Rational® ClearCase®

Building Software

VERSION: 2003.06.00 AND LATER

PART NUMBER: 800-026158-000

WINDOWS EDITION

Rational
the software development company

support@rational.com
http://www.rational.com

Third Party Notices, Code, Licenses, and Acknowledgements

Contents

Figures

Tables

Preface

Rational ClearCase is a comprehensive software version control and configuration management system.

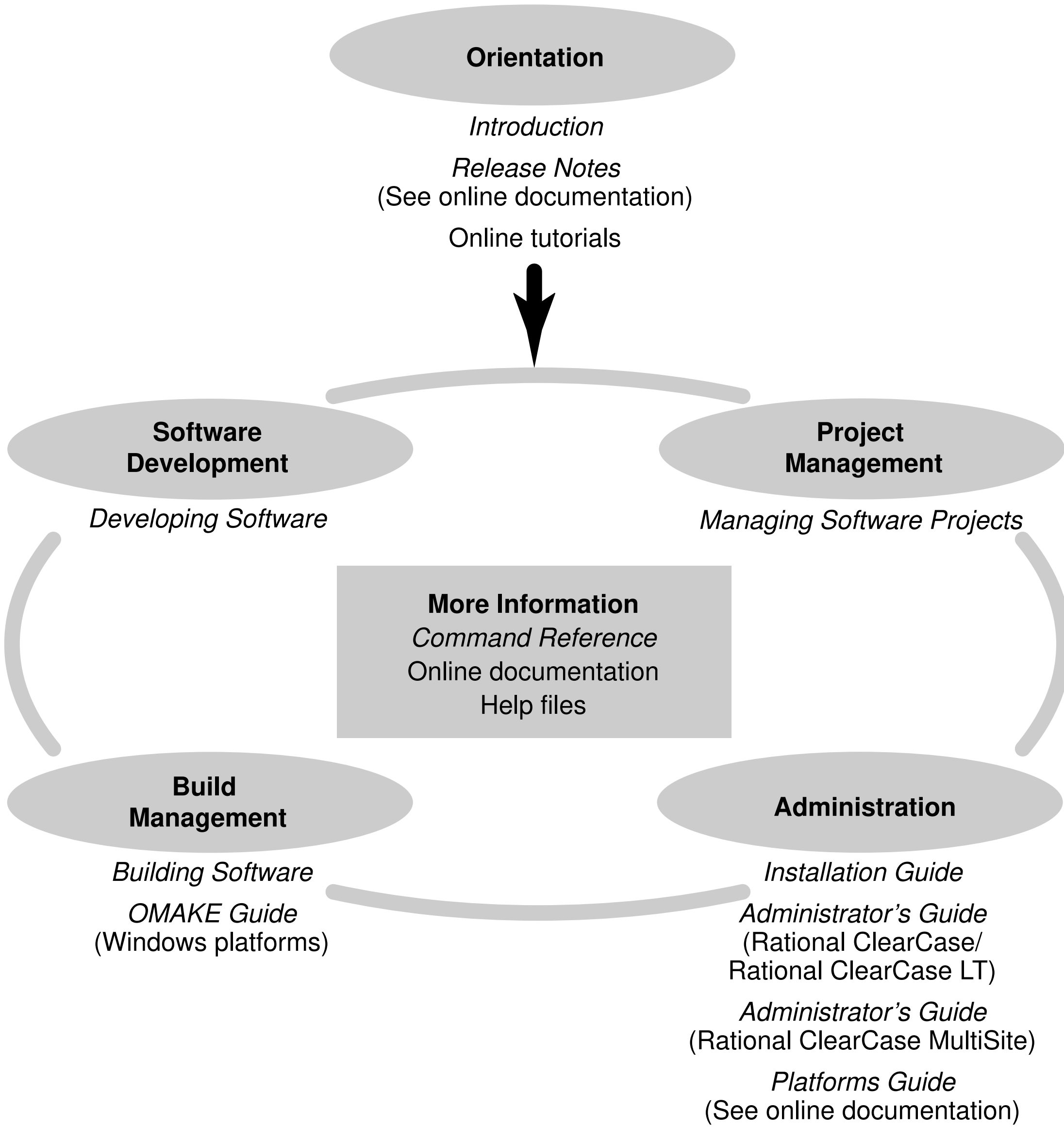
About This Manual

This manual provides an overview of ClearCase build management features and describes how to use ClearCase build tools. It is for new or experienced users of ClearCase who are familiar with software build concepts.

If you are not familiar with ClearCase build concepts and tools, read Chapter 1, *ClearCase Build Concepts*, Chapter 2, *Derived Objects and Configuration Records*, and Chapter 3, *Pointers on Using ClearCase Build Tools*.

For information about using ClearCase build tools with C++ programs or with Java tools, read Chapter 7, *Using ClearCase to Build C++ Programs* or Chapter 8, *Using ClearCase Build Tools with Java*.

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 /opt/rational/clearcase on UNIX and C:\Program Files\Rational\ClearCase on Windows.
- *cquest-home-dir* represents the directory into which Rational ClearQuest has been installed. By default, this directory is /opt/rational/clearquest on UNIX and C:\Program Files\Rational\ClearQuest on Windows.
- **Bold** is used for names the user can enter; for example, command names and branch names.
- A sans-serif font is used for file names, directory names, and file extensions.
- **A sans-serif bold font** is used for GUI elements; for example, menu names and names of check boxes.
- *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 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, you can use "**...**" within a pathname as a wildcard, similar to "*" or "?". For more information, see the **wildcards_ccase** reference page.
- If a command or option name has a short form, a "medial dot" (·) character indicates the shortest legal abbreviation. For example:

  lsc·heckout

Online Documentation

The ClearCase Product Family (CPF) includes online documentation, as follows:

Help System: Use the **Help** menu, the **Help** button, or the F1 key. To display the contents of the online documentation set, do one of the following:

- On UNIX, type **cleartool man contents**
- On Windows, click **Start > Programs > Rational Software > Rational ClearCase > Help**
- On either platform, to display contents for Rational ClearCase MultiSite, type **multitool man contents**
- Use the **Help** button in a dialog box to display information about that dialog box or press F1.

Reference Pages: Use the **cleartool man** and **multitool man** commands. For more information, see the **man** reference page.

Command Syntax: Use the **–help** command option or the **cleartool help** command.

Tutorial: Provides a step-by-step tour of important features of the product. To start the tutorial, do one of the following:

- On UNIX, type **cleartool man tutorial**
- On Windows, click **Start > Programs > Rational Software > Rational ClearCase > ClearCase Tutorial**

PDF Manuals: Navigate to:

- On UNIX, *ccase-home-dir*/doc/books
- On Windows, *ccase-home-dir*\doc\books

Customer Support

If you have any problems with the software or documentation, please contact Rational Customer Support by telephone, fax, or electronic mail as described below. For information regarding support hours, languages spoken, or other support information, click the **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**

ClearCase Build Concepts 1

Rational ClearCase supports makefile-based building of software systems and provides a software build environment closely resembling that of the **make** program. **make** was developed for UNIX systems and has been ported to other operating systems. You can use files controlled by ClearCase to build software, and use native **make** programs, third-party build utilities, your company's own build programs, or the ClearCase build tools **clearmake**, **omake**, and **clearaudit**.

The ClearCase build tools, **clearmake** and **omake**, provide compatibility with other **make** variants, along with powerful enhancements:

- Build auditing, with automatic detection of source dependencies, including header file dependencies
- Automatic creation of permanent bill-of-materials documentation of the build process and its results
- Sophisticated build-avoidance algorithms to guarantee correct results when building in a parallel development environment
- Sharing of binaries among views, saving both time and disk storage
- Parallel building, applying the resources of multiple processors to builds of large software systems, is available within clearmake

The **clearaudit** build tool provides build auditing and creation of bill-of-materials documentation.

clearmake, **omake**, and **clearaudit** are intended for use in dynamic views. You can use them in a snapshot view, but the features that distinguish them from ordinary **make** programs (build avoidance, build auditing, derived object sharing, and so on) are not enabled in snapshot views; therefore, these features are not available when using **clearmake** with Rational ClearCase LT.

Both **clearmake** and **omake** incorporate the major ClearCase build-related features described in the following sections. The **omake** program's strength lies primarily in its support for users who require compatibility with other Windows- based build programs. For details specific to **omake**, see the *OMAKE Guide*. In all other build-related documentation, the primary emphasis is on **clearmake** behavior.

Overview of the ClearCase Build Scheme

Developers perform builds, along with all other work related to ClearCase, in views. Typically, developers work in separate, private views. Sometimes, a team shares a single view (for example, during a software integration period).

As described in *Developing Software*, each view provides a complete environment for building software that includes a particular configuration of source versions and a private work area in which you can modify source files, and use build tools to create object modules, executables, and so on.

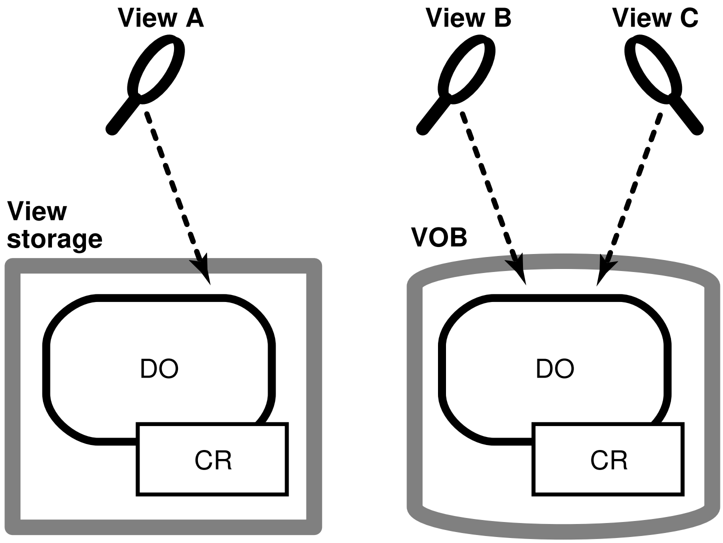
As a build environment, each view is partially isolated from other views. Building software in one view never disturbs the work in another view, even another build of the same program at the same time. However, when working in a dynamic view, you can examine and benefit from work done previously in another dynamic view. A new build shares files created by previous builds, when appropriate. This sharing saves the time and disk space involved in building new objects that duplicate existing ones.

You can (but need not) determine what other builds have taken place in a directory, across all dynamic views. ClearCase includes tools for listing and comparing past builds.

The key to this scheme is that the project team's VOBs constitute a globally accessible repository for files created by builds, in the same way that they provide a repository for the source files that go into builds. A file produced by a software build is a derived object (DO). Associated with each derived object is a configuration record (CR), which **clearmake** or **omake** uses during subsequent builds to determine whether the DO can be reused or shared.

Figure 1 illustrates the ClearCase software build scheme.

Figure 1 Building Software with ClearCase: Isolation and Sharing



The section *Dependency Tracking of MVFS and Non-MVFS Files* on page 5 describes how ClearCase keeps track of the objects produced by software builds. *Build Avoidance* on page 6 describes the mechanism that enables such objects to be shared among views.

View Context Required

For a build that uses the data in one or more VOBs, the shell or command interpreter from which you invoke **clearmake** must have a view context. On Windows systems, you must be on the dynamic-views drive (by default, drive M) or a drive assigned to a view. If you want derived objects to be shared among views, you must be on a drive assigned to a view.

You can build objects in a standard directory, without a view context, but doing so disables many of **clearmake**'s special features.

How Builds Work

In many ways, ClearCase builds adhere closely to the standard **make** paradigm:

1. You invoke **clearmake**, optionally specifying the names of one or more targets. (Such explicitly specified targets are called *goal targets*.)
2. **clearmake** reads zero or more makefiles, each of which contains targets and their associated build scripts. It also reads zero or more build options specification (BOS) files, which supplement the information in the makefiles.

3 **clearmake** supplements the makefile-based software build instructions with its own built-in rules or, when it runs in a compatibility mode, with built-in rules specific to that mode.

4 For each target, **clearmake** performs build avoidance, determining whether it actually needs to execute the associated build script (target rebuild). It takes into account both source dependencies (Have any changes occurred in source files used in building the target?) and build dependencies (Must other targets be updated before this one?).

5 If a target meets **clearmake**'s rebuild criteria, **clearmake** executes its build script.

Build Reference Time and Build Sessions

As your build progresses, other developers can continue to work on their files and may check in new versions of elements that your build uses. If your build takes an hour to complete, you do not want build scripts executed early in the build to use version 6 of a header file and scripts executed later to use version 7 or 8. To prevent such inconsistencies, **clearmake** locks out any version that meets both of these conditions:

- The version is selected by a configuration specification rule that includes the **LATEST** version label.
- The version was checked in after the time the build began (the build reference time).

This reference-time facility applies to checked-in versions of elements only; it does not lock out changes to checked-out versions, other view-private files, and non-MVFS objects. **clearmake** adjusts for time discrepancies between system clocks on different hosts in a network (clock skew).

Exit Status

clearmake returns a zero exit status if all goal targets are successfully processed. It returns a nonzero exit status in two cases:

- **clearmake** itself detects an error, such as a syntax error in the makefile. In this case, the error message includes the string `clearmake`.
- A makefile build script terminates with a nonzero exit status (for example, a compiler error).

Dependency Tracking of MVFS and Non-MVFS Files

During build-script execution in a dynamic view, a host's MVFS (multiversion file system) audits low-level system calls performed on ClearCase data: **create**, **open**, **read**, and so on. Calls involving the following objects are monitored:

- Versions of elements used as build input
- View-private files used as build input (for example, the checked-out version of a file element)
- Files created within VOB directories during the build

Some of these objects are stored in the VOB, and others are view-private files. The view combines them into a virtual work area, where they appear to be located in VOB directories. They are called *MVFS files* because they are accessed through the MVFS.

Automatic Detection of MVFS Dependencies

Because auditing of MVFS files is completely automated, you do not have to keep track of which files are being used in builds. ClearCase does the tracking instead. For example, it determines which C-language source files referenced with **#include** directives are used. Tracking eliminates the need both to declare such files in the makefile and for dependency-detection tools.

If you store your build tools (compilers, linkers, and so on) as ClearCase elements and run them from the VOB, they are recorded in the configuration record as implicit detected dependencies.

Tracking Non-MVFS Files

A build can also involve files that are not accessed through VOB directories. Such non-MVFS files are not audited automatically, but are tracked if you declare them as dependencies in a makefile. This tracking enables auditing of build tools that are not stored as ClearCase elements (for example, a C-language compiler), flag files in the user's home directory, and so on. Tracking information on a non-MVFS file includes its absolute path, time stamp, size, and checksum.

Derived Objects and Configuration Records

When it finishes executing a build script, **clearmake** or **omake** records the results, including build audit information, in the form of derived objects and configuration records.

A derived object (DO) is a file created in a VOB during a build or build audit with **clearmake** or **omake**. Each DO has an associated configuration record (CR), which is the bill of materials for the DO. The CR documents aspects of the build environment, the assembly procedure for a DO, and all the files involved in the creation of the DO.

Note: All derived objects created by executing a build script have equal status, even though some of them may be explicit build targets, and others may be created as side effects of the build script (for example, compiler listing files). The term *siblings* describes a group of DOs created by the same script and associated with a single CR.

For more information about DOs and CRs, see Chapter 2, *Derived Objects and Configuration Records*.

Build Avoidance

The build tool attempts to avoid rebuilding derived objects. If an appropriate derived object exists in the view, **clearmake** or **omake** reuses that DO. If there is no appropriate DO in the view, **clearmake** or **omake** looks for an existing DO built in another view that can be winked in to the current view. The search process is called *shopping*.

The process of qualifying a candidate DO is called *configuration lookup*. It involves matching information in the VOB from the candidate DO's config record against the user's current *build configuration*. This process guarantees correct results in a parallel development environment, which the standard time-stamp-based algorithm used by **make** cannot do. Even if an object module is newer than a particular version of its source file, the module may have been built using a different version. In fact, reusing object modules and executables built recently is likely to be incorrect when rebuilding a previous release of an application from old sources. The configuration lookup algorithm that ClearCase uses guarantees that your builds are both correct (inappropriate objects are not reused) and optimal (appropriate objects are always reused).

For a DO to be reused or winked in, the build configuration documented in its configuration record must match the current view's build configuration. The build configuration consists of two items:

- **Files**. The versions of elements listed in the CR must match the versions selected by the view in which the build is performed. Any view-private files or non-MVFS files listed in the CR must also match.
- **Build procedure**. The build options in the CR must match the build options specified on the command line, in the environment, in makefiles, or in build options specification files. The build script listed in the CR must match the script that will be executed if the target is rebuilt. The scripts are compared with all make

macros expanded; thus, a match occurs only if the same build options apply (for example, “compile for debugging”).

The search ends when **clearmake** or **omake** finds a DO whose configuration matches the view’s current build configuration exactly. In general, a configuration lookup can have three outcomes:

- **Reuse**. If the DO (and its siblings) in the view match the build configuration, **clearmake** or **omake** keeps them.
- **Winkin**. If a DO built previously matches the build configuration, **clearmake** or **omake** causes that DO and its siblings to appear in this view. This operation is called *winkin*.

  Note: The build tool does not contact all views to determine whether they contain DOs that can be winked in. Instead, it uses DO information in the VOB to eliminate inappropriate candidates. Only if it finds a candidate does it contact the containing view to retrieve the DO’s configuration record.
- **Rebuild**. If configuration lookup fails to find a DO that matches the build configuration, **clearmake** or **omake** executes the target’s build script, which creates one or more new DOs and a new CR.

Reuse and winkin take place only if **clearmake** or **omake** determines that a newly built derived object would be identical to the existing one. Winkin takes place when two or more views select the same versions of source elements used in a build. For example, you can create another view that has the same configuration as an existing view. Initially, the new view sees all the sources but contains no derived objects. Running **clearmake** or **omake** winks in many derived objects from the existing view.

Hierarchical Builds

In a hierarchical build, some objects are built and then used to build others. The build tool performs configuration lookup separately for each target. To ensure a consistent result, the build tool also applies this principle: When a new object is created, all targets that depend on it are rebuilt. Note that winkin does not cause rebuilds of dependencies.

Automatic Dependency Detection

Configuration records enable automatic checking of source dependencies as part of build avoidance. All such dependencies (for example, on C-language header files) are logged in a build’s configuration record, whether or not they are explicitly declared in a makefile.

Express Builds

During an audited build, the build tool writes to the VOB information about a newly built DO. Configuration lookup by future builds uses that information to determine whether the DO is a candidate for winkin.

There is a performance tradeoff when you create DOs. While the build is writing the DO information to the VOB database, other users cannot write to the VOB. This performance loss is offset when the DO is used by subsequent builds, which can make those builds faster. But if the DO is never used by another view, the performance loss is not offset.

ClearCase express builds create derived objects, but do not write information to the VOB. These DOs are nonshareable and are not considered for winkin by other views. They can be reused only by the view in which they were built.

Express builds offer two advantages over regular builds:

- Scalability. During an express build, write access to the VOB is not blocked by time-consuming DO write operations. More users can build in a VOB without making VOB access slower.
- Performance. Express builds are faster than regular builds, because the build does not write DO information into the VOB.

Which kind of build occurs when you invoke **clearmake** or **omake** depends on how your view is configured. To use express builds, you must use a dynamic view whose DO property is set to nonshareable. For information about enabling express builds, see *Using Express Builds to Prevent Winkin to Other Views* on page 52.

Build Auditing with clearaudit

Some organizations, or some developers, may want to use ClearCase build auditing without using the **clearmake** or **omake** program. Others may want to audit development activities that do not involve makefiles. These users can do their work in an audited shell, which is a standard task with build auditing enabled.

All MVFS files read during execution of the audited shell are listed as inputs to the build. All MVFS files created become derived objects, associated with the single configuration record.

For more information, see the **clearaudit** reference page.

Compatibility with Other make Programs

Many **make** utilities are available in the multiple-architecture, multiple-vendor world of open systems. The ClearCase **clearmake** and **omake** programs share features with many of them and have some unique features.

You can adjust the level of compatibility that **clearmake** or **omake** has with other **make** programs:

- Suppress special features of **clearmake** or **omake**.

  Use command options to turn off such features as winkin, comparison of build scripts, comparison of detected dependencies, and creation of DOs and CRs. You can turn off configuration lookup altogether, so that the standard time-stamp-based algorithm is used for build avoidance.

- Enable features of other **make** programs.

  clearmake and **omake** have several compatibility modes, which provide for partial emulations of popular **make** programs. For more information, see the *OMAKE Guide* and the makefile compatibility commands in the *Command Reference*.

To achieve absolute compatibility with other **make** programs, you can actually use them to perform builds. However, builds with a standard **make** do not provide build auditing, configuration lookup, or sharing of DOs. The MVFS files that the build creates are view-private files, not derived objects. To perform an audited build, you can execute the **make** program in a **clearaudit** shell.

Parallel Building

The Parallel Build Procedure

Before starting a build, **clearmake** parses the makefile to analyze the hierarchy of intermediate targets needed to build the final target. In a serial build, **clearmake** constructs each target (or reuses an existing DO) before continuing the analysis of subsequent builds.

In a parallel build, when **clearmake** detects that a target is out of date, it dispatches the build script for that target to the host. **clearmake** continues to analyze the build hierarchy to detect other targets that can be built at the same time. The total number of build scripts being executed at any particular time is equal to or less than the number you specify with **–J**. Each target is built as soon as system resources on the build host allow.

