/does not match/xyzzy/)	`NT_L`

Glossary

ATTRIBUTES. Properties that can be associated with targets. They include **.MAKE**, **.REREAD**, **.SILENT** and others.

BASE NAME. A file name without any directory components or extension.

BUILD. To bring a target up to date (make it current).

BUILD AVOIDANCE. The ability to fulfill a build request by using an existing derived object, instead of creating a new derived object by executing a build script.

BUILD SCRIPT. A command executed by **omake** or passed to the shell program for execution. Build scripts are used to update targets.

COMSPEC. This environment variable names the program that is the operating system command processor or shell program. For Windows NT, this is usually **cmd.exe**.

CONFIGURATION LOOKUP. The process by which **omake** determines whether to produce a derived object by performing a target rebuild (executing a build script) or by reusing an existing instance of the derived object. This involves comparing the configuration records of existing derived objects with the build configuration of the current view: its set of source versions, the current build script that would be executed, and the current build options.

DEFAULT TARGET. The first normal target defined in the first makefile. A normal target is not a directive, special target, attribute, or inference rule. It is the target that is made when **omake** is run without any command-line targets.

DEPENDENCY. Anything a target depends on. Dependencies are themselves targets because they can be made.

DEPENDENCY LINE. The line in a build script that indicates the dependence of a target on its dependencies. For example,

```
test.exe : main.obj sub.obj
```

declares that **test.exe** depends on **main.obj** and **sub.obj**.

DIRECTIVE. The instructions that control how **omake** proceeds. There are run-time directives and read-time directives, and some work both at read and run time. Directives are indicated by the appearance of either `%` or `!` as the first non-white-space character on a makefile line, followed by the directive name. If the `%` (or `!`) is in the leftmost column of the makefile, the directive is interpreted at read time; otherwise, it is interpreted at run time.

DIRECTORY SEPARATOR. The character in a pathname that separates directory names and the file name. For Windows NT, the separator is either \ or **/**. **omake** uses either character as the directory separator.

DOT DIRECTIVES. A read-time directive that modifies the operation of **omake** from the point at which it is encountered in the makefile.

EXIT STATUS. A number returned by an executed program and testable by **omake**. At the command line, you can check the exit status of the last executed program by using the **if errorlevel** command.

By convention, programs return a zero exit status when they finish without error and nonzero when an error occurs.

EXPLICIT RULE. A dependency line and build scripts that are used to make a target, in this form:

```
target : dependencies
   build script
   .
   .
   .
```

EXPLICIT DEPENDENCY. A dependency declared in a dependency line. For example, **def.h** is an explicit dependency of **main.obj** in the following:

```
main.obj : def.h
```

EXTENSION. The suffix part of the file name, usually used to denote the type of file. The extension consists of the characters that follow the last period in the file name to the end of the name.

FILE NAME. The part of the pathname after the last directory separator.

HEADER FILE. A file, containing source code, that is included into the body of a source file.

INCLUDE FILE. See header file.

INFERENCE RULE. A rule that generalizes the build process so that you do not have to specify how to build each target. For example, here is the built-in inference rule for making an **.obj** file from a **.c** file:

```
%.obj : %.c
    $(CC) $(CFLAGS) –c $.SOURCE
```

INFERRED DEPENDENCY. The dependency determined with an inference rule. For example, if **main.obj** is being built and the inference rule `%.obj : %.c` is used, **main.c** is an inferred dependency of **main.obj**.

INIT. An environment variable whose value is a semicolon-separated list of directory names. For example:

```
INIT = c:\home;c:\msc
```

KEEP WORKING. An operating mode of **omake**. While updating a target, if an executed build script returns a nonzero exit status, **omake** stops updating this target immediately. Any other targets that depend on this target as a source are not updated. This mode maximizes the amount of *safe* making and is ideal for running unattended builds (for example, rebuilding a large project overnight).

MACRO. The association of a *name* and a *value*. The macro expansion of macro *name* returns the *value*. Macros are used at read time as a means of organizing names of files, compiler options, and so on. At run time, macros also allow you to refer to the current target being built.

MAKEFILE. A file from which **omake** reads its instructions. An initialization file is read first, followed by one or more makefiles. The initialization file holds instructions for all make programs and is used to customize the operation of **omake**. The makefile has instructions for a specific project.

MODULE. A single file, such as a source or object file, that is combined with other files to build a project, such as an executable.

NULL STRING. A string with no characters.

PATH. The environment variable used to indicate the order in which directories are searched for executable files. For example, with a PATH of **c:\bin;c:\utils**, the program searches for executables in the current directory, in directory **c:\bin**, and in directory **c:\utils** in that order.

PATHNAME. The location of a target or file on disk, including any directory components.

READ TIME. The phase of **omake** in which it reads the makefiles.

READ-TIME DIRECTIVE. A directive that appears in the makefile with the directive character `%` or `!` in the leftmost column. Read-time directives are interpreted while the makefile is being read.

REGULAR EXPRESSION. A string of characters (some normal, some special) that allows the specification of a pattern to be matched.

RECURSION, RECURSIVE MAKE. The act of calling a program (or function) from itself. The usual context is in the expression "call **omake** recursively." This means a running copy of **omake** uses a build script to call a second copy of **omake**. Recursive makes are often used for projects that are split into multiple directories.

RESPONSE FILE. A text file used to hold long command lines. For many programs, the command-line option @*file* indicates that the program is to read its command line from the *file* response file.

ROOT NAME. For a target, its pathname minus its file name extension. This pathname includes any directory components.

RUN TIME. The phase of **omake** when it builds targets.

RUN-TIME DIRECTIVE. A directive that appears as a target's build script (the directive character `%` or `!` is indented from the leftmost column). Run-time directives are interpreted when the target's build scripts are executed.

SHELL LINE. See *build script*.

SOURCE. See *dependency.*

SOURCE FILE. A file containing source code that can be compiled into object code.

SPECIAL TARGET. A target whose name is of the form **.***NAME* and that has special meaning to **omake**.

STANDARD ERROR/OUTPUT. The two output streams (or file descriptors) that write to the console. Standard output is usually used to output general messages. Standard error is usually used to output error messages.

TARGET. Something that can be made. A target is usually a file, such as a source, object, or executable. The target is said to exist if the file is present on disk.

TARGET ATTRIBUTE. Properties assigned to targets.

TIME STAMP. The time stamp of a target is the time and date that a target was last changed. It is usually the creation or modification time of the file, as stored by the operating system.

UPDATE (A TARGET) . To execute a target's build scripts. This brings the target up to date.

UP TO DATE TARGET. A target that is current; that is, it has been compiled, linked, and so on, and is newer than all its dependencies.

WHITE SPACE. One or more space or tab characters.

WILDCARD CHARACTER. Characters used to match ambiguous part of a file name. Windows NT treats **?** as a wildcard character that matches any single character and ***** as a wildcard character that matches any number of characters in a file name or file name extension.

Index