Execution of build scripts is managed on each remote host by the ClearCase audited build executor (**abe**), which is invoked through standard remote-shell facilities.

For more information about parallel builds, see Chapter 9, *Setting Up a Parallel Build*.

Derived Objects and Configuration Records

2

This chapter describes derived objects and configuration records. Rational ClearCase creates derived objects and configuration records only if you build in a dynamic view with one of the ClearCase build tools. For information about managing derived objects and configuration records, see Chapter 4.

Derived Objects Overview

As described in Chapter 1, derived objects are created during builds with ClearCase build tools. They are used for build avoidance and derived object sharing.

In a parallel-development environment, it is likely that many DOs with the same pathname will exist at the same time. For example, suppose that source file msg.c is being developed on three branches concurrently, in three different views. ClearCase builds performed in those three views produce object modules named msg.obj. Each of these is a DO, and each has the same standard pathname, for example, \proj\src\msg.obj.

Note: Symbolic links created by a build script and files created in non-VOB directories are not DOs.

In addition, each DO can be accessed with ClearCase extended names:

- Within each dynamic view, a standard Windows NT pathname accesses the DO referenced by that view. This is another example of the ClearCase transparency feature.

`msg.obj`	*(the DO in the current view)*

- You can use a view-extended pathname to access a DO in any view:

`M:\drp\proj\src\msg.obj`	*(the DO in view drp)*
`M:\R2_integ\proj\src\msg.obj`	*(the DO in view R2_integ)*

Derived Object Naming

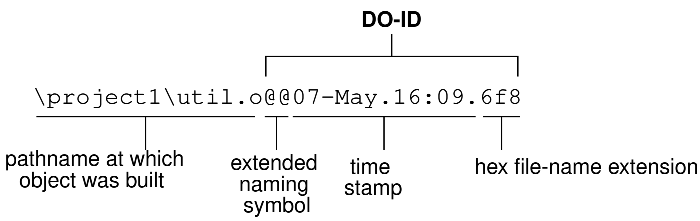
No name collisions occur among derived objects built at the same pathname, because each DO is cataloged in the VOB database with a unique identifier, its *DO ID*. The DO

ID references a DO independently of views. The **lsdo** command can list all DOs created at a specified pathname, regardless of which views (if any) can select them:

```
Z:\myvob> cleartool lsdo hello.obj
07-May.16:09   akp   "hello.obj@@07-May.16:09.623" on neptune
06-May.12:47   akp   "hello.obj@@06-May.12:47.539" on neptune
01-May.21:49   akp   "hello.obj@@01-May.21:49.282" on neptune
03-Apr.21:40   akp   "hello.obj@@01-May.21:40.226" on neptune
```

Together, a DO's standard name (hello.o) and its DO ID (**07-May.16:09.623**) constitute a VOB-extended pathname to that particular derived object (Figure 2). (The extended naming symbol is host specific; most organizations use the default value, @@.)

Figure 2 Extended Pathname of a Derived Object



Standard software must access a DO through a dynamic view, using a standard pathname or view-extended pathname. You can use such names with debuggers, profilers, and so on. Only ClearCase programs can reference a DO using a VOB-extended pathname, and only the DO's metadata is accessible in this way. You can use a view-extended pathname with the **winkin** command, to make the file system data of any DO available to your view. See *Winking In a DO Manually* on page 51.

The following example describes a DO with an extended pathname (hello.exe@@07-Mar.11:40.217) and its configuration record:

```
Z:\akp_hw\src> cleartool describe hello.exe@@07-Mar.11:40.217
  created 07-Mar-03.11:40.217 by akp.users@phobos
  references: 1 =>   C:\users\views\akp\tut\old.vws
```

```
Z:\akp_hw\src> cleartool catcr hello.exe@@07-Mar.11:40.217
Target hello.exe built by akp.user
Host "cobalt" running NT 3.50 (i586)
Reference Time 07-Mar-03.11:40:41, this audit started
 07-Mar-03.11:40:46
View was C:\users\views\akp\tut\old.vws
Initial working directory was M:\akp_main\akp_hw\src
----------------------------
MVFS objects:
----------------------------
\akp_hw\src\hello.exe@@07-Mar.11:40.217
\akp_hw\src\hello.obj@@07-Mar.11:40.213
\akp_hw\src\util.obj@@07-Mar.11:40.215
----------------------------
Variables and Options:
----------------------------
MKTUT_LK=link
----------------------------
Build Script:
----------------------------
  link -out:hello.exe hello.obj util.obj
----------------------------

Z:\akp_hw\src> dir hello.exe@@07-Mar.11:40.217
...
File Not Found
```

Configuration Records

A configuration record (CR) is the bill of materials for a derived object or set of DOs. The CR documents aspects of the build environment, the assembly procedure for a DO, and all the files involved in the creation of the DO.

Configuration Record Example

The **catcr** command displays the configuration record of a specified DO. Figure 3 shows a CR, with annotations to indicate the various kinds of information in the listing.

Figure 3 Kinds of Information in a Configuration Record

```
Z:\vob_hw\src> cleartool catcr util.obj
Target util.obj built by mike.dvt
Host "proton" running NT 3.51 (i586)
Reference Time 26-Feb-03.20:41:33,
this audit started 26-Feb-03:20:41:34
View was C:\views\mike\mike.vws
Initial working directory was M:\mike_vw\vob_hw\src
-------------------------
MVFS objects:
-------------------------
\vob_hw\src\hello.h@@\main\2      <25-Feb-03.17:03:11
\vob_hw\src\my.flag.file          <26-Feb-03.20:21:56
\vob_hw\src\util.c                <25-Feb-03.17:02:27
\vob_hw\src\util.obj@@26-Feb.20:41:465
-------------------------
non-MVFS objects:
-------------------------
c:\tmp\build.notes.1              <26-Feb-03.20:31:30
-------------------------
Variables and Options:
-------------------------
CCPU=nt_i386
-------------------------
Build Script:
-------------------------
    cl util.c
-------------------------
```

version of source element, listed with version-ID

view-private file

checked-out version, highlighted and listed with standard pathname

DO created in this build, listed with DO-ID

information from *makefile*

Some notes on Figure 3:

- 03Version of source element, listed with version-ID. By default, **catcr** does not list versions of the VOB directories involved in a build. To list this information, use the **–long** option:

  cleartool catcr –long util.obj

  ```
  directory version \vob_hw\.@@\main\1     <25-Feb-03.16:59:31>
  directory version \vob_hw\src@@\main\3   <26-Feb-03.20:53:07>
  ...
  ```

- Declared dependencies. One principal feature of ClearCase is the automatic detection of source dependencies on MVFS files: versions of elements and objects in view-private storage.

- Listing of checked-out versions. Checked-out versions of file elements are highlighted. Checked-out versions of directory elements are listed like this:

  ```
  directory version \vob_hw\src@@\main\CHECKEDOUT
  <26-Feb-03.17:05:23>
  ```

  When the elements are subsequently checked in, a listing of the same configuration record shows the updated information. For example,

  ```
  \vob_hw\src\util.c                <25-Feb-03.17:02:27>
  ```

  becomes

```
\vob_hw\src\util.c@@\main\4        <25-Feb-03.17:02:27>
```

The actual configuration record contains a ClearCase internal identifier for each MVFS object. After the version is checked in, **catcr** lists that object differently.

Notes:

- The time stamps in the configuration record are for informational purposes; they are not used for rebuild or winkin decisions. ClearCase uses OIDs to track versions used in builds.
- It is possible to produce two configuration records that show different time stamps for the same version of an object. If an object is checked out before a build, **clearmake** records the checkout time as the most recent modification time of the object. If you then check in the object and rebuild, **clearmake** records the checkin time as the most recent modification time of the object. Comparing the configuration records from both builds then shows that the same version of the object has different time stamps.

Contents of a Configuration Record

The following sections describe the contents of configuration records.

Header Section

As displayed by **catcr**, the header section of a CR includes the following lines:

- Makefile target associated with the build script and the user who started the build:

  ```
  Target util.obj built by akp.dvt
  ```

  For a CR produced by **clearaudit**, the target is `ClearAudit_Shell`.

- Host on which the build script was executed:

  ```
  Host 'mars' running Windows NT 4.0
  ```

- Reference time of the build (the time **clearmake**, **omake**, or **clearaudit** began execution), and the time when the build script for this particular CR began execution:

  ```
  Reference Time 15-Sep-03.08:18:56, this audit started
  15-Sep-03.08:19:00
  ```

  In a hierarchical build, involving execution of multiple build scripts, all the resulting CRs share the same reference time. (For more about reference time, see the **clearmake** reference page.)

- View storage directory of the view in which the build took place:

  ```
  View was \\mars\views\930825.vws
  ```

- Working directory at the time build script execution or **clearaudit** execution began:

```
Initial working directory was s:\proj\hw\src
```

MVFS Objects Section

An *MVFS object* is a file or directory in a VOB. The MVFS Objects section of a CR includes this information:

- Each MVFS file or directory read during the build. These include versions of elements and view-private files used as build input, checked-out versions of file elements, DOs read, and any tools or scripts used during the build that are under version control.
- Each derived object produced by the target rebuild.

Non-MVFS Objects Section

A non-MVFS object is an object that is accessed outside a VOB (compiler, system-supplied header file, temporary file, and so on). The Non-MVFS Objects section of a CR includes each non-MVFS file that appears as an explicit dependency in the makefile or is a dependency inferred from a suffix rule. See *Declaring Source Dependencies in Makefiles* on page 33.

This section is omitted if there are no such files or if the CR was produced by **clearaudit**.

Variables and Options Section

The Variables and Options section of a CR lists the values of make macros referenced by the build script.

This section is omitted from a CR produced by **clearaudit**.

Build Script Section

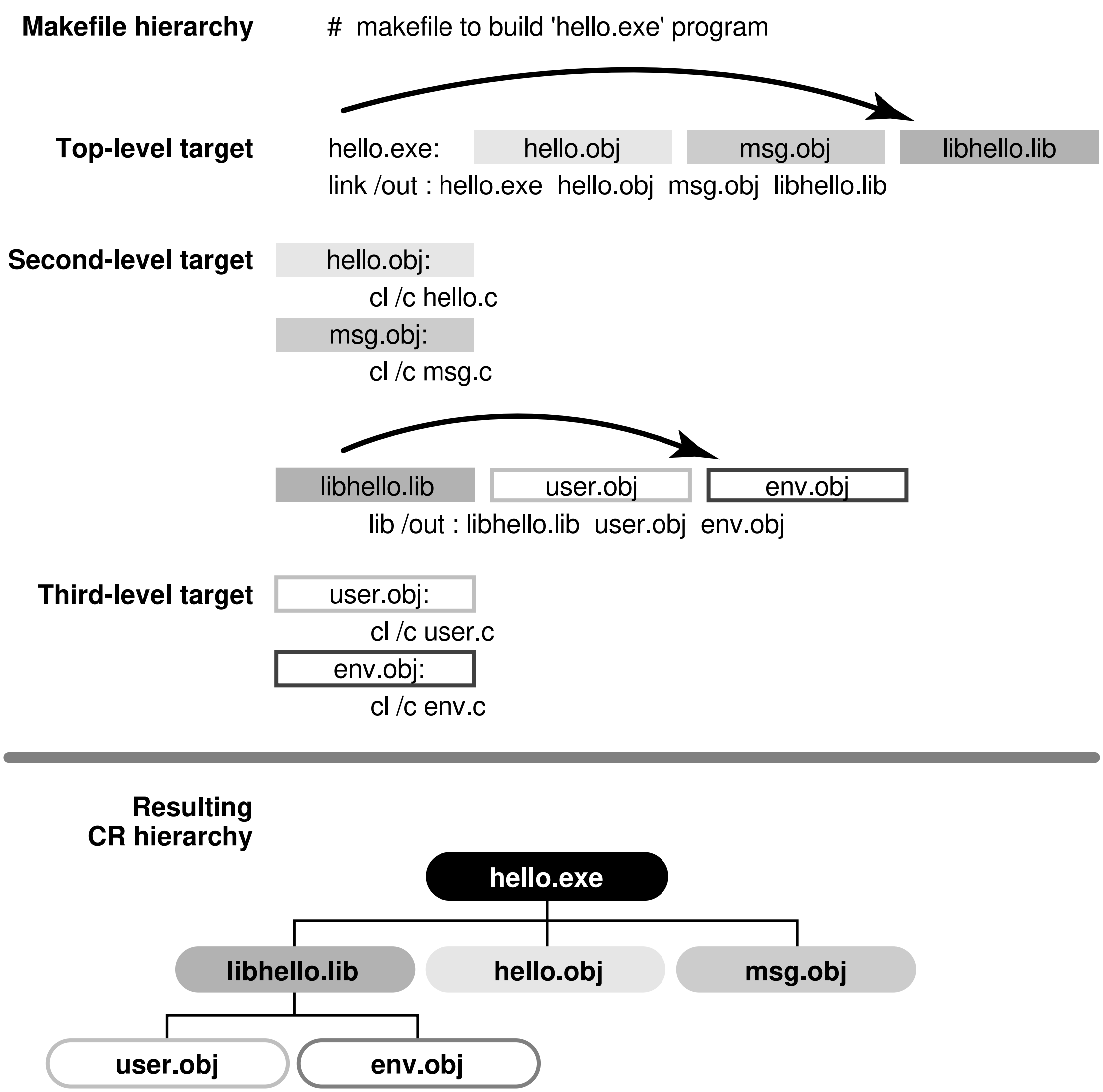
The Build Script section of a CR lists the script that was read from a makefile and executed by **clearmake** or **omake**.

This section is omitted from a CR produced by **clearaudit**.

Configuration Record Hierarchies

A typical makefile has a hierarchical structure. Thus, running **clearmake** or **omake** once to build a high-level target can cause multiple build scripts to be executed and multiple CRs to be created. Such a set of CRs can form a configuration record hierarchy, which reflects the structure of the makefile (Figure 4).

Figure 4 Configuration Record Hierarchy



An individual parent/child link in a CR hierarchy is established in one of two ways:

- In a target/dependencies line

For example, the following target/dependencies line declares derived objects hello.obj, msg.obj, and libhello.lib as build dependencies of derived object hello.exe:

```
hello.exe: hello.obj msg.obj libhello.lib
...
```

Accordingly, the CR for hello.exe is the parent of the CRs for the .obj files and the .lib file.

- In a build script

  For example, in the following build script, derived object libhello.lib in another directory is referenced in the build script for derived object hello.exe:

```
hello.exe: $(OBJS)
     cd ..\lib & $(MAKE) libhello.lib
     link /out: hello.exe $(OBJS) ..\lib\libhello.lib
```

  Accordingly, the CR for hello.exe is the parent of the CR for libhello.lib.

  Note: The recursive invocation of **clearmake** in the first line of this build script produces a separate CR hierarchy, which is not necessarily linked to the CR for hello.exe. The second line of the build script links the CR for ..\lib\libhello.lib with that of hello.exe by causing **link** to read ..\lib\libhello.lib and making it a detected dependency.

The **catcr** and **diffcr** commands have options for handling CR hierarchies:

- By default, they process individual CRs.
- With the **–recurse** option, they process the entire CR hierarchy of each derived object specified, keeping the individual CRs separate.
- With the **–flat** option, they combine (or flatten) the CR hierarchy of each specified derived object.

Some ClearCase features process entire CR hierarchies automatically. For example, when the **mklabel** command attaches version labels to all versions used to build a particular derived object (**mklabel –config**), it uses the entire CR hierarchy of the specified DO. Similarly, ClearCase maintenance procedures do not scrub the CR associated with a deleted DO if it is a member of the CR hierarchy of a higher-level DO.

Configuration Record Cache

When a derived object is created in a view, both its data container and its associated configuration record are stored in the view's private storage area. The CR is stored in the view database, in compressed format. To speed configuration lookup during subsequent builds in this view, a compressed copy of the CR is also cached in a view-private file, .cmake.state, located in the directory that was current when the build started.

When a DO is winked in for the first time, the associated CR moves from the view's private storage area to the VOB database, as shown in Figure 5.

Kinds of Derived Objects

During a regular build, ClearCase build tools create shareable derived objects. During an express build, they create nonshareable derived objects. Both kinds of DOs have configuration records, but only shareable DOs can be winked in by other views.

The following sections describe the kinds of DOs and their life cycles.

Shareable DOs

When a ClearCase build tool creates a shareable DO, it creates a configuration record for the DO and writes information about the DO into the VOB. (At this point, the DO is shareable but unshared.) Builds in other views use this information during configuration lookup. If the build determines that it can wink in an existing DO, it contacts the view containing the DO and promotes the DO to the VOB. (The DO is now shareable and shared.)

As noted in *Express Builds* on page 8, you must consider whether the performance benefit of winking in DOs is worth the performance cost of making them available for winkin.

Note: The process of looking for a DO to wink in uses an efficient algorithm to eliminate mismatches. The build tool does not contact other views to retrieve configuration records unless the configuration lookup process determines that there is a winkin candidate.

The configuration lookup process cannot guarantee that the DO is suitable for use. The process uses details in the config record to determine whether a DO is suitable for winkin, but the config record does not record all parameters of a build. For example, a config record may list only a compiler's name and the options used. If two builds use incompatible compilers with the same name, unwanted winkins from one build to the other can occur.

Note: To minimize occurrences of incorrect winkin, all developers must use the same set of tools. An effective way to do so is to put your build tools under version control and always run them from the VOB.

Nonshareable DOs

During an express build, the ClearCase build tool creates nonshareable DOs. The build tool creates a configuration record for the DO, but does not write information about the DO into the VOB. Because scanning the information in the VOB is the only method other builds use to find DOs, other builds cannot wink in nonshareable DOs. However, a nonshareable DO can be reused by the view in which it was built.

A nonshareable DO can have shareable sub-DOs, but not shareable siblings. A nonshareable DO can be built using a winked-in shareable DO. (However, a shareable DO cannot have nonshareable sub-DOs or siblings.)

For information about enabling express builds, see *Using Express Builds to Prevent Winkin to Other Views* on page 52.

You can use the same commands that you use with shareable DOs on nonshareable DOs, but some commands work differently on the two kinds of DOs. The reference pages for the commands describe the differences.

Storage of Derived Objects

When a DO is created, its data container is located in the view storage area. For a shareable DO, the ClearCase build tool creates the VOB database object for the DO; it also writes to the VOB information about the DO that can be used during configuration lookup. A nonshareable DO has no VOB database object, and the build tool does not write any configuration lookup information into the VOB (Figure 5).

A DO consists of the following parts:

- VOB database object (shareable DOs only). Each DO is cataloged in the VOB database, where it is identified by an extended name that includes both its standard pathname (for example, \hw\src\hello.c) and a unique DO ID (for example, `23-Feb.08:41.391`).
- Data container. The data portion of a derived object is stored in a standard file within a ClearCase storage area. This file is called a *data container*; it contains the DO's file system data.
- Configuration record. Actually, a CR is associated with a DO; it is not part of the DO itself. More precisely, a CR is associated with the entire set of sibling DOs created by a particular invocation of a particular build script. See *Configuration Records* on page 13.

When a shareable DO is first created, it is unshared:

- It appears only in that view.
- Its data container is a file in the view's private storage area.
- **clearmake** or **omake** writes information about the DO into the VOB.

Promotion and Winkin

The first time a shareable derived object is winked in by another dynamic view or when either kind of DO is promoted manually with a **winkin** or **view_scrubber –p** command, its status changes to shared:

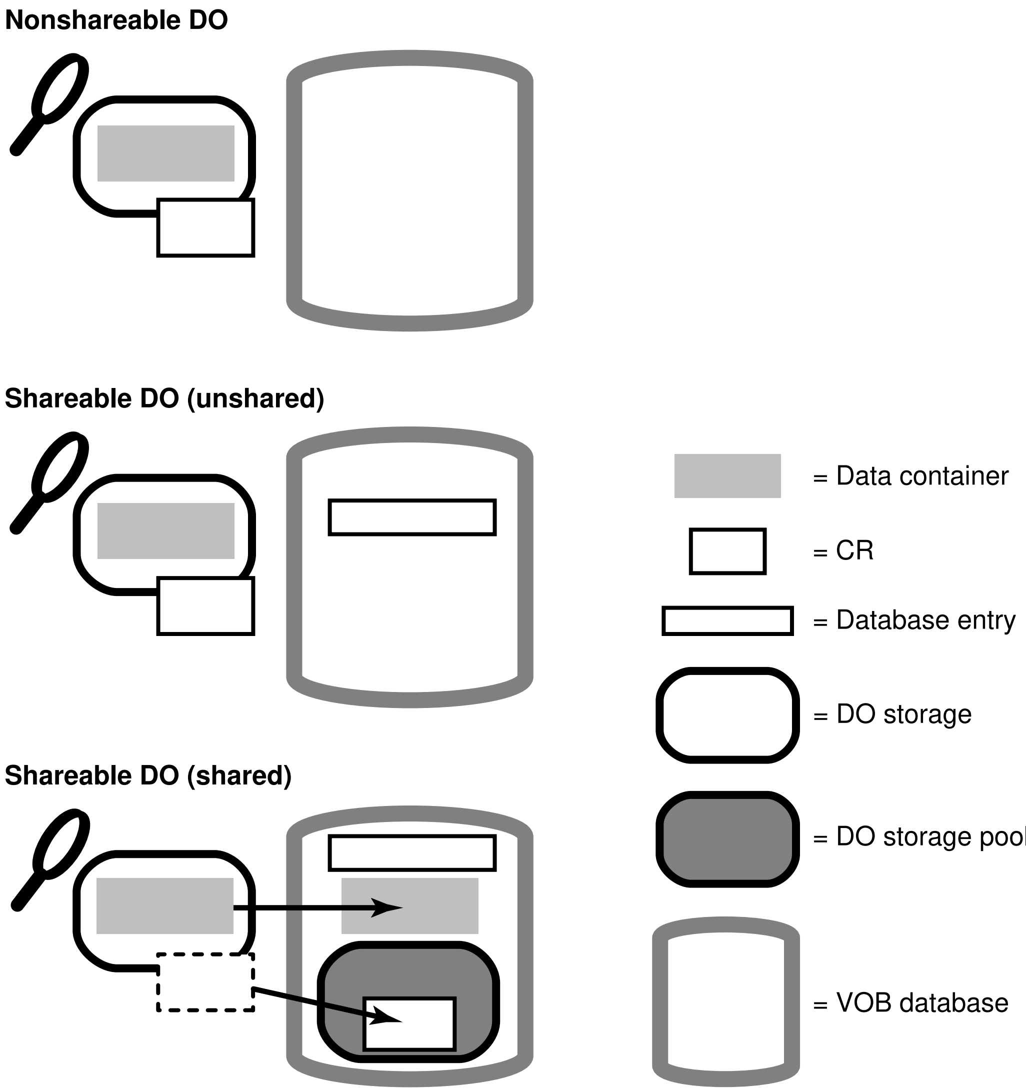
- Its data container is *promoted* to a derived object storage pool in the VOB.
- (Shareable DOs only) If the winkin was done by the build tool or the command was executed in another view, the DO now appears in two dynamic views.

When the winkin occurs during a **clearmake** or **omake** build:

- The dynamic view to which the DO is winked in, and all other views to which the DO is subsequently winked in, use the data container in VOB storage.
- The original view continues to use the data container in view storage. (The **view_scrubber** utility removes this asymmetry, which causes all dynamic views to use the data container in VOB storage.)

When the winkin is done with the **winkin** or **view_scrubber –p** command, the data container in the view is removed after it is promoted to VOB storage. The original view and all other views to which the DO is subsequently winked in use the data container in VOB storage.

Figure 5 Storage of a Shareable Derived Object



After a derived object is winked in, it remains shared, no matter how many times it is winked in to additional dynamic views, and even if subsequent rebuilds or deletion commands cause it to appear in only one dynamic view (or zero views).

When a derived object's data container is in the VOB, any number of views can share the derived object without having to communicate with each other directly. For example, view **alpha** can be unaware of views **beta** and **gamma**, with which it shares a derived object. The hosts on which the view storage directories are located need not have network access to each other's disk storage.

If **clearmake** attempts a winkin that fails, it checks to see if any VOBs are locked. If it finds a locked VOB, it waits until the VOB is unlocked and then retries the winkin.

For more information, see the **winkin** and **view_scrubber** reference pages.

DO Versions

You can check in a derived object as a version of an element, creating a *DO version*. Other versions of such an element can also be, but need not be, derived objects. A DO version behaves like both a version and a derived object:

- You can use its version ID to reference it as both a VOB database object and a data file.
- You can apply a version label to it and use that label to reference it.
- You can display its configuration record with **catcr** or compare the CR to another with **diffcr**.
- A **clearmake** or **omake** build can wink it in if the element is located at the same pathname where the DO was originally built.
- You can wink it in with a **winkin** command.
- The **describe** command lists it as a `derived object version`. (The **lsdo** command does not list it at all.)

For more information about DO versions, see *Working with DO Versions* on page 54.

Reuse of DO IDs

The DO ID for a shareable derived object is guaranteed to be unique within the VOB, for all views. That is, if you delete a shareable DO, its numeric file name extension is not reused (unless you reformat the VOB that contains it).

The DO ID for a DO created by an express build (a nonshareable derived object) is unique only at a certain point in time. If you delete a nonshareable DO, the ClearCase build tools can reuse its numeric file name extension. (Because ClearCase tracks derived objects using their VOB database identifiers, no build confusion occurs if a file name extension is reused.)

DO IDs change when any of these events occur:

- The DO passes its first birthday. The time stamp changes to include the year the DO was created:

  `util.obj@@15-Jul.15:34.8896` *(when first created)*

  `util.obj@@15-Jul-2002.8896` *(after a year)*

- You convert a nonshareable DO to a shareable DO. (See *Converting Nonshareable DOs to Shared DOs* on page 59.)

- You process a VOB's database with **reformatvob**. All DO-IDs receive new numeric file-name extensions:

  `util.obj@@15-Jul.15:34.8896` *(before reformatvob)*

  `util.obj@@15-Jul.17:08.734` *(after reformatvob)*

The configuration record reflects these DO-ID changes.

Derived Object Reference Counts

A DO's reference count is the number of times the derived object appears in ClearCase dynamic views throughout the network. ClearCase also tracks the identifiers for the views that reference the DO. When a new derived object is created, **clearmake** sets its reference count to **1**, indicating that it is visible in one view. Thereafter, each winkin of the DO to an additional view increments the reference count.

The **lsdo –long** command lists the reference count and referencing views for a DO. For example:

cleartool lsdo –long

```
01-Sep-03.18:56:45     Suzanne Lee (sgl.user@neon)
  create derived object "file.txt@@01-Sep.18:56.2147483683"
  size of derived object is: 10
  last access: 01-Sep-03.18:56:46
  references: 1  => neon:C:\views\sgl_test.vws
01-Sep-03.19:03:19     Suzanne Lee (sgl.user@neon)
  create derived object "util@@01-Sep.19:03.81"
  size of derived object is: 10
  last access: 01-Sep-03.19:03:33
  references: 2 (shared)
  => neon:C:\views\sgl_test.vws
  => neon:C:\views\point_of.vws
```

For a nonshareable DO, the reference count is always **1**.

A reference count can also decrease. When a program running in any of the views that reference a shared derived object overwrites or deletes that object, the link is broken and the reference count is decremented. That is, the program deletes the view's reference to the DO, but the DO itself remains in VOB storage. This occurs most often when a compiler overwrites an old build target. You can also remove the derived object with a standard **del** command, or if the makefile has a **clean** rule, by running **clearmake clean**.

A derived object's reference count can become zero. For example, suppose you build program hello.exe and rebuild it a few minutes later. The second hello.exe overwrites the first hello.exe, decrementing its reference count. Because the reference count probably was **1** (no other view has winked it in), it now becomes **0**. Similarly, the reference counts of old DOs, even of DOs that are widely shared, eventually decrease to zero as development proceeds and new DOs replace the old ones.

The **lsdo** command ignores such DOs by default, but you can use the **–zero** option to list them:

```
Z:\vob_hw\src>cleartool lsdo –zero –long hello.obj
  .
  .
08–Mar–03.12:47:54     akp.user@cobalt
  create derived object "hello.obj@@08–Mar.12:47.259"
  references: 0
  ...
```

A derived object that is listed with a `references: 0` annotation does not currently appear in any view. However, some or all of its information may still be available:

- If the DO was ever promoted to VOB storage, its data container is still in the VOB storage pool (unless it has been scrubbed), and its CR is still in the VOB database. You can use **catcr** and **diffcr** to work with the CR. You can get to its file system data by performing a **clearmake** build in an appropriately configured view or by using the **winkin** command.
- If the DO was never promoted, its CR may be gone forever. Until the scrubber runs and deletes the data container, the **catcr** command prints the message `Config record data no longer available for DO–pname`.

Pointers on Using ClearCase Build Tools

3

This chapter presents some pointers on making best use of **clearmake** and **omake**.

Rational ClearCase includes two independent build programs, **clearmake** and **omake**. The sample build scenario that follows uses **clearmake**. However, both programs incorporate the major ClearCase build-related features, including configuration lookup, derived object sharing, and config record maintenance.

The **omake** program's strength lies primarily in its support for users who require compatibility with other build programs designed for personal computers, including Borland Make, Microsoft NMAKE, Intersolv Polymake, and OPUS Make.

For more information anout **omake**, see the following:

- **omake** reference page
- The *OMAKE Guide*

Running omake or clearmake

Typically, you run **omake** or **clearmake** from a dynamic view context using the following procedure:

1 Set a view context by assigning a drive to a dynamic view (in Windows Explorer, click **Tools > Map Network Drive** or run the **net use** command). Then change to that view:

```
C:\> net use F: \\view\myview
C:\> F:
F:\>
```

2 Change to the appropriate directory and run **omake** or **clearmake**:

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

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

or

```
F:\myvob\src> clearmake options
```

A view context prevents VOB pathnames from being dependent on the view the build occurs in. From drive **F**, you and your makefiles can access versioned objects with non-view-extended, absolute VOB pathnames like \vob2\src\main.c in either **cleartool** subcommands or standard operating system commands. If you work directly on drive M, in view-extended namespace, full pathnames to VOB objects include a view-specific prefix, which affects configuration lookup so as to prevent DO sharing between views.

A Simple clearmake Build Scenario

clearmake is designed to let developers in makefile-based build environments continue working in their accustomed manner. The following simple build scenario demonstrates how little adjustment is required to begin building with **clearmake**.

1 Go to a development directory within any VOB.

2 Edit some source files. Typically, you need to edit some sources before performing a build; accordingly, you check out some file elements and revise the checked-out versions.

3 Start a build. You can use your existing makefiles, but invoke **clearmake** instead of your standard **make** program. For example:

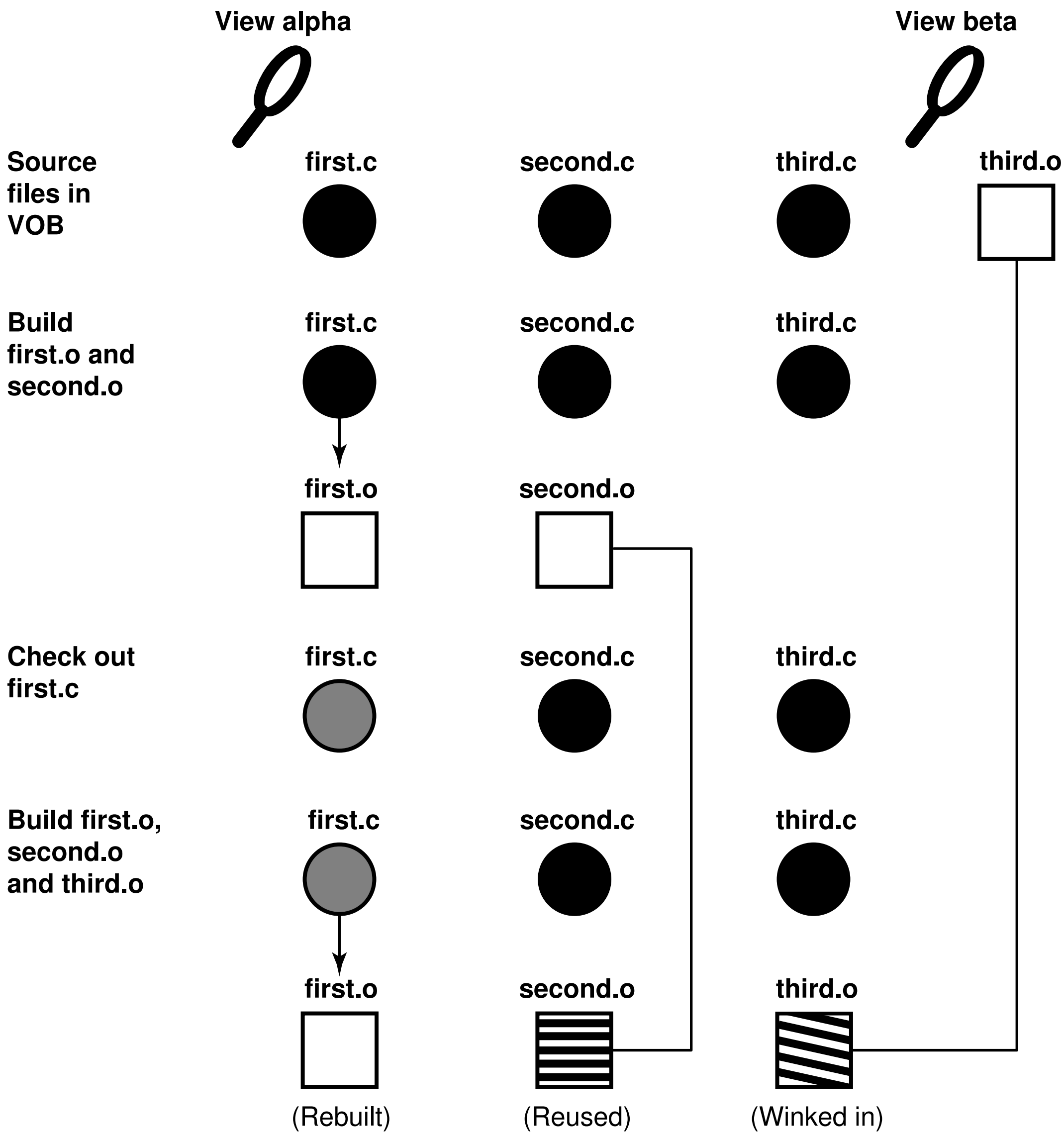
clearmake	*(build the default target)*
clearmake cwd.obj libproj.lib	*(build one or more particular targets)*
clearmake –k monet CFLAGS=–g	*(use standard options and make-macro overrides)*

(We recommend that you avoid specifying **make**-macro overrides on the command line. See *Specifying Build Options* on page 31)

clearmake builds targets (or avoids building them) in a manner similar to, but more sophisticated than, other **make** variants.

Figure 6 illustrates some typical build scenarios in which derived objects are rebuilt, reused, or winked in.

Figure 6 clearmake Build Scenario



clearmake builds new derived objects for checked-out source files, reuses derived objects for checked-in source files that have previously built the object, and winks in derived objects from other views as appropriate for checked-in source files that have not previously built the object.

Note that **clearmake** does not attempt to verify that you have actually changed the file; the checkout makes a rebuild necessary. As you work, saving a file or invoking **clearmake** causes a rebuild of the updated file's dependents, in the standard **make** manner.

For source files that you have not checked out, **clearmake** may or may not build a new derived object:

- It may reuse a derived object that appears in your view, produced by a previous build.
- It may wink in an existing derived object built in another view. (It's even possible that a winked-in DO was originally created in your view, shared, and then deleted from your view, for example, by a **make clean** command.)
- Changes to other aspects of your build environment may trigger a **clearmake** rebuild: revision to a header file; change to the build script, use of a **make**-macro override; change to an environment variable used in the build script.

Accommodating Build Avoidance

When you first begin to build software systems with Rational ClearCase, the fact that **omake** (or **clearmake**) uses a different build-avoidance algorithm than other **make** variants may occasionally surprise you. This section describes several such situations and presents simple techniques for handling them.

Increasing the Verbosity Level of a Build

If you do not understand **omake**'s (or **clearmake**'s) build-avoidance decisions, use the **–v** (somewhat verbose) or **–d** (extremely verbose) option. For **clearmake** you can also set environment variable CCASE__VERBOSITY to **1** or **2**, respectively, for equivalent behavior. When **clearmake** rebuilds a target because its build scripts do not match, the **-v** and **-d** options also return a summary of differences between the build scripts.

Handling Temporary Changes in the Build Procedure

Typically, you do not edit a target's build script in the makefile very often. But you may often change the build script by specifying overrides for **make** macros, either on the command line or in the Windows NT environment. For example, target hello.obj is specified as follows in the makefile:

```
hello.obj: hello.c hello.h
       del hello.obj
       cl /c $(CFLAGS) hello.c
```

When it executes this build script, **omake** (or **clearmake**) enters the build script, after macro substitution, into the config record. The command

Z:\myvob> omake hello.obj CFLAGS="/02 /G5"

produces this configuration record entry:

```
--------------------------------
Build script:
--------------------------------
        cl /c /02 /G5 hello.c
```

The **omake** (or **clearmake**) build-avoidance algorithm compares effective build scripts. If you then use the command **omake hello.obj** without specifying **CFLAGS="/02 /G5"**, **omake** (or **clearmake**) rejects the existing derived object, which was built with those flags. The same mismatch occurs if you create a CFLAGS environment variable with a different value, and then invoke **omake** (or **clearmake**) with the **–e** option.

Specifying Build Options

To manage temporary overrides for **make** macros and environment variables, place macro definitions in build options specification (BOS) files. **clearmake** provides several ways for using a BOS file. For example, if your makefile is named project.mk, macro definitions are read from project.mk.options. You can also keep a BOS file in your home directory or specify one or more BOS files with **clearmake –A**. For details, see *Build Options Specification Files* on page 66.

To manage these temporary overrides with **omake**, you can place macro definitions in a separate makefile. To include this makefile, specify the **–f** *makefile* option.

Using a BOS file or separate makefile to specify **make** macro overrides preserves these options, which makes it easier to reuse them. If you do not modify the BOS or separate makefile frequently, derived objects in your view are not disqualified for reuse on the basis of build script discrepancies. Some of the sections that follow describe other applications of BOS files.

Handling Targets Built in Multiple Ways

Because **omake** (and **clearmake**) compare build scripts, undesirable results may occur if your build environment includes more than one way to build a particular target. For example, suppose that the target test_prog_3 appears in two makefiles in two directories. The first is in its source directory, util_src:

```
test_prog_3.exe: ...
       cl /Fl test_prog_3.c ...
```

The second is in another directory, app_src:

```
..\util_src\test_prog_3.exe: ...
       cd ..\util_src & cl /Fl test_prog_3.c
```

Derived objects built with these scripts may be equivalent, because they are built as the same file name (test_prog_3) in the same VOB directory (util_src). But by default, a build in the app_src directory never reuses or winks in a DO built in the util_src directory, because build-script comparison fails.

You can suppress build-script comparison for this target by using an **omake** special build target, **.NOCMP_SCRIPT**, or a **clearmake** special build target, **.NO_CMP_SCRIPT** in the makefile or in an associated BOS file:

```
.NO_CMP_SCRIPT: ..\util_src\test_prog_3.exe        (clearmake build target)
.NOCMP_SCRIPT: ..\util_src\test_prog_3.exe         (omake build target)
```

To suspend build-script comparison once, you can use either **omake –O** or **clearmake –O**.

Using a Recursive Invocation of omake or clearmake

You can eliminate the problem of different build scripts described in *Handling Targets Built in Multiple Ways* by adding a recursive invocation of **clearmake** to the makefile in app_src:

```
..\util_src\test_prog_3.exe: ...
     cd ..\util_src & $(MAKE) test_prog_3.exe   ($(MAKE) invokes omake or
                                                 clearmake recursively)
```

Now, target test_prog_3 is built the same way in both directories. You can turn on build-script comparison again, by removing the **.NOCMP_SCRIPT** or **.NO_CMP_SCRIPT** special target.

Optimizing Winkin by Avoiding Pseudotargets

Like other **make** variants, **omake** (or **clearmake**) always executes the build script for a pseudotarget, a target that does not name a file system object built by the script. For example, in the section *Using a Recursive Invocation of omake or clearmake*, you may be tempted to use a pseudotarget in the app_src directory's makefile:

```
test_prog_3.exe: ...                  (shortened from ..\util_src\test_prog_3)
       cd ..\util_src & $(MAKE) test_prog_3.exe
```

A build of any higher-level target that has test_prog_3.exe as a build dependency always builds a new test_prog_3.exe, which in turn triggers a rebuild of the higher-level target. If the rebuild of test_prog_3.exe was not necessary, the rebuild of the higher-level target may not have been necessary, either. Such unnecessary rebuilds decrease the extent to which you can take advantage of derived object sharing.

Accommodating the Build Tool's Different Name

The fact that the ClearCase build utility has a unique name, **omake** (or **clearmake**), may conflict with existing build procedures that implement recursive builds. Most make variants automatically define the **make** macro **$(MAKE)** as the name of the build program, as it was typed on the command line:

`Z:\avob>`	**make hello.obj**	*(sets MAKE to "make")*
`Z:\avob>`	**omake hello.obj**	*(sets MAKE to "omake")*
`Z:\avob>`	**clearmake hello.obj**	*(sets MAKE to "clearmake")*
`Z:\avob>`	**my_make hello.obj**	*(sets MAKE to "my_make")*

This definition enables recursive builds to use **$(MAKE)** to invoke the same build program at each level. The section *Optimizing Winkin by Avoiding Pseudotargets* includes one such example; here is another one:

```
SUBDIRS = lib util src

all:
       for %DIR in ($(SUBDIRS)) do cd %DIR & $(MAKE) all
```

Executing this build script with **omake** (or **clearmake**) invokes **omake all** (or **clearmake all**) recursively in each subdirectory.

Declaring Source Dependencies in Makefiles

To implement build avoidance based on time stamps, standard **make** variants require you to declare all the source file dependencies of each build target. For example, object module hello.obj depends on source files hello.c and hello.h in the same directory:

```
hello.obj: hello.c hello.h
      del  hello.obj
      cl /c hello.c
```

Typically, these source files depend on project-specific header files through **`#include`** directives, perhaps nested within one another. The standard files do not change very often, but programmers often lament that "it didn't compile because someone changed the project's header files without telling me."

To alleviate this problem, some organizations include every header file dependency in their makefiles. They rely on utility programs to read the source files and determine the dependencies.

omake and **clearmake** do not require that source-file dependencies be declared in makefiles (but see *Source Dependencies Declared Explicitly* on page 34). The first time a derived object is built, its build script is always executed; thus, the dependency declarations are irrelevant for determining whether the target is out of date. After a derived object has been built, its configuration record provides a complete list of source-file dependencies used in the previous build, including those on all header files (nested and nonnested) read during the build.

Note: **clearmake** will rebuild a target if there have been changes to any directories listed as source-file dependencies.

You can leave source-file dependency declarations in your existing makefiles, but you need not update them as you revise the makefiles. And you need not place source-file dependencies in new makefiles to be used with **omake** or **clearmake**.

Note: Although source-file dependency declarations are not required, you may want to include them in your makefiles, anyway. The principal reason for doing so is portability: you may need to provide your sources to another team (or another company) that is not using ClearCase.

Source Dependencies Declared Explicitly

The ClearCase build auditing facility tracks only the MVFS objects used to build a target. Sometimes, however, you may want to track other objects. For example:

- The version of a compiler that is not stored in a VOB
- The version of the operating system kernel, which is not referenced at all during the build
- The state of a flag-file, used to force rebuilds

You can force such objects to be recorded in the CR by declaring them as dependencies of the makefile target:

```
hello.obj: hello.c hello.h  my.flag
      del hello.obj
      cl /c hello.c
```

This example illustrates dependency declarations for these kinds of objects:

- (hello.c, hello.h) - Dependencies on MVFS objects are optional. These are recorded by clearmake and MVFS anyway.
- my.flag - Dependencies on view-private objects can implement a flag-file capability.

Explicit Dependencies on Searched-For Sources

There are situations in which the configuration lookup algorithm that **clearmake** and **omake** use qualifies a derived object, even though rebuilding the target would produce a different result. Configuration lookup requires that for each object listed in an existing CR, the current view must select the same version of that object. However, when search paths must be used to find an object, a target rebuild may use a different object than the one listed in the CR. Configuration lookup does not take this possibility into account.

When files are accessed by explicit pathnames, configuration lookup qualifies derived objects correctly. Configuration lookup may qualify a derived object incorrectly if files are accessed at build time by a search through multiple directories, for example, when the **–I** option to a C or C++ compiler specifies a header file or when the **–L** option to a linker specifies a library file. The following build script uses a search to locate a header file, fio.h:

```
hello.obj:
       cl /c /I \projvob\privh /I \projvob\stdh hello.c
```

The command **clearmake hello.obj** may qualify an existing derived object built with C:\projvob\privh\fio.h, even though rebuilding the target would now use C:\projvob\stdh\fio.h instead.

omake and **clearmake** address this problem in the same way as some standard **make** implementations:

- You must declare the searched-for source object as an explicit dependency in the makefile:

  ```
  hello.obj: fio.h
  ...
  ```

- You must use the **VPATH** macro to specify the set of directories to be searched:

  ```
  VPATH = \projvob\privh;projvob\stdh
  ```

Given this makefile, **omake** (or **clearmake**) uses the **VPATH** (if any) when it performs configuration lookup on fio.h. If a candidate derived object was built with C:\projvob\privh\fio.h, but would be built with C:\projvob\stdh\fio.h in the current view, the candidate is rejected.

Note: The **VPATH** macro is not used for all source dependencies listed in the config record. It is used only for explicitly declared dependencies of the target. Also, **clearmake** searches only in the current view.

Build Tool Dependencies. You can use this mechanism to implement dependencies on build tools. For example, you can track the version of the C compiler used in a build as follows:

```
msg.obj: msg.c $(CC)
       $(CC) /c msg.c
```

With this makefile, either your **VPATH** must include the directories on your search path, or you must use a full pathname as the **$(CC)** value.

Note: If your C compiler is stored in a VOB and you invoke it from the VOB, ClearCase tracks its version and you do not have to include it as a dependency.

Build-Order Dependencies

In addition to source dependencies, makefiles also contain build-order dependencies. For example:

```
hello.exe: hello.obj libhello.lib
      ...
libhello.lib: hello_env.obj hello_time.obj
      ...
```

These dependencies are buildable objects known as *subtargets*. The executable hello.exe must be built after its subtargets, object module hello.obj and library libhello.lib, and the library must be built after its subtargets, object modules hello_env.obj and hello_time.obj.

ClearCase does not detect build-order dependencies; you must include such dependencies in makefiles used with **omake** (or **clearmake**), as you do with other **make** variants.

clearmake Build Script Execution and cmd.exe

By default, **clearmake** invokes cmd.exe to execute build scripts. It finds cmd.exe in %SYSTEMROOT%\system32; if SYSTEMROOT is not set, **clearmake** uses C:\winnt as the default system root directory. If it cannot find the shell, **clearmake** exits.

You can override the default shell by using a **SHELL** make macro (note that you must specify a full pathname and include the file name extension). The following anomalies may occur if you use a different shell or if you execute a batch file from a build script:

- **clearmake** may not recognize failed commands and continue the build with unpredictable results.
- **clearmake** may display an extra command prompt after a build script command completes successfully.

Note: If **clearmake** determines that it can execute the build script directly, it does not use the shell program even if you specify one explicitly. If you use .bat files in build scripts, you must make them executable (use the **cleartool protect** command). To force **clearmake** to use the shell program, set the environment variable CCASE_SHELL_REQUIRED.

Build Scripts and the rm Command

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

Pathnames in CRs

In a config record created on Windows, 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-03.17:49:53>
\proj1\lib\fsutl.h@@\main\12                 <22-Jun-03.12:07:24>
...
```

Pathnames in this format can be cut or 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).

Problems with Forced Builds

omake has an unconditional option **–a** (on **clearmake** it is **–u**), which forces rebuilds. Using this option reduces the efficiency of derived object sharing, however. If you force **clearmake** or **omake** to build a target when it would have winked in an existing DO, you create a new DO with the same configuration as an existing one. In such situations, a developer who expects a build to share a particular existing DO may get another, identically configured DO instead. This may confuse the team and waste disk space.

We suggest that you use a flag file to force a rebuild, rather than using **omake –a** or **clearmake –u**. (See *Source Dependencies Declared Explicitly* on page 34.)

How clearmake Interprets Double-Colon Rules

Double-colon rules are a special kind of makefile construct that allows several independent rules for one target, each with a possibly different build script. The semantics given to these rules by other **make** programs (and for **clearmake** when CRs are not being generated) are that commands within each double-colon rule are executed if the target is older than any dependencies of that particular rule. The result can be that none, any, or all of the double-colon rules are executed.

However, when **clearmake** creates CRs and associates them with the results of its builds, this interpretation runs the risk of generating incomplete CRs, which do not contain all the versions and build scripts used to build the targets. For this reason, **clearmake** interprets these rules in a more conservative way.

When building a target specified by a number of double-colon rules, **clearmake** concatenates all build scripts from all the double-colon rules for that target and runs them in a single audited script.

To produce the correct results, any subtargets must already have been built, so **clearmake** builds any out-of-date subtargets before it executes the concatenated build script.

As a result, you may observe these differences in behavior between **clearmake** and other **make** programs concerning double-colon rules:

- **clearmake** runs more of the build scripts than other **make** programs.
- **clearmake** may run the build scripts in a different order than other **make** programs.

However, given the intended use and standard interpretation of double-colon rules, these differences still produce correct builds and complete correct CRs.

Continuing to Work During a Build

As your build progresses, other developers continue to work on their files and may check in new versions of elements that your build uses. If your build takes an hour to complete, you do not want build scripts executed early in the build to use version 6 of a header file and scripts executed later to use version 7 or 8.

To prevent such inconsistencies, any version whose selection is based on a **LATEST** config spec rule is locked out if it is checked in after the instant that **omake** (or **clearmake**) was invoked. The moment that the **omake** (or **clearmake**) build session begins is the build reference time.

The same reference time is reported in each configuration record produced during the build session, even if the session lasts hours (or days):

```
Z:\avob> cleartool catcr hello.obj
Target hello.obj built by drp.dvt
Host "fermi" running  Windows NT 3.5 (807)
Reference Time 26-Feb-03.16:53:58, this audit started
26-Feb-03.16:54:10 ...
```

Note: The `reference time` is the build reference time, when the overall **omake** (or **clearmake**) build session began. The `this audit started` time is when the execution of the individual build script began.

When determining whether an object was created before or after the build reference time, **omake** (or **clearmake**) adjusts for clock skew, the inevitable small differences among the system clocks on different hosts. For more information about build sessions, see *Build Sessions, Subsessions, and Hierarchical Builds* on page 40.

Caution: A build's coordinated reference time applies to elements only, providing protection from changes made after the build began. You are not protected from changes to view-private objects and non-MVFS objects. For example, if you begin a build and then change a checked-out file used in the build, a failure may result. To avoid this problem, do not work on the same project in a view in which a build is in progress.

Using Config Spec Time Rules

Note: If you use a UCM view, your config spec is generated by ClearCase. Do not add time rules to your config spec.

Using the reference time facility described in *Continuing to Work During a Build*, **omake** (or **clearmake**) blocks out potentially incompatible source-level changes that take place after your build begins. For example, if a rule contains a remote branch type and a synchronization takes place during a build, **omake** (or **clearmake**) uses the latest version of the remote files that were checked in before the time lock. If, however, an incompatible change has already taken place, ClearCase allows you to block out recently created versions.

A typical ClearCase development strategy is for each team member to work in a separate view, but to have all the views use the same config spec. In this way, the entire team works on the same branch. As long as a source file remains checked out, its changes are isolated to a single view; when a developer checks in a new version, the entire team sees it on the dedicated branch.

This incremental integration strategy is often very effective. But suppose that another user's recently checked-in version causes your builds to start failing. Through an exchange of e-mail, you trace the problem to header file project_base.h, checked in at 11:18 A.M. today. You and other team members can reconfigure your views to roll back that one element to a safe version:

```
element project_base.h ...\onyx_port\LATEST -time 5-Mar.11:00
```

If many interdependent files have been revised, you can roll back the view for all checked-in elements:

```
element * ...\onyx_port\LATEST -time 5-Mar.11:00
```

For a complete description of time rules, see the **config_spec** reference page.

Inappropriate Use of Time Rules

Your view interprets time rules with respect to the `create version` event record written by the **checkin** command. The checkin is read from the system clock on the VOB server host. If that clock is out of sync with the clock on the view server host, your attempt to roll back the clock may fail. Thus, do not strive for extreme precision with time rules: select a time that is well before the actual cutoff time (for example, a full hour before, or in the middle of the night).

Do not use time rules to freeze a view to the current time immediately before you start a build. Allow **omake**'s (or **clearmake**'s) reference time facility to perform this service. Here's an inappropriate use scenario:

1. You check in version 12 of util.c at 7:05 P.M. on your host. You do not know that clock skew on the VOB host causes the time 7:23 P.M. to be entered in the `create version` event record.
2. To freeze your view, you change your config spec to include this rule:

   ```
   element * \main\LATEST -time 19:05
   ```

3. You issue an **omake** (or a **clearmake**) command immediately (at 7:06 P.M.) to build a program that uses util.c. When selecting a version of this element to use in the build, your view consults the event history of util.c and rejects version 12, because the 7:23 P.M. time stamp is too late for the **–time** configuration rule.

Build Sessions, Subsessions, and Hierarchical Builds

The following terms are used to describe the details of ClearCase build auditing:

- Invoking **clearmake**, **omake**, or **clearaudit** starts a build session. The time at which the build session begins becomes the build reference time for the entire build session, as described on *Continuing to Work During a Build* on page 38.
- During a build session, one or more target rebuilds typically take place.
- Each target rebuild involves the execution of one or more build scripts. (A double-colon target can have multiple build scripts; see *How clearmake Interprets Double-Colon Rules* on page 37.)
- During each target rebuild, **clearmake**, **omake**, or **clearaudit** conducts a build audit.

Subsessions

A build session can have any number of subsessions, all of which inherit the reference time of the build session. A subsession corresponds to a nested build or recursive make, which is started when a **clearmake**, **omake**, or **clearaudit** process is invoked in the process family of a higher-level **clearmake**, **omake**, or **clearaudit**. For example:

- Including a **clearmake**, **omake**, or **clearaudit** command in a makefile build script executed by **clearmake**, **omake**, or **clearaudit**
- Entering a **clearmake**, **omake**, or **clearaudit** command in an interactive process started by **clearaudit**

A subsession begins while a higher-level session is still conducting build audits. The subsession conducts its own build audits, independent of the audits of the higher-level session; that is, the audits are not nested or related in any way, other than that they share the same build reference time.

Versions Created During a Build Session

Any version created during a build session and selected by a **LATEST** config spec rule is not visible in that build session. For example, a build checks in a derived object that it has created; subsequent commands in the same build session do not select the checked-in version, unless it is selected by a config spec rule that does not use the version label **LATEST**.

An effect of this behavior is that you cannot check in and label a version during a single build session. Instead, you must check in the version during one build session, and label it during another build session. Use the **mklabel –config** command to label versions associated with a specific derived object.

Coordinating Reference Times of Several Builds

Different build sessions have different reference times. The best way to assign a series of builds the same reference time is to structure them as a single, hierarchical build.

An alternative approach is to run all the builds within the same **clearaudit** session. For example, you can write a batch file, **multi_make.bat**, that includes several invocations of **clearmake**, **omake**, or **clearaudit** (along with other commands). Running the script as follows ensures that all the builds are subsessions that share the same reference time:

clearaudit –c multi_make.bat

Objects Written at More Than One Level

Problems occur when the same file is written at two or more session levels (for example, a top-level build session and a subsession): the build audit for the higher-level session does not contain complete information about the file system operations that affected the file. For example:

clearaudit /c "clearmake shuffle > logfile"

The file logfile may be written twice:

- During the **clearaudit** build session, by the command invoked from **clearaudit**
- During the **clearmake** subsession, when the **clearaudit** build session is suspended

In this case, **clearaudit** issues this error message:

```
clearaudit: Error: Derived object modified; cannot be stored in VOB.
Interference from another process?
```

To work around this limitation, postprocess the derived object at the higher level with a **copy** command:

clearaudit /c "clearmake shuffle > log.tmp& copy log.tmp logfile& del log.tmp"

Build Auditing and Background Processes

The ClearCase build programs—**omake**, **clearmake**, and **clearaudit**—use the same procedure to produce configuration records:

1. Send a request to the host's multiversion file system (MVFS), to initiate build auditing.
2. Start one or more child processes (typically, commands), in which makefile build scripts or other commands are executed.
3. Turn off MVFS build auditing.
4. If all the subprocesses have indicated success and at least one MVFS file has been created, compute and store one or more configuration records.

Any subprocesses of the child processes started in Step 2 inherit the same MVFS build audit. (Recursive invocations of ClearCase build programs conduct their own, independent audits; see *Build Sessions, Subsessions, and Hierarchical Builds* on page 40.)

A problem can occur if a build script (or other audited command) invokes a background subprocess and exits without waiting for it to complete. The build program has no knowledge of the background process and may proceed to Step 3 and Step 4 before the background process has finished its work. In such situations, ClearCase cannot guarantee what portion, if any, of the actions of background commands will be included in the resulting CR. The contents of the CR depend on system scheduling and timing behavior.

The ClearCase build programs audit background processes correctly only if both of the following conditions are true:

- The build script does not complete until all background processes are known to have finished.
- Each background process performs its first MVFS file access while it is still a descendant process of the **clearmake**, **omake**, or **clearaudit** process. (The ClearCase kernel component determines whether to audit a given process when that process first accesses the MVFS. If the process's ancestors include a process already being audited, the descendant process is similarly marked for auditing.)

If either or both of these conditions are false, avoid using background processes in audited build scripts.

Working with Incremental Update Tools

The design of the build auditing capability makes it ideal for use with tools that build derived objects from scratch. Because newly created objects have no history, ClearCase can learn everything it needs to know at build time. But this reliance on auditing at the files system level at build time can cause ClearCase to record incomplete information for objects that are updated incrementally, which do have a history.

In ClearCase, incremental updating means that an object is updated partially during the builds of multiple makefile targets, instead of generated completely by the build of one target. By default, **omake** and **clearmake** do not update an existing CR incrementally when they build a target. Instead, they do the following:

- Each time a build script incrementally updates an object's file system data, **omake** and **clearmake** write a completely new CR, which describes only the most recent update, not the entire build history.
- The new CR does not match the desired build configuration for any of the other targets that update the object incrementally.

The result is a situation that is both unstable and incorrect: all incremental-update targets are rebuilt each time that **omake** (or **clearmake**) is invoked; when the build is finished, the DO has the correct file system data, but its CR may not describe the DO's configuration accurately.

omake and **clearmake** provide a special makefile target **.INCREMENTAL_TARGET**, which can be used to guarantee correct CR information for incremental updates. The following sections give examples of how to use **.INCREMENTAL_TARGET**.

Example: Incremental Linking

If your makefile is structured properly, configuration records are not likely to lose information during incremental links.

Incremental linkers typically work by determining which object files have changed since the last link and relinking only those objects. Because the linker may not read every object each time it links, a CR can, in theory, lose information as repeated links are made. But in practice, because all dependencies of the link are listed in the build script, the build script does not change from one link invocation to the next. And, because you typically list the objects or predefined dependencies of the link, those dependencies are included in the CR.

Additional Incremental-Update Situations

You may encounter incremental updating in other situations, as well. For example, Visual C++ supports a program database file (PDB) that contains debugging information and is updated incrementally as different targets are built. ClearCase includes special makefile rules to work around problems associated with incremental files produced by Visual C++. For more information, see Chapter 7, *Using ClearCase to Build C++ Programs*.

Temporary Build Audit Files

By default, **omake** and **clearmake** generate temporary build audit files in the directory identified by the TMP environment variable. You can set the CCASE_AUDIT_TMPDIR environment variable to relocate these files. If you do not set this variable, verify that TMP is set to a valid temporary storage directory on a FAT, NTFS, or LAN Manager file system.

Auditing 16-bit Tools

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

The program **vdmaudit** allows auditing of 16-bit tools and does not affect programs run outside a build. To use **vdmaudit**, you need to have **clearmake** or **omake** run **vdmaudit** and let it call the tool. This involves either editing the **makefile** where it calls the tool or if your **makefile** defines a macro for each tool, redefining the macros in the **makefile** or on the **clearmake** or **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 such as

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 **clearmake** or **omake** command line:

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

Call all 16-bit tools from **vdmaudit**. If you do not, **clearmake** and **omake** do 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.

Adding a Version String or Time Stamp to an Executable

This section describes a simple technique for incorporating a version string and/or time stamp into a C-language compiled executable. Including a version string or time stamp allows anyone (for example, a customer) to determine the exact version of a program by entering a shell command.

The technique involves adding a "what version?" command-line option to the executable itself:

```
Z:\> monet –Ver
monet R2.0 Baselevel 1 (Thu Feb 11 17:33:23 EST 2003)
```

After the particular version of the program is determined, you can use ClearCase commands to find a local copy, examine its config record, and if appropriate, reconstruct the source configuration with which it was built. (Presumably, the local copy is a derived object that has been checked in as a version of an element.)

You can identify the appropriate derived object by attaching a ClearCase attribute with the version string to the checked-in executable.

Implementing a –Ver Option

You can write a program to display the information stored in the **version_string** and **version_time** variables. An example of such a program follows:

```
#include <stdio.h>

main(argc,argv)
    int argc;
    char **argv;
{
/*
 * implement -Ver option
 */
    if (argc > 1 && strcmp(argv[1],"-Ver") == 0) {
        char *version_string = "monet R2.0 Baselevel 1";
        char *version_time= "Thu Feb 11 17:33:23 EST 2003";
        /*
         * Print version info
         */
        printf ("%s (%s)\n",
                    version_string, version_time);
        exit(0);
    }
}
```

Working with DOs and Configuration Records

4

This chapter describes the operations you can perform on derived objects and configuration records. For more information and examples, see the reference pages for the commands.

The information in this chapter applies only to dynamic views.

Setting Correct Permissions for Derived Objects

If you and other members of your team want to share derived objects (DOs), make sure that your views are configured to create shareable DOs and that the DOs are created with a mode that grants both read and write access to team members.

Permissions on DOs affect the extent to which they are shareable:

- When you perform a build, the ClearCase build tool winks in a derived object to your view only if you have read permission on the DO.
- The ClearCase build tool can wink in DOs for which you do not have write permission. But `permission denied` errors may occur during a subsequent build, when a compiler (or other build script command) attempts to overwrite such a DO. To work around this problem, you can rewrite your makefile to remove the target before rebuilding it. You can also set a policy for how users must set their permissions.

For information about fixing the permissions of DO versions, see the **protect** reference page.

Listing and Describing Derived Objects

The following sections describe how to use the **lsdo**, **describe**, **ls**, and **lsprivate** commands to list derived objects.

Listing Derived Objects Created at a Certain Pathname

Use the **lsdo** command to list derived objects created at a specific pathname. For information about the kinds of DOs included in the listing, see the **lsdo** reference page.

- To list all DOs created at the pathname adm.h:

  cleartool lsdo adm.h

  ```
  01-Jul.13:49 "adm.h@@01-Jul.13:49.1286781"
  30-Jun.20:03 "adm.h@@30-Jun.20:03.1278990"
  30-Jun.18:14 "adm.h@@30-Jun.18:14.1277470"
  29-Jun.19:11 "adm.h@@29-Jun.19:11.1253509"
  29-Jun.18:13 "adm.h@@29-Jun.18:13.1252790"
  29-Jun.16:09 "adm.h@@29-Jun.16:09.1249897"
  ```

- To list all DOs created by you at the pathname adm.h:

  cleartool lsdo –me adm.h

  ```
  30-Jun.18:14 "adm.h@@30-Jun.18:14.1277470"
  ```

Listing a Derived Object's Kind

To display a derived object's kind, use the **cleartool** commands **ls –l**, **lsprivate –l –do**, or **describe –fmt "%[DO_kind]p"**. The kind can be one of the following values:

nonshareable	The DO was created during an express build and cannot be winked in by other views.
unshared	The DO was created during a regular build. Its data container is located in view storage, not in the VOB.
promoted	The DO's data container has been promoted to the VOB by a **winkin** or **view_scrubber –p** command. The DO is referenced by only one view.
shared	The DO's data container has been promoted to the VOB by a ClearCase build tool or a manual **winkin** or **view_scrubber –p** command.

- List, in long form, a particular DO.

  cleartool ls –l util

  ```
  derived object (non-shareable)  util@@01-Sep.10:54.2147483681
  ```

- To list all DOs created in the current view in the **\dev** VOB, including the DO kind:

  cleartool lsprivate –long –invob \dev –do

  ```
  derived object (unshared)  \dev\file2.txt@@02-Jul.13:51.124
  derived object (unshared)  \dev\file2sub.txt@@02-Jul.13:51.123
  ```

- To list the name and kind of all DOs created in the current view:

cleartool lsprivate –long –do

```
derived object (shared)  \dev\file2.txt@@02-Jul.13:51.124
derived object (unshared)  \dev\file2sub.txt@@02-Jul.13:51.123
derived object (unshared)  \dev\dir1\x.obj@@01-Jul.14:23.186
derived object (unshared)  \dev\api\bin\adm.exe@@04-Jul.04:01.776
```

Displaying a DO's OID

A derived object's OID is the permanent identifier recorded in the VOB database for the DO. It does not change over the life of the DO, unlike the DO-ID (see *Reuse of DO IDs* on page 23). To display the OID, use the command **describe –fmt "%On"**. For example:

cleartool describe –fmt "%On\n" x.o

```
b7afc83e.2f2311d3.a382.00:01:80:7b:09:69
```

Displaying a Description of a DO Version

The **describe** command displays descriptions of DO versions, as it does descriptions of regular versions. You can use the **–fmt** option to extract parts of the description. For example, the following command prints the name, predecessor version, and element type of a DO version:

cleartool describe –fmt "%n\t%[version_predecessor]p\t%[type]p\n" file1.obj

```
file1.obj@@\main\2     \main\1  text_file
```

For more information about the **–fmt** option, see the **fmt_ccase** reference page.

Identifying the Views That Reference a Derived Object

The VOB stores information about which views reference a derived object. To display this information, use the **lsdo** command.

Caching Unavailable Views

When **clearmake** shops for a derived object to wink in to a build, it may find DOs from a view that is unavailable (because the view server host is down, the **albd_server** is not running on the server host, and so on). Attempting to fetch the DO's configuration record from an unavailable view causes a long time-out, and the build may reference multiple DOs from the same view.

clearmake and other **cleartool** commands that access configuration records and DOs (**lsdo**, **describe**, **catcr**, **diffcr**) maintain a cache of tags of inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

Note: The cache is not persistent across **clearmake** sessions. Each recursive or individual invocation of **clearmake** attempts to contact a view whose tag may have been cached in a previous invocation.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

Specifying Views That Can Wink In Derived Objects

You can use the CCASE_WINKIN_VIEWS environment variable to specify a list of views that can wink in derived objects. If this variable is set in the environment or in the makefile, **clearmake** winks in only derived objects that were built in the specified views. If no derived objects are available, **clearmake** rebuilds in the current view.

```
Z:\vob_hw\src> cleartool lsdo –l hello.obj
10-Mar-03.15:25:52 akp.user@copper
  create derived object "hello.obj@@10-Mar.15:25.213"
  size of derived object is: 450
  last access: 15-Mar-99.14:22:17
  references: 2 (shared)
  => copper:C:\views\akp\tut\old.vws
  => copper:C:\views\akp\tut\fix.vws
```

Specifying a Derived Object in Commands

In general, you use standard pathnames to access DOs when you're working in a view that references them. To standard software (for example, linkers and debuggers), the standard pathname of a derived object (util.o) references the DO.

This is another example of ClearCase transparency: a standard pathname accesses one of many different variants of a file system object. Note this distinction, however:

- A version of an element appears in a dynamic view because it is selected by a configuration specification rule.
- A particular derived object appears in a dynamic view as the result of a build or a winkin.

To access a DO in another dynamic view, use a view-extended pathname:

```
M:\drp\vob_proj\src\msg.obj            (the DO in view drp)

M:\R2_integ\vob_proj\src\msg.obj       (the DO in view R2_integ)
```

Note: You cannot use view-extended pathnames in makefiles.

To specify a certain DO in a ClearCase command, use the DO-ID. For example, you can use a DO-ID in the **catcr** command to view the contents of a specific DO's config record:

cleartool catcr x.o@@29-Jun.14:40.88

Note: You cannot use a DO-ID in standard commands.

Because DO-IDs can change, avoid using them in files or scripts that operate on a DO. Instead, use a standard pathname or the derived object's object identifier (OID), which never changes. To determine a DO's object identifier, use **cleartool describe –fmt "%On\n"**. For example:

cleartool describe –fmt "%On\n" x.o@@29-Jun.14:40.88

```
2c5fc68a.2e5311d3.a382.00:01:80:7b:09:69
```

Also, a derived object is assigned a permanent identifier when it is checked in as a version of an element. See *Working with DO Versions* on page 54.

Winking In a DO Manually

You can manually wink in any DO to your view, using the **winkin** command. For example:

```
Z:\vob_hw> cleartool lsdo hello.exe
08-Mar.12:48   akp        "hello.exe@@08-Mar.12:48.265"
07-Mar.11:40   george     "hello.exe@@07-Mar.11:40.217"

Z:\vob_hw> cleartool winkin hello.exe@@07-Mar.11:40.217
Winked in derived object "hello.exe"

Z:\vob_hw> hello
...
```

You can wink in a DO that does not match your build configuration for any of the following reasons: to run it, to perform a byte-by-byte comparison with another DO, or to perform any other operation that requires access to the DO's file system data.

The **winkin** command can also wink in the set of DOs in a hierarchy of CRs. You can use this recursive winkin to seed a new view with a set of derived objects. For example:

cleartool winkin -recurse x@@20-Jul.14:32.146

```
Winked in derived object "\smg_test\file2.txt"
Winked in derived object "\smg_test\file2sub.txt"
Promoting unshared derived object "\smg_test\x".
Winked in derived object "\smg_test\x"
```

You can use the **winkin** command to convert a nonshareable DO to a shared DO. For more information, see *Converting Nonshareable DOs to Shared DOs* on page 59.

Preventing Winkin

The following sections describe how to prevent winkin to or from your view during a build or audit. (You can prevent winkins altogether by building in a snapshot view or by not using the ClearCase build tools.)

Preventing Winkin to Your View

To direct **clearmake** or **omake** to limit reuse to DOs created in the current view, use **clearmake –V** or **omake –W**. For more information, see the **clearmake** or **omake** reference page.

Preventing Winkin to Other Views

To prevent any derived objects that you create from being winked in to other views, use one of the following techniques:

- Use express builds. See *Using Express Builds to Prevent Winkin to Other Views*.
- Use the **–T** or **–F** options with **clearmake** or the **–L** option with **omake** to create view-private files with no configuration records. **clearmake** and **omake** do not perform configuration lookup, but this does not matter if you are not changing other files.
- Use special targets that prevent winkin. For example, use **.NO_WINK_IN** with **clearmake**. For more information, see *Special Targets* on page 73.

Using Express Builds to Prevent Winkin to Other Views

During an express build, Rational ClearCase creates DOs that are nonshareable and cannot be used by builds in other views. These nonshareable DOs have configuration records, but the ClearCase build tools do not write information into the VOB for these DOs. Therefore, the DOs are invisible to builds in other views.

Note: During an express build, the ClearCase build tools wink in DOs from other views. For information about avoiding winkins from other views, see *Preventing Winkin to Your View* on page 52.

Use express builds when DOs created by the build are not appropriate for use by other views. As a general rule:

- Use express builds for development builds that use relatively unstable, checked-out versions.
- Use regular builds for release or nightly builds that use stable, checked-in versions. DOs created by these builds are more likely to be winked in by other views.

Enabling Express Builds

When you invoke a ClearCase build tool, the kind of build that occurs depends on how your view is configured. To use express builds, configure an existing dynamic view with the nonshareable DOs property or create a new dynamic view with the nonshareable DOs property. Then, run your ClearCase build tool (**clearmake**, **omake**, or **clearaudit**) in the view.

The following sections describe how to configure your view to use express builds.

Configuring an Existing View for Express Builds

Use the command **chview –nshareable_dos** *view-tag*. For more information, see the **chview** reference page.

Future builds in the view will create nonshareable derived objects. However, existing DOs in the view are shareable; they are not converted to nonshareable. These existing DOs can still be winked in by other views.

Creating a New View That Uses Express Builds

To create a new view, use one of the following methods:

- Use the **mkview** command with the **–nshareable_dos** option. For more information, see the **mkview** reference page.
- Use the View Creation Wizard:
  - **a** In Step 3, click **Advanced Options**.
  - **b** Clear the **Create shareable derived objects** check box.
  - **c** Follow the prompts of the View Creation Wizard.

Note: The ClearCase administrator can set a site-wide default for the DO property. If a site-wide default exists and you do not specify the DO property when you create the view, ClearCase uses the site-wide default. For more information, see the **setsite** reference page.

Preventing Winkin to or from Other Architectures

By default, the build tool winks in derived objects built on different architectures. For example, a build on a Sun host may wink in a DO built on a Windows NT host. If you want to prevent this behavior, use one of the following techniques:

- Differentiate the build script for different architectures:
  - Add the architecture name to your build script so that the build tool differentiates among the build scripts.
  - Store the architecture name in a macro and pass it to the build tool on the command line.
  - Use architecture-specific subdirectories to store DOs.
- Make your tools a build dependency by storing them in a VOB. Also, store your system header files in a VOB.

Converting Derived Objects to View-Private Files

Using a standard command or program to modify a derived object in any way converts it from a DO to a view-private file. For example, use **ls –long** to list a derived object:

cleartool ls –long msg.obj
```
derived object (shared)     msg.obj@@10-Mar.15:33.333
```

Modify the DO with a standard command:

echo "" >>msg.obj

The **ls –long** command now lists the file as a view-private object:

cleartool ls –long msg.obj
```
view private object     msg.obj
```

Working with DO Versions

The following sections describe how to create and manipulate DO versions.

Creating DO Versions

You can convert DOs to elements or check them in as versions of existing elements. The element-creation and version-creation processes are the same for shareable and nonshareable DOs, with this exception: when you check in a nonshareable DO, it is converted to a shared DO before being checked in.

For more information, see the **mkelem** and **checkin** reference pages.

Checking In DOs During a Build

You can write a build script that creates a derived object and then checks it in or converts it to an element. However, the ClearCase build tool does not create a configuration record until the build script has completed (all commands after the *target-name*: have executed). Therefore, if the same build script that created the DO checks it in or converts it to an element, the resulting version is not a DO version.

For example, the version created by the following build script is not a DO version:

```
buildit : buildit.c
       cleartool co -unres -nc $@
       del /F $@
       cl /c $@ $*.c
       cleartool ci -nc buildit
```

You can work around this problem by building and checking in a derived object in two steps. For example, the makefile contains one build script that creates the DO, and another build script that checks it in, as shown here:

```
buildit : buildit.c
       cleartool co -unres -nc $@
       del /F $@
       cl /c $@ $*.c

stageit : buildit
       cleartool ci -nc buildit
```

The command **clearmake stageit** performs the following steps:

1. Brings the target buildit up to date. This creates a DO named buildit and an associated configuration record.
2. Brings the target stageit up to date. This step checks in the buildit derived object as a DO version.

Accessing DO Versions

When you check out a DO version, it is winked in to your dynamic view. You can use a standard pathname to access the DO's file system data. However, VOB-database access is handled in the following ways:

- A standard pathname to the DO references the version in the VOB database from which the checkout was made:

```
Z:\myvob\bin> cleartool checkout -nc hello          (wink in derived object hello)
Checked out "hello" from version "\main\3".


Z:\myvob\bin> cleartool mklabel EXPER hello         (use standard pathname to
Created label "EXPER" on "hello" version            access version from which
"\main\3".                                          checkout was made)
```

- To access the checked-out placeholder version, you must use an extended pathname:

```
Z:\myvob\bin> cleartool mklabel -replace EXPER
hello@@\main\CHECKEDOUT
Moved label "EXPER" on "hello" from version "\main\3" to
"\main\CHECKEDOUT".
```

If you process a checked-out DO version, as described in *Converting Derived Objects to View-Private Files* on page 54, ClearCase reverts to its usual handling of checked-out versions. In this case, a standard pathname references the placeholder version in the VOB database.

Displaying Configuration Records for DO Versions

The **catcr** command displays the configuration record for a DO version. When you use **catcr –recurse** to display the CRs for a DO and all its subtargets, it does not display the CRs for DO versions unless you use the **–ci** option.

catcr allows precise control over report contents and format. It includes input and output filters and supports a variety of report styles. Input filters, such as **–select**, control which DOs are evaluated. All DOs that are evaluated can potentially appear in the final listing. Output filters, such as **–view_only**, control which DOs actually appear in the final listing. Often, this is a subset of all evaluated DOs.

You can tailor the report in several ways:

- Generate a separate report for each derived object on the command line (default), or a single, composite report for all derived objects on the command line (**–union**).
- Specify which derived objects to consider when compiling report output. The **–recurse**, **–flat**, **–union**, **–ci**, and **–select** options control which subtargets are evaluated. They generate recursive or flat-recursive reports of subtargets, evaluate checked-in DOs, and allow you to evaluate DOs with a particular name only.

- Select the kinds of items that appear in the report. The **–element_only**, **–view_only**, **–type**, **–name**, and **–critical_only** options exclude certain items from the report.
- Display the CR in makefile format (**–makefile**), rather than in a section-oriented format.
- Choose a normal, long, or short report style. Expanding the listing with **–long** adds comments and supplementary information; restricting the listing with **–short** lists file system objects only. You can also list simple pathnames rather than version-extended pathnames (**–nxname**), and relative pathnames rather than full pathnames (**–wd**).

The **–check** option determines whether the CR contains any unusual entries. For example, it determines whether the CR contains multiple versions of the same element, or multiple references to the same element with different names.

By default, **catcr** suppresses a CR entirely if the specified filters remove all objects (useful for searching). With the **–zero** option, the listing includes the headers of such CRs.

The following examples show how to display configuration records for DO versions.

- To display the configuration record for a single DO version:

  cleartool catcr test

```
Target test built by smg.user
Host "duck" running NT 4.0 (i586)
Reference Time 20-Jul-99.14:27:38, this audit started
20-Jul-99.14:27:39
View was swan:C:\views\smg_build2.vws
Initial working directory was V:\smg_build\dir
----------------------------
MVFS objects:
----------------------------
\smg_build\dir\test@@20-Jul.14:27.71
\smg_build\dir\test.txt@@\main\1              <20-Jul-99.14:27:23>
----------------------------
Build Script:
----------------------------
        copy test.txt test
----------------------------
```

- To display the configuration record for a derived object, including the CRs of all subtargets except DO versions:

cleartool catcr –recurse x

```
----------------------------
----------------------------
Target x built by smg.user
...
Target file2.txt built by smg.user
...
Target file2sub.txt built by smg.user
...
```

- To display the configuration record for a derived object, including the CRs of all subtargets:

cleartool catcr –recurse –ci x

```
----------------------------
----------------------------
Target x built by smg.user
...
Target file1.txt built by smg.user
...
Target file1sub.txt built by smg.user
...
Target file2.txt built by smg.user
...
Target file2sub.txt built by smg.user
...
```

DOs in Unavailable Views

catcr maintains a cache of tags of inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

For more information, see the **catcr** reference page.

Releasing DOs

A project team can use DO versions to make its product (for example, a library) available to other teams. Typically, the team establishes a release area in a separate VOB. For example:

- A library is built by its project team in one location—perhaps \monet\lib\libmonet.lib.
- The team periodically releases the library by creating a new version of a publicly accessible element—perhaps \publib\libmonet.lib.

You can generalize the idea of maintaining a development release area to maintaining a product release area. For example, a Release Engineering group maintains one or more release tree VOBs. The directory structure of the trees mirrors the hierarchy of files to be created on the release medium. (Because a release tree involves directory elements, it is easy to change its structure from release to release.) A release tree can be used to organize Release 2.4.3 as follows:

1. When an executable or other file is ready to be released, a release engineer checks it in as a version of an element in the release tree.
2. An appropriate version label (for example, **REL2.4.3**) is attached to that version, either manually by the engineer or automatically with a trigger.
3. When all files to be shipped have been labeled in this way, a release engineer configures a view to select only versions with that version label. As seen through this view, the release tree contains exactly the set of files to be released.
4. To cut a release tape, the engineer issues a command to copy the appropriately configured release tree.

Converting Nonshareable DOs to Shared DOs

Note: You cannot convert a shared or unshared DO to a nonshareable DO.

To convert a nonshareable DO to a shared DO, use the **winkin** command. **winkin** advertises the DO by making it shareable and writing information into the VOB and then promotes it (makes it shared). The command also advertises the DO's sub-DOs and siblings, even if you did not specify the **–siblings** option. This process changes the DO ID for each derived object.

The **view_scrubber –p** command performs the same operation. See the **winkin** and **view_scrubber** reference pages.

Automatic Conversion of Nonshareable DOs to Shareable DOs

Because you can change a view's DO property and shareable DOs cannot have nonshareable sub-DOs or siblings, situations can occur in which the build tool must convert nonshareable DOs into shareable DOs.

For example, you set your view's DO property to nonshareable and perform a build, creating nonshareable DOs. You then set your view's DO property to shareable and perform another build. The build tool determines that it can reuse some of the nonshareable DOs created in the first build to create shareable DOs in the second. It converts the nonshareable DOs to shareable DOs and reuses them.

Displaying VOB Disk Space Used for Derived Objects

The **dospace** command and the ClearCase Administration Console report VOB disk space used for shared derived objects. For more information, see the **dospace** reference page and the *Administrator's Guide* for Rational ClearCase.

Deleting Derived Objects

The **rmdo** command removes the data container and the VOB database object for a derived object. For more information, see the **rmdo** reference page.

Shareable derived objects and their data containers can be deleted independently. Deleting a nonshareable derived object deletes the DO.

Removing Data Containers for Derived Objects

The standard **del** command causes a shareable derived object to disappear from the dynamic view. The effect on physical data storage is as follows:

- If the DO's data container is in the view's private storage area, **del** deletes that data container.
- If the DO's data container is in a VOB storage pool, the data container is not affected.

In both cases, the derived object in the VOB database is not deleted. The only change to the derived object is that its reference count is decremented.

When a build overwrites a nonshareable or unshared DO, the MVFS removes the old data container from the dynamic view's private storage area, and creates a new one there. It also creates a new CR. At the operating system level, the effect is that an existing file is overwritten.

Scrubbing Derived Objects and Data Containers

A reference count of zero means that the derived object has been deleted or overwritten in every view that ever used it. This situation calls for *scrubbing*: automatic deletion of DO-related information from the VOB. Scrubbing can remove the derived object from the VOB database, its data container from a VOB storage pool (if the DO had ever been shared), and in some cases its associated CR, as well.

The **scrubber** utility removes derived objects from a VOB database and data containers from VOB storage pools. The **view_scrubber** utility removes data containers from a dynamic view's private storage area. For more information about scrubbing, see the *Administrator's Guide* for Rational ClearCase.

Degenerate Derived Objects

A derived object is complete if its VOB database object, data container, and configuration record (CR) are accessible. Because these entities exist independently, a derived object can become incomplete, or degenerate, if one entity is missing.

Data Container Deleted

When an unshared DO is removed with **del** or by a target rebuild, its VOB database object continues to exist in the VOB database (with a zero reference count), but the data container no longer exists. Such DOs are usually ignored by **lsdo**, but can be listed with the **–zero** option. The **scrubber** utility deletes zero-referenced DOs.

The **checkvob** command can find and fix missing container problems.

DO Deleted from VOB Database

When an unshared DO is removed from its VOB database with **rmdo**, the data container continues to be visible:

```
Z:\myvob\test> cleartool rmdo Vhelp.log
Removed derived object "Vhelp.log@@14-Sep.72783".
```

```
Z:\myvob\test> cleartool ls Vhelp.log
Vhelp.log [no config record]
```

In general, try to avoid using the **rmdo** command.

CR Unavailable

A newly created CR is stored in the dynamic view where its associated DOs were built. If that view becomes unavailable (for example, it is inadvertently destroyed or its host is temporarily down), the DO continues to exist in the VOB database, but operations that must access the CR fail:

```
cleartool: Error: Unable to find view '\\mars\vw_store\pink.vws'
from albd: error detected by ClearCase subsystem
cleartool: Error: See albd_log on host mars
cleartool: Error: Unable to contact View - error detected by ClearCase
subsystem
```

Displaying Contents of Configuration Records

The **catcr** command displays the contents of a configuration record. For more information, see the **catcr** reference page.

Comparing Configuration Records

Because config records provide complete records of how DOs are built, you can use them to determine how two builds differ. For example, you expected the build to reuse or wink in a DO that it rebuilt instead. You can compare the CRs for the two DOs to find out what aspect of the build environment was different.

To compare two existing CRs, use the **diffcr** command. For more information, see the **diffcr** reference page.

Attaching Labels or Attributes to Versions in a CR

You can attach a label or an attribute to the versions in the CR hierarchy of a derived object.

For example, to attach the **SMG_BUILD_5_03** label to the versions in the CR hierarchy of file.obj:

cleartool mklabel –c "may 99 build" –config file.obj SMG_BUILD_5_99

```
Created label "SMG_BUILD_5_99" on "\smg_test\" version
"\main\CHECKEDOUT".
Created label "SMG_BUILD_5_99" on "\smg_test\acc.c" version "\main\2".
Created label "SMG_BUILD_5_99" on "\smg_test\file.c" version
"\main\1".
```

For more information, see the description of the **–config** option in the **mkattr** and **mklabel** reference pages.

Configuring a View to Select Versions Used to Build a DO

To select the versions in the CR hierarchy of a derived object, use the **–config** version selector in your view's config spec. For example, the following config spec selects the versions in the CR hierarchy for hello.obj:

```
element * CHECKEDOUT
element * –config \dev\lib\hello.obj
element * \main\v3.8\LATEST
```

For more information, see the **config_spec** reference page.

Including a Makefile Version in a Configuration Record

To record a makefile version in a CR, use one of the following methods:

- Declare it as an explicit dependency in the makefile. To do this, you can use the **$(MAKEFILE)** variable. You must explicitly list any included makefiles that you want to record.

  The drawback to this method is that it causes targets that depend on the makefile to be rebuilt if there is any change to the makefile.

- Make it an implicit dependency by referring to it in a build script and use the special target **.DEPENDENCY_IGNORED_FOR_REUSE** to ignore it in subsequent rebuild decisions. You must explicitly list any included makefiles that you want to record.

  For example:

  ```
  .DEPENDENCY_IGNORED_FOR_REUSE: $(MAKEFILE)
  targ: dep1 dep2
     type $(MAKEFILE) > c:\temp\makefile
     touch targ
  ```

  The drawback to this method is that the makefile dependency is ignored for reuse, but it is not ignored for winkin.

- Use the **.MAKEFILES_IN_CONFIG_REC** special target. See *Special Targets* on page 73.

clearmake Makefiles and BOS Files

5

This chapter describes makefiles processed by the Rational ClearCase build program **clearmake**. This is a discussion of differences and ClearCase extensions rather than a complete description of makefile syntax. This chapter also describes build option specification files (BOS files), which contain temporary macros and ClearCase special targets.

For information on the **omake** build program, see the *OMAKE Guide*.

Makefile Overview

A makefile contains a sequence of entries, each of which specifies a build target, some dependencies, and the build scripts of commands to be executed. A makefile can also contain **make** macro definitions, target-dependent macro definitions, and build directives (special targets.)

- **Target/dependencies line**. The first line of an entry is a white-space-separated, nonnull list of targets, followed by a colon (**:**) or a double colon (**::**), and a (possibly empty) list of dependencies. Both targets and dependencies may contain ClearCase pathname patterns. (See the **wildcards_ccase** reference page.)

  The list of dependencies may not need to include source objects, such as header files, because **clearmake** detects these dependencies. However, the list must include build-order dependencies, for example, object modules and libraries that must be built before executables. (See *Build-Order Dependencies* on page 36.)

- **Build script**. Text that follows a semicolon (**;**) on the same line and all subsequent lines that begin with a <TAB> character constitute a build script: a set of commands to be executed in a command interpreter. A command can be continued onto the next text line with a \<NL> sequence. Any line beginning with a number sign (**#**) is a comment.

  A build script ends at the first nonempty line that does not begin with a <TAB> or number sign (**#**); this begins a new target/dependencies line or a make macro definition.

Build scripts must use standard pathnames only. Do not include view-extended or version-extended pathnames in a build script.

Executing a build script updates the target and is called a *target rebuild*. The commands in a build script are executed one at a time, each in its own instances of the command interpreter.

Note that **clearmake** always completely eliminates a \<NL> sequence, even in its compatibility modes. Some other **make** programs sometimes preserve such a sequence.

- **Make macro**. A **make** macro is an assignment of a character-string value to a simple name. By convention, all letters in the name are uppercase (for example, **CFLAGS**).
- **Target-dependent macro definitions**. A target-dependent macro definition takes the form *target-list* **:=** *macro_name* **=** *string*

  You can use macros in makefiles or in BOS files. For more information, see *Target-Dependent Macro Definitions* on page 82.
- **Special targets**. A line that begins with a dot (**.**) is a special target, which acts as a directive to **clearmake**.
- **Special characters in target names**. You can use special characters in target names by immediately preceding each special character with a backslash (\).

Build Options Specification Files

A build options specification (BOS) file is a text file containing macro definitions and/or ClearCase special targets. We recommend that you place temporary macros (such as **CFLAGS=/Zi** and others not to be included in a makefile permanently) in a BOS file, rather than specifying them on the **clearmake** command line.

By default, **clearmake** reads BOS files in this order:

1 The default BOS files

  a The file .clearmake.options in your home directory (as indicated by the HOME environment variable or in the user profile), which is the place for macros to be used every time you execute **clearmake**.

  b One or more local BOS files, each of which corresponds to one of the makefiles specified with a **–f** option or read by **clearmake**. Each BOS file has a name in the form makefile-name.options. For example:

```
makefile.options
```

```
Makefile.options

project.mk.options
```

2 BOS files specified in the CCASE_OPTS_SPECS environment variable.

3 BOS files specified on the command line with **–A**.

If you specify **–N**, **clearmake** does not read default BOS files.

clearmake displays the names of the BOS files it reads if you specify the **–v** or **–d** option, or if %CCASE_VERBOSITY %>= 1.

For information about the contents of BOS files, see *Setting Up the Client Host* on page 107.

When **clearmake** shops for a derived object to wink in to a build, it may find DOs from a view that is unavailable (because the view server host is down, the **albd_server** is not running on the server host, and so on). Attempting to fetch the DO's configuration record from an unavailable view causes a long time-out, and the build may reference multiple DOs from the same view.

clearmake and other **cleartool** commands that access configuration records and DOs (**lsdo**, **describe**, **catcr**, **diffcr**) maintain a cache of tags for inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

Note: The cache is not persistent across **clearmake** sessions. Each recursive or individual invocation of **clearmake** attempts to contact a view whose tag may have been cached in a previous invocation.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

Format of Makefiles

The following sections describe the special considerations for using makefiles with **clearmake**.

Note: For information about environment variables that affect clearmake, see the **env_ccase** reference page. You can also use the **-d** or **-v** option to the **clearmake** command to view a list of environment variables that **clearmake** reads during the build.

Restrictions

clearmake does not support the use of standard input as a makefile.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive (library) created by **lib**, or some other librarian. For example:

```
hello.lib : hello.lib(mod1.obj) hello.lib(mod2.obj)
```

The string within parentheses refers to a member (object module) within the library. Use of function names within parentheses is not supported. Thus, **hello.lib(mod1.obj)** refers to an archive that contains object module **mod1.obj**. The expression **hello.lib(mod1.obj mod2.obj)** is not valid.

Inference rules for archive libraries have this form:

.sfx.lib
: where *sfx* is the file name extension (suffix) from which the archive member is to be made.

The way in which **clearmake** handles incremental archive construction differs from other **make** variants. For more information, see *Working with Incremental Update Tools* on page 43.

Command Echoing and Error Handling

You can control the echoing of commands and the handling of errors that occur during command execution on a line-by-line basis or on a global basis.

You can prefix any command with one or two characters, as follows:

–	Causes **clearmake** to ignore any errors during execution of the command. By default, an error causes **clearmake** to terminate. The command-line option **–i** suppresses termination-on-error for all command lines.
@	Suppresses display of the command line. By default, **clearmake** displays each command line just before executing it. The command-line option **–s** suppresses display of all command lines. The **–n** option displays commands, but does not execute them.

–@ @– These two prefixes combine the effect of **–** and @.

The **–k** option provides for partial recovery from errors. If an error occurs, execution of the current target (that is, the set of commands for the current target) stops, but execution continues on other targets that do not depend on that target.

clearmake reserves **-2 (254)** as a return code meaning "unable to execute." This error is distinct from shell-returned errors and cannot be ignored by the use of **-i**. Users who plan to write scripts that return standard error codes should avoid the use of **-2 (254)**.

Built-In Rules

File name extensions (suffixes) and their associated rules in the makefile override any identical file name extensions in the built-in rules. **clearmake** reads built-in rules from the file *ccase-home-dir*\etc\builtin.mk when you run in standard compatibility mode. In other compatibility modes, other files are read.

Include Files

If a line in a makefile starts with the string **include** or **sinclude** followed by white space (at least one <SPACE> or <TAB> character), the rest of the line is assumed to be a file name. (This name can contain macros.) The contents of the file are placed at the current location in the makefile.

For **include**, a fatal error occurs if the file is not readable. For **sinclude**, a nonreadable file is silently ignored.

Macros

The following sections describe the order of precedence of macros in a **clearmake** build, and the different types of macros.

Order of Precedence of Make Macros and Environment Variables

By default, the order of precedence of macros and environment variables is as follows:

1. Target-dependent macro definitions
2. Macros specified on the **clearmake** command line
3. Make macros set in a BOS file
4. Make macro definitions in a makefile
5. Environment variables

For example, target-dependent macro definitions override all other macro definitions, and macros specified on the **clearmake** command line override those set in a BOS file.

If you use the **–e** option to **clearmake**, environment variables override macro definitions in the makefile.

All BOS file macros (except those overridden on the command line) are placed in the build script's environment. If a build script recursively invokes **clearmake**:

- The higher-level BOS file setting (now transformed into an EV) is overridden by a make macro set in the lower-level makefile. However, if the recursive invocation uses **clearmake**'s **–e** option, the BOS file setting prevails.
- If another BOS file (associated with another makefile) is read at the lower level, its make macros override those from the higher-level BOS file.

For a list of all environment variables, see the **env_ccase** reference page.

Make Macros

A *macro definition* takes this form:

macro_name **=** *string*

Macros can appear in the makefile, on the command line, or in a build options specification file. (See *Build Options Specification Files* on page 66.)

Macro definitions require no quotes or delimiters, except for the equal sign (=), which separates the macro name from the value. Leading and trailing white-space characters are stripped. Lines can be continued using a \<NL> sequence; this sequence and all surrounding white space is effectively converted to a single <SPACE> character. *macro_name* cannot include white space, but *string* can; it includes all characters up to an unescaped <NL> character.

clearmake performs macro substitution whenever it encounters either of the following in the makefile:

$(*macro_name***)**
$(*macro_name***:***subst1***=***subst2***)**

It substitutes *string* for the macro invocation. In the latter form, **clearmake** performs an additional substitution within *string*: all occurrences of *subst1* at the end of a word within *string* are replaced by *subst2*. If *subst1* is empty, *subst2* is appended to each word in the value of *macro_name*. If *subst2* is empty, *subst1* is removed from each word in the value of *macro_name*.

For example:

```
z:\myvob> type Makefile
C_SOURCES = one.c two.c three.c four.c
test:
     echo OBJECT FILES are: $(C_SOURCES:.c=.obj)
     echo EXECUTABLES are: $(C_SOURCES:.c=.exe)

z:\myvob> clearmake test
OBJECT FILES are: one.obj two.obj three.obj four.obj
EXECUTABLES are: one.exe two.exe three.exe four.exe
```

Internal Macros

clearmake maintains these macros internally. They are useful in rules for building targets.

$*
(Defined only for inference rules) The file name part of the inferred dependency, with the file name extension deleted.

$@
The full target name of the current target.

$<
(Defined only for inference rules) The file name of the implicit dependency.

$?
(Defined only when explicit rules from the makefile are evaluated) The list of dependencies that are out of date with respect to the target. When configuration lookup is enabled (default), it expands to the list of all dependencies, unless that behavior is modified with the **.INCREMENTAL_TARGET** special target. In that case, **$?** expands to the list of all dependencies different from the previously recorded versions.

When a dependency is an archive library member of the form lib(file.obj), the name of the member, file.obj, appears in the list.

$%
(Defined only when the target is an archive library member) For a target of the form lib(file.obj), **$@** evaluates to `lib` and **$%** evaluates to the library member, file.obj.

MAKE
The name of the make processor (that is, **clearmake**). This macro is useful for recursive invocation of **clearmake**.

MAKEFILE During makefile parsing, this macro expands to the pathname of the current makefile. After makefile parsing is complete, it expands to the pathname of the last makefile that was parsed. This holds only for top-level makefiles, not for included makefiles or for built-in rules; in these cases, it echoes the name of the including makefile.

Use this macro as an explicit dependency to include the version of the makefile in the CR produced by a target rebuild. For example:

```
supersort: main.obj sort.obj cmd.obj $(MAKEFILE)
    link /out:$@ $?
```

For more information, see *Including a Makefile Version in a Configuration Record* on page 63.

MAKEFLAGS This macro passes flags to sub-makes, including flags that take arguments and macro definitions. **clearmake** reads the contents of **MAKEFLAGS** at startup and amends it to include any flags not specific to ClearCase passed at the command line. Any flags specific to ClearCase are passed through **CCASE_MAKEFLAGS** and if **clearmake** detects these flags in **MAKEFLAGS**, it moves them to **CCASE_MAKEFLAGS**.

Flags passed through **MAKEFLAGS**:

-I, **-p**, **-N**, **-w**, **-e**, **-r**, **-i**, **-k**, **-n**, **-q**, **-s**

Flags passed through **CCASE_MAKEFLAGS**:

-A, **-B**, **-N**, **-b**, **-v**, **-C**, **-U**, **-M**, **-V**, **-O**, **-T**, **-F**, **-R**, **-c**, **-u**, **-d**

This functionality is available for all compatibility modes.

VPATH Macro

The **VPATH** macro specifies a search path for targets and dependencies. **clearmake** searches directories in **VPATH** when it fails to find a target or dependency in the current working directory. **clearmake** searches only in the current view. The value of **VPATH** can be one directory pathname, or a semicolon-separated list of directory pathnames. (In Gnu compatibility mode, you can also use spaces as separators.)

As **clearmake** qualifies makefile dependencies (explicit dependencies in the makefile), the process of configuration lookup is **VPATH**-sensitive. Thus, if a newer version of a dependent file appears in a directory on the search path before the pathname in the CR (the version used in the previous build), **clearmake** rejects the previous build and rebuilds the target with the new file.

The **VPATH** setting may affect the expansion of internal macros, such as **$<**.

Special Targets

Like other build tools, **clearmake** interprets certain target names as declarations. Some of these special targets accept lists of patterns as their dependents, as noted in the description of the target. 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 file name extension of the name must match the part of the pattern after the %. For example:

Name	Matches	Does not match

`\dir\subdir\x.obj`	`%.obj`
	`x.obj`
	`subdir\%.obj`
	`subdir\x.obj`
		`\dir\subdir\otherdir\x.obj`

The following targets accept lists of patterns:

- **.DEPENDENCY_IGNORED_FOR_REUSE**
- **.INCREMENTAL_REPOSITORY_SIBLING**
- **.INCREMENTAL_TARGET**
- **.NO_CMP_NON_MF_DEPS**
- **.NO_CMP_SCRIPT**
- **.NO_CONFIG_REC**
- **.NO_DO_FOR_SIBLING**
- **.NO_WINK_IN**
- **.SIBLING_IGNORED_FOR_REUSE**

Special Targets for Use in Makefiles

.DEFAULT :
If a file must be built, but there are no explicit commands or relevant built-in rules to build it, the commands associated with this target are used (if it exists).

.IGNORE :
Same effect as the **–i** option.

.PRECIOUS : *tgt* ...
The specified targets are not removed when an interrupt character (typically, CTRL+C) is typed.

.SILENT :
Same effect as the **–s** option.

Special Targets for Use in Makefiles or BOS Files

You can use the following special targets either in the makefile itself or in a build options specification file. See *Build Options Specification Files* on page 66.

.DEPENDENCY_IGNORED_FOR_REUSE: *file* ...

The dependencies you specify are ignored when **clearmake** determines whether a target object in a VOB is up to date and can be reused. By default, **clearmake** considers that a target cannot be reused if its dependencies have been modified or deleted since it was built. This target applies only to reuse, not to winkin. Also, this target applies only to detected dependencies, which are not declared explicitly in the makefile.

You can specify the list of files with a tail-matching pattern, for example, **%.module**.

Unlike the files listed in most special targets, the files on this list refer to the names of dependencies and not the names of targets. As such, the special target may apply to the dependencies of many targets at once. This special target 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.

Note: **clearmake** does not provide a special target to ignore dependencies for winkin because, in general, is it not safe to winkin the partial results of a build. If the siblings are not used by the build tools, then they should probably be deleted by the build script or created outside the VOB. If the siblings are used by the build tools, then their contents may depend on the sequence of builds performed in that view and it is unsafe for **clearmake** to winkin the other build artifacts while ignoring the dependent siblings.

.INCREMENTAL_REPOSITORY_SIBLING: *file* ...

The files listed are incremental repository files created as siblings of a primary target and may contain incomplete configuration information. This special target is useful when a toolset creates an incremental sibling object and you want to prevent clearmake from winking in a primary target that has this sibling.

You can specify the list of files with a tail-matching pattern, for example, **%.pdb**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target may apply to the siblings of many targets at once. This special target 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.

Caution: If you have an incremental sibling and you do not use **.INCREMENTAL_REPOSITORY_SIBLING**, you might wink in the incremental sibling inadvertently, which could overwrite the version of the sibling in your view and cause problems.

.INCREMENTAL_TARGET: *tgt* ...

This target performs incremental configuration record merging for the listed targets; in other words, it combines dependency information from instances of this target generated previously with the current build of this target. This special target is most useful when building libraries, 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, **%.lib**.

For information about restructuring a makefile to build incremental archive files, see *Working with Incremental Update Tools* on page 43.

Note: .INCREMENTAL_TARGET applies only to makefile targets built incrementally using a single make rule. Do not use it for the following kinds of files:

- Files built incrementally that are not makefile targets. For example, sibling objects like log files or template repositories.
- Files built incrementally from several different build scripts.

The general guideline is that if you are not building a library in a single makefile rule, and you are not building an executable using an incremental linker, do not use **.INCREMENTAL_TARGET**.

.JAVA_TGTS: *file ...*

This special target is used to handle subclasses generated by Java compilers.

In the makefile, any file name that matches the pattern will allow a **$** to be escaped by another **$**. For example, to specify **a$x.class**:

You can specify the list of files with a tail-matching pattern; for example, **%.class:**

```
.JAVA_TGTS: %.class
.java.class:
        javac $<
a$$x.class: a.class
```

Note that **$$** mapping to a single **$** is default behavior in Gnu make compatibility mode. For more information, see the **makefile_gnu** reference page.

.JAVAC:

Use this special target to enable **clearmake** to use heuristics on audits of Java builds to accurately evaluate .class dependencies. These dependencies are then stored in .class.dep files for future **clearmake** runs, and they enable those runs to build .class targets in the same order that the Java compiler does.

This special target must be used with no dependencies and no build script:

```
JAVAC:
```

Other than that, makefiles must use implicit suffix or pattern rules. For example:

```
.SUFFIXES: .java .class

.java.class:

        rm -f $@

        $(JAVAC) $(JFLAGS) $<
```

For compatibility modes that support them, use implicit pattern rules. For example:

```
%.class: %.java

        rm -f $@

        $(JAVAC) $(JFLAGS) $?
```

The makefiles must also use absolute paths for .class targets. For example:

```
all_classes:\proj\src\pkg1\foo.class
```

clearmake contains a builtin macro function you can use to specify absolute paths:

```
$(javaclasses)
```

For more information about the **.JAVAC** special target, see Chapter 8.

.MAKEFILES_IN_CONFIG_REC: *file ...*

Use this special target to record the versions of makefiles in the configuration records of derived objects.

This target takes an optional dependency list, which may be a pattern. When used without a dependency list, this target causes all makefiles read by a build session to be recorded in the configuration record of all derived objects built during that build session.

To conserve disk space, you may want to supply a dependency list to this target so that, for example, only DOs built for top-level targets have the makefiles recorded in their configuration records.

.MAKEFILES_AFFECT_REUSE:

By default, makefiles recorded by using the **.MAKEFILES_IN_CONFIG_REC** special target do not affect DO reuse. You can use this target to enable recorded makefiles to affect DO reuse. (If you want to have some recorded makefiles affect reuse, but not all, you can also use the **.DEPENDENCY_IGNORED_FOR_REUSE** special target in conjunction with this target.)

Note: If a makefile is declared an explicit dependency of a target, it always affects DO reuse for that target, whether or not .**MAKEFILES_AFFECT_REUSE** was used.

Makefiles recorded in a configuration record are labeled by **mklabel –config**. Makefiles that were recorded in a configuration record but that were not recorded by using .**MAKEFILES_AFFECT_REUSE** are ignored by **catcr –critical_only** and **diffcr –critical_only**.

.NO_CMP_NON_MF_DEPS: *tgt* ...

The specified targets are built as if the **–M** option were specified; if a dependency is not declared in the makefile, it is not used in configuration lookup.

You can specify the list of files with a tail-matching pattern, for example, %.obj.

.NO_CMP_SCRIPT : *tgt* ...

The specified targets are built 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.

You can specify the list of files with a tail-matching pattern, for example, %.obj.

.NO_CONFIG_REC : *tgt* ...

The specified targets are built as if the **–F** option were specified; modification time is used for build avoidance, and no CRs or derived objects are created.

You can specify the list of files with a tail-matching pattern; for example, %.obj.

.NO_DO_FOR_SIBLING : *file* ...

Use this target to disable 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, **%.tmp**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target may apply to the siblings of many targets at once. This special target 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.

.NO_WINK_IN : *tgt* ...

The specified targets are built as if the **–V** option were specified; configuration lookup is restricted to the current view.

You can specify the list of files with a tail-matching pattern, for example, %.obj.

.NOTPARALLEL : *tgt* ...

Without any *tgt* arguments, disables parallel building for the current makefile. **clearmake** builds the entire makefile serially, one target at a time. With a set of *tgt* arguments, prevents **clearmake** from building any of the targets in the set in parallel with each other. However, targets in a set can be built in parallel with targets in a different set or with any other targets. For example:

.NOTPARALLEL:%.a
.NOTPARALLEL:acc1 acc2

clearmake does not build any .a file in parallel with any other .a file, and acc1 is not built in parallel with acc2. However, **clearmake** may build .a files in parallel with acc1 or acc2.

.NOTPARALLEL does not affect lower-level builds in a recursive make, unless you specify it in the makefiles for those builds or include it in a BOS file.

You can specify the list of files with a tail-matching pattern, for example, %.a.

See also Chapter 9, *Setting Up a Parallel Build*.

.SIBLING_IGNORED_FOR_REUSE: *file* ...

The *files* are ignored when **clearmake** determines whether a target object in a VOB is up to date and can be reused. This is the default behavior, but this special target can be useful in conjunction with the **.SIBLINGS_AFFECT_REUSE** special target or **–R** command-line option. This target applies only to reuse, not to winkin.

You can specify the list of files with a tail-matching pattern, for example, **%.sbr**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target 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:

Build as if the **–R** command line option were specified; examine 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 is up to date, **clearmake** ignores modifications to objects created by the same build rule that created the target (sibling derived objects). This directive tells **clearmake** to consider a target out of date if its siblings have been modified or deleted.

Sharing Makefiles Between UNIX and Windows

clearmake is available on both UNIX and Windows NT. In principle, you can write portable makefiles, but in practice, the obstacles are substantial. The variations in tool and argument names between systems makes writing portable build scripts particularly challenging. If you choose to pursue portable makefiles, use the following general procedures to produce usable results.

- **Start on UNIX; avoid most compatibility modes.** On Windows NT, **clearmake** supports Gnu compatibility mode but does not support others (for example, Sun compatibility mode). Instead, it supports basic **make** syntax. To write or tailor transportable makefiles, begin makefile development on UNIX, without compatibility modes other than Gnu in effect. Gnu generates errors and warnings for problematic syntax. When things work cleanly on UNIX, move your makefiles to Windows NT for testing.
- **Use a makefile-generating utility, such as imake, to generate makefiles**. Use **imake** or some other utility to generate the makefiles you will need, including **clearmake** makefiles for Windows NT.

Using Makefiles on Windows

There are several rules to follow when constructing or converting makefiles for use by **clearmake** on a Windows host. Note that, as a general rule, your makefiles must match the syntax required by **clearmake** on UNIX.

Case-Sensitivity Guidelines

The following sections describe how you must specify build macros, targets, and dependencies in makefiles to avoid case problems.

Build Macros and Case-Sensitivity

clearmake is case-sensitive with respect to makefile macros. Consider a makefile macro reference, `$(CPU)`. There are numerous input sources from which to satisfy this macro:

- From the makefile itself
- From the current table of environment variables
- From the command line
- From a build option specification (BOS) file

For any macro to be expanded correctly from any of these sources, the macro definition and macro reference must be in the same case. For example, `$(CPU)` is not replaced by the value of a CPU environment variable.

Makefile Target/Dependency Pathnames

When you write makefiles, you must be aware of the MVFS setting on your computer and specify targets and dependencies accordingly. If the MVFS is case-preserving, you must use case-correct pathnames in makefiles to guarantee the consistency of the resulting config records. Even if your MVFS is not case-preserving, we recommend that you use case-correct pathnames so that users on case-preserving computers can share the makefile.

Note: The **–d** option to **clearmake** warns you when case is the only difference in pathnames in the makefile and on the file system.

Table 1 describes makefile requirements for the different MVFS settings.

Table 1 MVFS Settings and Case Requirements for Makefiles

MVFS settings	Build tool and MVFS behavior	Makefile requirements
Case-insensitive and case preserving	The MVFS preserves the case of created files. The build tool looks for the file as it is specified in the makefile.	The case of the target must match the case of the file produced by the MVFS.
Case-insensitive and non-case-preserving	The MVFS converts the names of all files created to lowercase. The build tool looks for a lowercase file name.	The case of the target does not matter.

Table 1 MVFS Settings and Case Requirements for Makefiles

MVFS settings	Build tool and MVFS behavior	Makefile requirements
Case-sensitive and case-preserving	The MVFS preserves the case of created files. The build tool looks for the file as it is specified in the makefile.	The case of the target must match the case of the file produced by the MVFS.

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.

Using UNIX-Style Command Shells in Makefiles

On Windows, **clearmake** accepts either slashes (/) or backslashes (\ ) in pathnames. However, **clearmake** uses a backslash as the separator in any pathnames that it constructs in build scripts (for example, as a result of **VPATH** directory searching). This can cause problems with command shells similar to UNIX that require slashes in any pathnames supplied to them in command lines.

If you are using such a shell (for example, by setting the SHELL makefile variable accordingly), you can force **clearmake** to use slashes when constructing pathnames. To do this, set the CMAKE_PNAME_SEP variable:

CMAKE_PNAME_SEP = /

You can set CMAKE_PNAME_SEP in the makefile, in the BOS file, on the command line, or as an environment variable. **clearmake** recognizes CMAKE_PNAME_SEP for pattern rules as well as standard targets.

BOS File Entries

The following sections describe the entries you can put in BOS files.

Standard Macro Definitions

A standard macro definition has the same form as a make macro defined in a makefile:

macro_name **=** *string*

For example:

```
CDEBUGFLAGS = /Zi
```

Target-Dependent Macro Definitions

A target-dependent macro definition takes this form:

target-pattern-list **:=** *macro_name* **=** *string*

Any standard macro definition can follow the **:=** operator; the definition takes effect only when the targets that match patterns in *target-pattern-list* and their dependencies are processed. Patterns in the *target-pattern-list* must be separated by white space. For example:

```
x.o y.o := CDEBUGFLAGS=/Zi
```

Two or more higher-level targets can have a common dependency. If the targets have different target-dependent macro definitions, the dependency is built using the macros for the first higher-level target **clearmake** considered building (whether or not **clearmake** actually built it).

Shell Command Macro Definitions

A shell command macro definition replaces a macro name with the output of a shell command:

macro_name **:sh =** *string*

This defines the value of *macro_name* to be the output of *string*, any shell command. In command output, <NL> characters are replaced by <SPACE> characters. For example:

```
NT_VER :sh = VER
```

Special Targets

You can use some ClearCase special targets in a build options spec. See *Special Targets for Use in Makefiles or BOS Files* on page 74.

Include Directives

To include one BOS file in another, use the **include** or **sinclude** (silent include) directive. For example:

```
include \lib\aux.options

sinclude $(OPTS_DIR)\pm_build.options
```

Comments

A BOS file can contain comment lines, which begin with a number sign (#).

SHELL Environment Variable

clearmake does not use the SHELL environment variable to select the shell program in which to execute build scripts. It uses Windows **cmd.exe**, unless you specify another program with a **SHELL** macro. You can specify **SHELL** on the command line, in the makefile, or in a build options spec; the value of **SHELL** must be a full pathname, and on Windows, it must include the file extension.

Note: If **clearmake** determines that it can execute the build script directly, it does not use the shell program even if you specify one explicitly. If you use Windows .bat files in build scripts, you must make them executable (use the **cleartool protect** command). To force **clearmake** to always use the shell program, set the environment variable CCASE_SHELL_REQUIRED.

CCASE_BRANCH0_REUSE Environment Variable

When **clearmake** evaluates a derived object for usability in the view, it compares versions of files recorded in the derived object's configuration record to versions of those same files that are in the current view. Generally, any version mismatch prevents **clearmake** from reusing the DO. However, if the DO used version main\123 and the view sees version main\123\branch\0, **clearmake** considers this to be a match. You can disable this default behavior by setting the environment variable CCASE_BRANCH0_REUSE in the shell before running **clearmake**.

Using clearmake Compatibility Modes

6

clearmake is designed for compatibility with existing make programs, which minimizes the changes you need to make to your makefiles. There are many variants of **make**, and each provides different sets of extended features. **clearmake** does not support all features of all variants, and we do not guarantee absolute compatibility.

If your makefiles use only the common extensions, they will probably work with **clearmake** without changes. If you must use features that **clearmake** does not support, consider using another make program in a **clearaudit** shell. This alternative provides build auditing (configuration records), but does not provide build avoidance (*winkin*).

Note: When building with configuration records, **clearmake** handles double-colon rules differently from other **make** programs. For more information, see *How clearmake Interprets Double-Colon Rules* on page 37.

To specify a compatibility mode, take one of the following actions:

- Use the environment variable CCASE_MAKE_COMPAT in a build options specification file or in your environment. For more information, see Chapter 5, *clearmake Makefiles and BOS Files*.
- Use the **–C** option with **clearmake**. For more information, see the **clearmake** reference page.

You can use the following compatibility modes:

gnu	Free Software Foundation Gnu make
std	Standard **clearmake** with no compatibility mode enabled. (Use this option to nullify a setting of the environment variable CCASE_MAKE_COMPAT.)

For information about specific **clearmake** compatibility modes, see the **makefile_gnu** and **makefile_ccase** reference pages.

Using ClearCase to Build C++ Programs

7

This chapter provides guidelines and instructions for using the ClearCase build utilities, **omake** and **clearmake**, in C++ development environments such as Microsoft Visual C++. The way that C++ development environments manage their files can cause conflicts with the ClearCase build utilities. Possible symptoms of the conflict:

- **clearmake** or **omake** rebuilds an object it could have winked in.
- When **clearmake** or **omake** winks in an incremental repository (database) file, information can be lost if the winked-in repository was not built with the same components that exist in the development view.

The exact nature of the symptoms depends on the compiler you use. For most Windows NT C++ compilers, these problems do not arise. The Microsoft Visual C++ environment causes a number of conflicts, which Rational ClearCase works around by including a special makefile fragment if you use the SCC integration with Visual C++. This chapter describes how this makefile fragment works and also presents alternate possibilities for working around the problems.

These C++ compilers apparently do not cause any conflict with ClearCase building:

- Borland C++
- Symantec C++
- Watcom C/C++

Using clearmake or omake Instead of Other make Programs

clearmake and **omake** ensure correct reuse decisions of different versions of files selected by a config spec, determine the complete dependency list accurately with build auditing, and allow identical derived objects from other views to be winked in to the current view. However, using another **make** program may be desirable for performance or compatibility reasons. If you use other **make** programs, do so with caution.

A standard **make** program compares the date and time of the target with that of its dependencies. If any dependency is newer than the target, **make** rebuilds the target.

Because of configuration specifications and the winkin feature, the set of files visible in a view at any given time is very dynamic. It is common to rebuild a target because the view selects a dependency that is different, though not necessarily newer, than the one used in the previous build of the target. The ClearCase build avoidance mechanism provides a more precise method of determining whether targets are up to date during builds by using the information stored in configuration records (CRs) to make correct decisions about reuse. CRs record the versions of all dependencies (whether listed in the makefile or not) of a target, rather than only the date and time of its last modification. If the version of a dependency from a CR doesn't match the version of the file selected by the view, **clearmake** or **omake** rebuilds the target.

The problem with using a **make** tool other than **clearmake** or **omake** is that the tool may make incorrect rebuild decisions if the view's config spec is changed and different file versions are selected. However, if you are sure that the config spec will not change and that any labels used in it will not be moved to versions with earlier dates, a standard **make** program will make correct reuse decisions. Using the **rmver** command to remove a version used in a build can also change the modification time and cause **make** to fail to rebuild when necessary. However, this is not a recommended practice; you do not obtain the benefits of CRs and it is not realistic to assume that no one will change the config spec or move any referenced labels improperly. Because it is somewhat risky to use common **make** programs, use them for debug builds, not for production.

Using Visual C++ with ClearCase

The ClearCase build utility **omake** (and, to a much lesser degree, **clearmake**) can be used to build in a Visual C++ development environment.

omake

omake can read Visual C++ makefiles, but only if NMAKE emulation mode is specified with the **–EN** parameter. In addition, if you are working outside the ClearCase MVFS (that is, neither on drive M nor on a drive in a ClearCase view), or if you disable configuration management (CM) features with the **–L** option, **omake** behaves like Visual C++ NMAKE, and no special options are needed. However, if you want to use **omake** within the MVFS (that is, in ClearCase VOBs) to produce DOs, you will find it most convenient to use the SCC integration with Visual C++, which maximizes the cooperation between ClearCase and Visual C++. For information about using **omake** with Visual C++, see the Help for the **omake** and Visual C++ integration.

If you are not using the SCC integration or are using Visual C++ 2.*x*, **omake** does not have access to the internal environment of the Visual C++ development environment.

Therefore, you must set the INCLUDE, LIB, and PATH environment variables before you run Visual C++ rather than rely on the directory lists set in the IDE options. Also, because **omake** cannot access the Visual C++ build rules, it must get build script information from either the builtins.nm file or the makefile.

If you are using Visual C++ 5.0 or 6.0, you must export your makefile before running **omake**. You can either click **Tools > Options > Build** and set the option **Export makefile when saving project file**, or export your makefile after making changes to the project settings. To export your makefile:

1. Check out the file *project-name*.mak (if it is an element).
2. Click **Project > Export Makefile**.
3. If you are prompted to select which configurations to export, select any configurations that will be built with **omake**.

   Visual C++ creates *project-name*.mak.

clearmake

You can use **clearmake** only if you are not using makefiles generated by Visual C++ — that is, if you write your own makefiles that call the Visual C++ compiler, linker, and so on.

Incremental Repositories in Visual C++

Program database files (PDBs) and incremental database files (IDBs) are built up with information from compilations of many .c or .cpp source files. This type of database file is referred to as an *incremental repository*. The consequence of using these incremental repositories is that object files cannot be winked in without making the PDB/IDB out of date. There is no way to wink in a portion of a file, a feature that would be required to wink in the object and the section of the PDB/IDB that holds the debug information for that specific object. For this reason, the SCC integration disables winkin for targets with a PDB or IDB sibling DO. Note that the **cleartool winkin** command can be used to wink in these objects manually; this method is described in *Using the winkin Command* on page 92.

Alternative: Using C7 Compatible Debug Information

The C/C++ compiler generates debug information in two different ways:

- C7 - information (**/Z7** parameter from the command line) which is compatible with the V7 compiler. Debug information is stored in resultant object files.
- Create one or more .PDB files that store the debug information.

C7-style debug information is the ideal choice when using **clearmake** or **omake** because information is stored directly in the object file. Because no files are shared by builds of multiple targets, winkin is automatic. If you use multiple views to build the same versions of files, this method is the most advantageous way of working. Automatic winkin means that **clearmake** or **omake** shops in the VOB for similar DOs and determines whether the previously built DO is the same as would be produced by executing the target's build script. If they would be the same, the target build previously is winked in (an operation that is in most cases faster than executing the build script), and the two views share a single disk image of the target file.

Base the decision on which debug information style to use on whether PDBs are necessary, whether other necessary features require PDBs (some types of precompiled headers and incremental compiles, for example), and whether an appropriate DO in your environment is likely to exist and be shareable.

Switching between PDBs and C7 debug information is simple. In the Visual C++ IDE:

1. Click **Project > Settings**.
2. Select the debug target in the **Settings For** tree.
3. Click the **C/C++** tab.
4. In the **Category** list, click **General**.
5. In the **Debug Info** list, click either **Program Database** or **C7 Compatible**.

Using vcmake.mak to Prevent Reuse Mismatches

The makefile fragment vcmake.mak defines files with a .pdb or .idb extension as incremental repositories. Specifying these siblings, which may contain incomplete information, prevents the ClearCase build tool from winking in their associated primary target.

To use the information in vcmake.mak, find your particular environment in Table 2 and follow the instructions.

Table 2 Using vcmake.mak

Environment	Instructions
omake and the SCC integration with Visual C++	No action. vcmake.mak is included by **omake**.

Table 2 Using vcmake.mak

<table>
<tr><th>Environment</th><th>Instructions</th></tr>
<tr><td>omake without Visual C++</td><td rowspan="2">Do one of the following:
▪ On the omake command line, add @ccase-home-dir\bin\vcmake.opts or specify –f ccase-home-dir\bin\vcmake.mak
▪ Set the OMAKEOPTS EV to @ccase-home-dir\bin\vcmake.opts
Note: Substitute your ClearCase installation directory for ccase-home-dir.</td></tr>
<tr><td>omake and Visual C++ without the SCC integration</td></tr>
<tr><td>clearmake</td><td>Insert the following lines in your makefile:
<code>.INCREMENTAL_REPOSITORY_SIBLING : %.idb %.IDB %.pdb %.PDB</code>
<code>.NO_WINK_IN : %.trg %.TRG</code></td></tr>
</table>

If you do not use the appropriate procedure, DOs built in one environment cannot be shared with other environments, and you see error message like this one:

```
Cannot reuse ".\..\bin.pcu\jdsample.obj" - mismatch between config
record flag and makefile directive for
"\adept\src\pub\libjpeg\bin.pcu\vc50.idb"
```

Browser Files

The browser generator, **bscmake**, has a performance enhancement that can cause problems with configuration records. By default, **bscmake** truncates all existing browser information files (called SBRs). If you use C7 debug information to allow winkin to work automatically, running **bscmake** in the default mode causes winkin to fail. **clearmake** or **omake** never finds a DO that can be winked in: the SBR created when the object file was created is no longer available because **bscmake** overwrote it. A **bscmake** option is available to cause it to run in nonincremental mode, where it reads all existing SBR files without modifying them. Using this parameter with C7 debug mode permits winkin to work correctly.

To run **bscmake** in nonincremental mode, you must specify the **/n** parameter. Add this parameter on all release builds in which the DOs are versioned. To add **/n** using the IDE:

1 Click **Project > Settings**.

2 Select the desired target in the **Settings For** tree.

3 Click the **Browse Info** tab.

4 In the **Project Options** box, add **/n**.

Using the winkin Command

If you choose a method that disables automatic winkin, you may still gain the space and performance benefits of DOs and CRs by winking in DOs manually. The **cleartool winkin** command can wink in a specified DO or, with the **–recurse** parameter, wink in a DO and its sub-DOs. If your team runs nightly builds, an entire project can be winked in (even if PDBs and IDBs were used.) Issue a **cleartool winkin –recurse** *do-pname* command in the development view. You can use view-extended naming to select the *do-pname*. The target and all targets used to build it are winked in to the development view.

Because the **winkin –recurse** command winks in a hierarchy of DOs without regard to the makefile or config spec selections in the current view, run **clearmake** or **omake** after completing the manual winkin to ensure that all DOs are up to date. If the development view selects the same versions of files that were referenced during the nightly build, this build does not require anything to be rebuilt. If development has continued since the nightly build on a subset of the files, only the necessary objects need to be recompiled or relinked.

For more information, see the **winkin** reference page.

Using ClearCase Build Tools with Java

8

The build behavior of Java tools causes various problems for **clearmake** and **omake**. This chapter presents these problems and some possible solutions.

These are the major Java development toolkits:

- The standard Java toolkit is Sun Microsystems' JDK, which includes the **javac** compiler and is available for many platforms.
- Microsoft provides its own extended version of the development kit, including the **jvc.exe** compiler, which is upward compatible with **javac** and has the same behavior. The Microsoft Visual J++ invokes **jvc.exe** and also has the same behavior, but does not use makefile-based building.
- Like Visual J++, Symantec Cafe invokes its own compiler (also called **javac.exe**) underneath. The Symantec compiler is fully compatible with the Sun compiler, and you can set up options in Cafe to use the Sun compiler instead. Cafe uses an internal dependency tracking mechanism to control rebuilds, without an external makefile format.

The remainder of this chapter uses **this compiler** as the example, but the discussion also applies to Microsoft **jvc.exe** and Symantec **javac.exe**.

Using make Tools with javac

Although **javac** handles dependency analysis well, using **javac** by itself misses rebuilds that a **make** tool does not. Specifically, **make** detects modifications of indirect dependencies that **javac** does not. If A.java depends on B.java and B.java depends on C.java, when you change C.java, the command **javac A.java** does not rebuild C.class. Therefore, if you are using **javac** directly, you must recompile each file as you change it.

Many Java applications have some components that are compiled natively or are written in another language. For at least those parts of their applications, developers need makefile-based building.

Using javac with clearmake or omake

There are additional benefits of using **clearmake** or **omake** instead of **make**, especially given the building behavior of **javac**. **clearmake** and **omake** are better at determining when a rebuild is required than are the Java tools, with or without **make**.

When using dynamic views, **clearmake or omake** detects the following rebuild cases that **javac** does not:

- Selection of an older version of a .java file. Because the rebuild decision is based on an older/newer comparison, **javac** does not detect that a rebuild is necessary.

  Note: Clock skew between hosts can cause similar time stamp problems outside ClearCase.

- Change of the **javac** command line. If the command-line options used to build a .class file have changed since the last build, **clearmake** or **omake** rebuilds the .class file. For example, if you add the **–g** option to direct the compiler to rebuild with debugging information, you must invoke the compiler on all your .java files to ensure that they are rebuilt to contain the debugging information.
- Manual winkin of a .class file that is out of sync with, but newer than, the corresponding .java source selected by the view. Because the rebuild decision is based on an older/newer comparison, **javac** does not detect that a rebuild is necessary.

However, some problems may arise when using ClearCase build tools to perform Java builds.

ClearCase Build Problems with javac

ClearCase problems with Java builds relate to conflicts between the dependency analyses of **clearmake** or **omake** and **javac**. **clearmake** is designed to control the build order and the dependency checking between parts of a build system. Java compilers are also designed to provide some of this functionality. When **clearmake** audits build tools that perform their own dependency checking, the result can be unnecessary rebuilds, builds that are never considered up to date, or derived objects that are not winked in. Consequently, the process of managing Java builds with **clearmake** can sometimes produce unexpected results.

For example, if makefiles declare all .class file dependencies accurately, **clearmake** can run the compiler in a sequence such that the compiler never builds more than is necessary. But if .class file dependencies are not declared, the compiler may build files that **clearmake** did not ask it to build. These inaccurate declarations may cause **clearmake** to treat .class targets as out of date, even if the compiler treats them as current. In some cases, **clearmake** may treat a build system as if it is never current. Moreover, the process of maintaining Java makefiles that accurately declare all .class dependencies is both difficult and tedious.

To solve ClearCase build problems with **javac**, you can use a special target called **.JAVAC**.

Using the clearmake makefile Special Target

The **.JAVAC** target enables **clearmake** to use heuristics on audits of Java builds to accurately infer .class dependencies. These dependencies are then stored in .class.dep files, enabling future **clearmake** runs to build .class targets such that each compiler invocation builds only one .class file.

Using .JAVAC causes **clearmake** to take the following actions:

- Examines the audit of any target with the file name extension .class
- Infers the class dependencies from the relative order of .java and .class files that the compiler accesses
- Records these dependencies in .class.dep files
- In subsequent builds, reads .class.dep files to augment the makefiles and build the .class targets in an order that agrees with the compiler's dependency checking and precludes it from building anything not specified on the command line.

Unsupported Builds

clearmake does not support the use of the **.JAVAC** target with nonaudited or parallel builds. The **.JAVAC** special target requires both a dynamic view context and audited builds.

When **clearmake** begins to build two targets in parallel, it does not account for the possibility that two targets may write common files. If they do, the builds may collide when writing those files.

If **.JAVAC** is used with parallel builds, **clearmake** prints the following message:

```
clearmake: Warning: Use of .JAVAC with parallel builds implies
.NOTPARALLEL; ignoring –J
```

Makefile Requirements for .JAVAC

The **.JAVAC** special target must be used with no dependencies and no build script:

```
.JAVAC:
```

clearmake issues a warning and ignores any dependencies or build script.

Other than that, makefiles must use implicit (suffix or pattern) rules. For example:

```
.SUFFIXES: .java .class
.java.class:
        rm -f $@
        $(JAVAC) $(JFLAGS) $<
```

Makefiles may also use pattern rules for compatibility modes that support them. For example:

```
%.class: %.java
        rm -f $@
        $(JAVAC) $(JFLAGS) $?
```

The makefiles must also use absolute paths for .class targets. For example:

all_classes: \proj\src\pkg1\foo.class

clearmake contains a built-in macro function you can use to specify absolute paths more easily:

```
$(javaclasses)
```

For more information about the javaclasses macro, see *Using the javaclasses Built-in Macro*.

The first time **clearmake** builds with **.JAVAC** enabled, **clearmake** is not making use of any inferred knowledge before it builds the first .class target.

After the first build session completes, the .class dependencies are known. As a result, some targets may be rebuilt in the second build session in that view, because **clearmake** is now using the .class.dep files, which may declare .class dependencies that require a different build order. After the second build session completes, the .class DOs are reusable in that view.

Using the javaclasses Built-in Macro

The $(**javaclasses**) function can be used to generate a list of .class targets, from existing .java files. The function takes one or two path arguments.

If your builds write .class files to the same directory as the .java files, use the single-argument form. For example:

```
ALL_CLASSES=$(javaclasses .)
```

If this makefile resides in /vobs/myvob/src, along with A.java and B.java, then the previous command sets ALL_CLASSES to

```
\vobs\myvob\src\A.class \vobs\myvob\src\B.class
```

To provide a way for **clearmake** to evaluate all the .class targets, use this macro in the dependency list of some other target. For example:

```
ALL_CLASSES=$(javaclasses .)
  all : $(ALL_CLASSES)
```

If your builds writes .class files to a different directory from the .java files, use the two-argument form. The first argument is the destination root, an absolute path beneath which your package .class files are written. The second argument is the source root, an absolute path beneath which your package sources reside.

If this makefile resides in \vobs\myvob\src\pkg1, along with files A.java and B.java, and you want the compiler to write the .class files into \vobs\deployvob\pkg1, use a function like this one:

```
ALL_CLASSES=$(javaclasses \vobs\deployvob, \vobs\myvob\src)
```

This sets ALL_CLASSES to

```
\vobs\deployvob\pkg1\A.class \vobs\deployvob\pkg1\B.class
```

A Java compiler that is run from the source directory for Package1 can compile .java files stored in the source directory for Package2. **clearmake** can manage this process if the makefile can always be used to find a rule to build a .class target, regardless of where the target's .java file resides.

To this end, we recommend that you use suffix rules in your makefiles. For example:

```
.JAVAC:
.java.class:
        rm -f $@
        javac $<
```

When using .JAVAC, **clearmake** can use this type of rule to build any .class file and correctly set the value of built-in macros like $@ and $<.

For example, if this makefile and the A.java file reside in /vobs/myvob/src/pkg1, you can instruct the compiler to write the .class files to /vobs/deployvob by using a makefile fragment like this one:

```
.JAVAC:
DEPLOY_ROOT=\vobs\deployvob
  ALL_CLASSES=$(javaclasses \vobs\myvob\src, $(DEPLOY_ROOT))
  all: $(ALL_CLASSES)
.java.class:
        rm -f $@
        javac -d $(DEPLOY_ROOT) $<
```

Deriving Class Dependencies

clearmake stores dependencies it derives in .class.dep files. Each .class target which produces .class DOs has a .class.dep file. For example, for a target named A.class, **clearmake** creates an A.class.dep file. **clearmake** reads .class.dep files when evaluating a target to augment its makefile-declared dependencies.

clearmake derives class dependencies as follows:

- **clearmake** records only direct dependencies in .class.dep files.

  If class A uses B uses C, then **clearmake** records in A.class.dep that A.class depends on B.class, and in B.class.dep that B.class depends on C.class. The transitive dependency of A.class on C.class is implicit in that pair of dependencies and is not explicitly stated in any .class.dep file.

Storing Class Dependencies

clearmake retains derived class dependencies by writing them to .class.dep files, one per .class target, to be used by future **clearmake** build sessions as explicit dependencies because they determine the order of target evaluations.

The .class.dep files are in XML format.

Java Cyclic Class Dependencies

When **clearmake** detects a new .class dependency cycle, it marks the targets as being part of a target group. After **clearmake** forms a target group for a cyclic class dependency, the existing .class DOs from the new target group cannot be used in the view because the build script for those DOs has changed. The build script difference causes the next **clearmake** invocation to rebuild all three DOs.

Using .class.dep Files

clearmake reads the contents of a .class.dep file when it evaluates the dependency list of the corresponding .class target. For example:

```
A.class: A.java
        rm -f A.class
        javac A.java
```

For this makefile, **clearmake** tries to find, open, and parse A.class.dep before it evaluates A.java. The stored dependencies, if any, are evaluated before A.java.

Because .class.dep files are DOs, **clearmake** can wink in these files from other views. The .class.dep files have their own config record, which has a single dependency on the corresponding .java version, and no build script. If two views can compile the same

.java version with different compiler options, and the same class dependencies are derived, a .class.dep file from one view could be winked in to the other.

Note that .class.dep and .class files have no config record relationship.

Dependencies and DO Reuse/Winkin

We recommend that you always delete .class.dep files and .class files together. Avoid deleting only one of these types of files while leaving the other type.

If A.java has been changed, **clearmake** audits a rebuild of A.class to determine whether the A.class.dep file's dependencies have changed. If they have not, A.class.dep can be reused. **clearmake** does not attempt to reuse .class.dep files as DOs.

Note that whenever a .class.dep file can be found in the view, **clearmake** does not shop for it in another view.

clearmake cannot wink in a .class DO if it does not know all of the class dependencies with which the DO was built.

Effects of Nested Classes

A Java class can be declared inside another class. For example:

```
class A {
    class inner {
        [...class definitions...]
    }
    [...class definitions...]
}
```

The compiler writes the code for nested classes into a separate .class file, whose file name is <*parent_class_name*> $ <*nested_class_name*>. In the previous example, the compiler writes A.class and A$inner.class.

clearmake does not create .class.dep files for inner classes. The .class.dep files are intended to declare only the dependencies of one .java file's classes on another .java file's classes. Inner .class files are always siblings. The use of anonymous inner classes, which have a compiler-generated name, is handled in the same way.

.JAVAC in BOS Files

Users can put the .JAVAC special target in build options specification (or BOS) files such as in Makefile.options, rather than in Makefile.

.SIBLINGS_AFFECT_REUSE

By default, **clearmake** does not check sibling DOs when determining whether a target DO in the view can be reused. Using .JAVAC causes **clearmake** to behave as though .SIBLINGS_AFFECT_REUSE was specified in the makefile.

Building Java Applications Successfully without .JAVAC

The following alternatives allow you to successfully build Java applications with **clearmake** or **omake**:

- Write the makefile correctly
- Allow **clearmake** or **omake** to rebuild
- Configure **clearmake** or **omake** makefiles to behave like **make**

The following sections describe each option in detail.

Writing Correct Makefiles

A correctly written makefile results in a correct set of configuration records, which gives you the full power of ClearCase configuration records and winkin without unnecessary rebuilding and without missing rebuilds. You can restructure a makefile to avoid **javac**'s automatic building behavior by enforcing that files on which other files depend are built before their dependents.

Note: **clearmake** and **omake** detect implicit dependencies but cannot determine build order dependencies. If you want files to build in a certain order, you must declare that order in the makefile by adding additional dependencies.

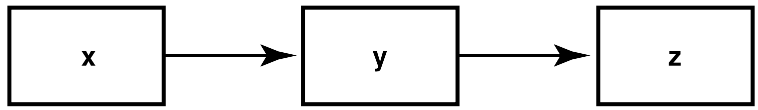
You must take extra care when handling mutually dependent files, because there is not necessarily a correct order for building them. One possibility is to always generate all mutually dependent files as one unit, that is, in one configuration record. You can write the build script for a set of mutually dependent files to delete all class files that correspond to those files before building any of them. This practice ensures that they are not overwritten individually and makes them available as a unit for winkin.

The advantage of writing your makefile correctly is that you avoid extra compilations or rebuilds. No special makefile directives are required, the configuration records have no unusual properties, and winkins work fully. The disadvantage is that the makefile must always be synchronized with the structure and dependencies of the application.

The following sections are makefile examples for applications with particular dependency characteristics.

No Mutually Dependent Files

In this application, classes x, y, and z have a hierarchical dependency graph:



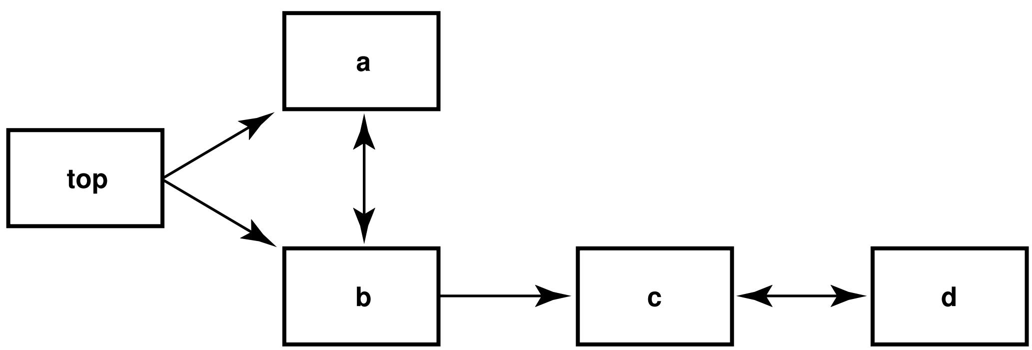
The makefile for such a dependency structure is very simple:

```
.SUFFIXES: .java .class

.java.class:
       javac $<
x.class: y.class
y.class: z.class
```

Mutually Dependent Files

This application consists of classes **top**, **a**, **b**, **c**, and **d**, which have a more complex dependency structure:



The makefile for this dependency structure is somewhat longer, but correct:

```
top.class: a.class b.class
       javac top.java

a.class: b.class

b.class:
     rm -f a.class b.class
     javac a.java b.java

b.class: c.class

c.class: d.class

d.class:
     rm -f c.class d.class
     javac c.java d.java
```

Allowing Rebuilds

If you continue to invoke **clearmake** or **omake** until it determines that all files are up to date, other ClearCase features work correctly. The configuration records record all .class files as implicit dependencies rather than as siblings, which allows winkin to work.

However, the number of rebuilds can become very large if the makefile is written incorrectly. It is possible to map out a correct set of dependencies, as described in *Writing Correct Makefiles* on page 100; it is also possible to request (however inadvertently) that **clearmake** or **omake** build the files in exactly the reverse, and most inefficient, order.

In addition, **clearmake** and **omake**'s default behavior is to ignore modifications to siblings for the purposes of rebuilding. For winkin to work correctly, you must reenable that behavior by using a command-line option or special makefile directive.

Another drawback to this method is that the builds of mutually dependent source files do not fit well, because the files are never up to date. The makefile for these must be written carefully, as described in *Writing Correct Makefiles* on page 100.

Configuring Makefiles to Behave Like make

By using special targets (called directives in **omake**), you can configure your **clearmake** or **omake** makefile so that **clearmake** or **omake** behaves as **make** does with regard to Java builds. The following targets eliminate the extra rebuilding described in *Allowing Rebuilds* on page 102:

`.NOCMP_SCRIPT: %.class` *(omake only)*

`.NO_CMP_SCRIPT: %.class` *(clearmake only)*

```
.DEPENDENCY_IGNORED_FOR_REUSE:
%.class
```

.NOCMP_SCRIPT and **.NO_CMP_SCRIPT** disable build script checking. However, relevant build-script changes are ignored. In addition, **.NOCMP_SCRIPT** and **.NO_CMP_SCRIPT** have no effect during winkin, so even when they are in use, winkins are prevented because of build script differences. Therefore, you must use manual winkins (see the **winkin** reference page) or forego them entirely.

.DEPENDENCY_IGNORED_FOR_REUSE disables the version checking of implicit dependencies when **clearmake** or **omake** is looking for DOs to reuse. This can cause desired rebuilds to be missed, however. One benefit of using **clearmake** or **omake** is automatic dependency detection (for example, of `.h` files in a C build), so it is not desirable to give this up.

To improve the missed implicit dependency checking caused by **.DEPENDENCY_IGNORED_FOR_REUSE**, you can add the missing dependencies as explicit dependencies in the makefile. However, this is a manual process, and you still lose build script checking and winkin. The remaining benefit of using **clearmake** or **omake** is configuration records (though the **catcr** output for them may be confusing).

Java Compilers and Case-Sensitivity Issues

The Microsoft Visual J++ 1.1 compiler (JVC) and the Sun JDK 1.1 compiler (and possibly others) require that the case of file names exactly match the case of the class names in the Java source. Running the MVFS in the Case Insensitive mode recommended on Windows converts view-private file names to lowercase. Instead, use the Case Insensitive, Case Preserving MVFS mode to get the correct behavior for Java builds.

Setting Up a Parallel Build

9

This chapter describes the process of setting up and running parallel builds.

Overview of Parallel Building

Rational ClearCase can perform builds in which multiple processes execute in parallel the build scripts associated with makefile targets. The processes executing the build scripts run on a single host. By using more concurrent processes, parallel builds can reduce the overall build time significantly. Instead of one process running one build script at a time, you can have multiple processors working in parallel. For large software systems, this performance improvement can make a critical difference. For example, you can use parallel builds to enable a build of your entire software system to run overnight and finish before developers and testers arrive for work in the morning.

You start a parallel build the same way as a single-host build: by entering a **clearmake** command. A command-line option or environment variable setting causes the build to run in parallel mode.

A parallel build is controlled by specifications on the host. The host on which you enter the **clearmake** command is the *build client* or *build controller*. On this host, you specify a limit to the number of build scripts to be executed concurrently.

When building in parallel, **clearmake** starts one or more audited build executor (**abe**) processes. An **abe** is a server process invoked by **clearmake** to control and audit execution of a build script during a parallel build. The first time it dispatches a build script to a host, **clearmake** starts an **abe** process there. Subsequent build scripts dispatched to the same host may be executed by the same **abe** process or by a different one.

An **abe** process starts by setting the same view as the calling **clearmake**. It executes a build script dispatched to it in much the same way as **clearmake—**each command in a separate shell process.

All **make** macros are expanded by the build script calling **clearmake**, but environment variables are expanded by the shell process in which a build command runs. This environment combines the **abe** startup environment and the entire environment of the calling **clearmake**. Where there are conflicts, the **abe** setting prevails. To this environment other macros are added:

- Special make macros, such as **MAKEFLAGS** and **MAKEARGS**. These are needed in case the build script invokes **clearmake** recursively.
- Macros assigned in a BOS file or on the **clearmake** command line. These settings are always placed in the build script's environment; they override, if necessary, settings in the environment of the calling **clearmake** or settings in the **abe** startup environment.

The **stdout** and **stderr** output that produce build scripts is sent back to **clearmake**, which stores it in a temporary file. When the build script terminates, **clearmake** prints its accumulated terminal output.

abe returns the exit status of the build script to the calling **clearmake**, which indicates whether the build succeeded or failed. If the build succeeded, **abe** creates derived objects and configuration records.

Note: The **abe** program is started by **clearmake** when needed. Never run it manually.

Each **abe** process sets to the same view and working directory as the **clearmake** process; each process then executes build scripts dispatched to it from the controlling **clearmake** process. A build script runs under **abe** control as if it were executed by **clearmake**, except that **abe** collects terminal output produced by the build script and sends it back to the build controller, where it appears in your **clearmake** window. The **abe** process terminates after waiting three minutes for an initial connection or after waiting three hours for a subsequent response.

Parallel Build Scheduler

clearmake schedules and manages target rebuilds as follows:

- It executes the build script for an out-of-date target as soon after detection as system build resources allow.
- It does not assume that executing a build script for a specific target implies that the target was updated.

clearmake evaluates the dependency graph, beginning with the targets specified on the command line. Before evaluating a specific target, **clearmake** ensures that all dependents of that target have been evaluated and brought up to date. As soon as a target is deemed to be out of date, it is made available for rebuilding. A rebuild is initiated as soon as system resources allow. Depending on the availability of build hosts and load-balancing settings, this may happen immediately or be delayed.

When DO shopping/winkin occurs, **clearmake** postpones DO lookup for any target that has scheduled dependents until the target is encountered in the rebuild logic.

When such a target is detected, **clearmake** then attempts the DO shopping/winkin only when the target's dependencies have completed. This delay eliminates unnecessary rebuilds in serial mode and allows a parallel **clearmake** to initiate rebuilds sooner.

Setting Up the Client Host

There are two issues to consider when you set up client-side processing for parallel builds:

- The number of parallel build processes to request. **clearmake** limits the number of concurrent target rebuilds to the value of the CCASE_CONC environment variable or the number that you specify with the **–J** command-line option. **clearmake** does not start more **abe** processes than specified.
- The limitations and requirements for system loading on the build host. **clearmake** submits as many build scripts as allowed by the **–J** or CCASE_CONC setting.

Starting a Parallel Build

To start a parallel build:

Invoke **clearmake** by using the **–J** command-line option or set the CCASE_CONC environment variable.

For example, to start a build that builds up to five targets concurrently, use one of the following methods:

```
% clearmake –J 5 my_target                (command-line options)
```

```
% setenv CCASE_CONC 5                     (environment variable)
% clearmake my_target
```

Preventing Parallel Builds of Targets

When **clearmake** builds a makefile target, there may be side effects that you cannot address in a makefile. For example, one of your build tools may create temporary files that are not guaranteed to have unique names and then delete them at the end of its processing. When you use this tool serially, there are no problems. However, if you invoke it in multiple parallel builds in **clearmake**, the tool may create identical files and cause the builds to interfere with each other.

You can solve this problem by using the **.NOTPARALLEL** special makefile target. To disable parallel building for a makefile, use this target without any arguments. For example:

.NOTPARALLEL:

To prevent specific targets from being built in parallel with each other, specify them as a set of arguments. Note that parallel builds are prevented only within the set of targets. For example:

.NOTPARALLEL: %.a
.NOTPARALLEL: x.c y.c

In this example, **clearmake** does not build any .a file in parallel with any other .a file, and x is not built in parallel with y. However, **clearmake** can build .a files in parallel with x, y, or any other file.

Preventing Exponential Invocations of abe

If **clearmake** is invoked recursively during a parallel build, the result may be more invocations of **abe** than you want or than the build server can handle. To prevent this situation, use the **.NOTPARALLEL** special makefile target for high-level invocations of **clearmake**.

Index

C

D

E

F

H

I

N

O

P

W