ElectricAccelerator Developer Edition
Installation and Configuration Guide

for version 7.1

Electric Cloud, Inc.
35 South Market Street, Suite 100
San Jose, CA 95113
www.electric-cloud.com

Published 9/22/2014

Contents

Chapter 1: Overview

This guide describes the system requirements necessary to install and run ElectricAccelerator® Developer Edition. Installation walkthroughs are available for each supported platform. Also described are upgrade instructions and initial configuration tasks essential to using the software successfully.

Topics:

About ElectricAccelerator Developer Edition

ElectricAccelerator Developer Edition (Accelerator) is a software build accelerator that dramatically reduces software build times by distributing the build across multiple agents. Using a patented dependency management system, Accelerator identifies and fixes problems in real time that would break traditional parallel builds. Accelerator plugs into existing Make-based infrastructures seamlessly.

ElectricAccelerator Developer Edition enables the use of local agents, which allow developers to make use of extra cores on their own systems, whether or not a cluster is currently available. A local agent is a regular agent that advertises itself to eMake processes on the same machine through a named pipe, instead of talking to a Cluster Manager.

During Accelerator installation, the following components will be installed:

- Electric File System
- ElectricAccelerator Agent
- Electric Make

Electric File System

Electric File System (EFS) is a special-purpose file system driver, monitoring every file access and providing Electric Make with complete usage information. This driver collects dependency information, which allows Electric Make to automatically detect and correct out-of-order build steps. Each EFS driver instance is paired with an ElectricAccelerator Agent. During the installation process, the Agent and EFS are installed at the same time.

ElectricAccelerator Agent

As the user-level component running on the hosts, the Agent and EFS are inseparable—the Agent is an intermediary between Electric Make and EFS. Depending on your system configuration requirements, you may have one EFS/Agent installed per virtual CPU.

Electric Make

Electric Make® (eMake), the main build application, is a new Make version invoked interactively or through build scripts. It reads makefiles in several different formats, including GNU Make and Microsoft NMAKE. Electric Make distributes commands to local agents for execution and services file requests.

Understanding Component Interactions

To a user, Accelerator may appear identical to other Make versions—reading makefiles in several different formats and producing identical results. Using multiple agents for builds is transparent to the Accelerator user.

Important differences in Accelerator build processing versus other distributed systems:

- ElectricAccelerator Developer Edition components work together to achieve faster, more efficient builds. Instead of running a sequential build on a single processor, ElectricAccelerator Developer Edition executes build steps in parallel using multiple local agents.
- For fault tolerance, job results are isolated until the job completes. If an Agent fails during a job, Accelerator discards any partial results it might have produced and reruns the job on a different Agent.
- Missing dependencies discovered at runtime are collected in a history file that updates each time a build is invoked. Accelerator uses this collected data to improve performance of subsequent builds.

Electric Make and EFS

High concurrency levels in Accelerator are enabled by the Electric File System (EFS). When a job such as a compilation runs on a host, it accesses files such as source files and headers through EFS. EFS records detailed file access data for the build and returns that data to Electric Make.

Electric Make acts as a file server for Agents, reading the correct file version from file systems on its machine and passing that information back to the Agents. Agents retain different file version information and do not rely on Electric Make's file sequencing ability to provide the correct version for a job. The Agent receives file data, downloads it into the kernel, notifying EFS, which then completes the original request. At the end of a job, Electric Agent returns any file modifications to Electric Make so it can apply changes to its local file systems.

Chapter 2: Supported Platforms and System Requirements

This section discusses ElectricAccelerator Developer Edition's hardware and system requirements. Make sure you are familiar with all of this section's information ***before*** installing Accelerator.

Topics:

Supported Linux Platforms

Refer to the Release Notes for the latest updates to this information.

ElectricAccelerator Developer Edition currently runs on:

- Red Hat Enterprise Linux Platforms
- SUSE Linux Enterprise Server Platforms
- Ubuntu Platforms

Red Hat Enterprise Linux Platforms

Platform	Notes
RHEL 6.4 (kernel 2.6.32-358) RHEL 6.3 (kernel 2.6.32-279) RHEL 6.2 (kernel 2.6.32-220) RHEL 6.1 (kernel 2.6.32-131) RHEL 6.0 (kernel 2.6.32-71) (32 and 64-bit)	• For 64-bit systems, you must install 32-bit libraries before invoking the installer. • Run `yum install glibc.i686`. If yum server is not present, the required RPMs are `glibc-2.11.1.1.10.el6.i686` and `nss-softokn-freebl-3.12.4-11.el6.i686`. • Be advised of a known Linux kernel issue. • SELinux must be disabled. To disable SELinux, modify `/etc/selinux/config` by changing SELINUX=enforcing to SELINUX=disabled. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.

Platform	Notes
RHEL 5.9 (kernel 2.6.18-348) RHEL 5.8 (kernel 2.6.18-308) RHEL 5.7 (kernel 2.6.18-274) RHEL 5.6 (kernel 2.6.18-238) RHEL 5.5 (kernel 2.6.18-194) RHEL 5.4 (kernel 2.6.18-164) (32 and 64-bit)	• For agent hosts, you must install the corresponding `kernel-devel` version package before invoking the installer. • Be advised of a known Linux kernel issue. • SELinux must be disabled. To disable SELinux, modify `/etc/selinux/config` by changing SELINUX=enforcing to SELINUX=disabled. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.
RHEL 4.9 (kernel 2.6.9-100) RHEL 4.8 (kernel 2.6.9-89) (32 and 64-bit)	• Agent hosts require these three packages: `kernel-devel`, `gcc`, and `gcc-c++`, before invoking the installer. • SELinux must be disabled. To disable SELinux, modify `/etc/selinux/config` by changing SELINUX=enforcing to SELINUX=disabled. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.

SUSE Linux Enterprise Server Platforms

Platform	Notes
SLES 11 SP2 (kernel 3.0.10) SLES 11 SP1 (kernel 2.6.32) SLES 11 (kernel 2.6.27) (32 and 64-bit)	• For agent hosts, you must install the `gcc` and `kernel-source` packages before invoking the installer. • To install `gcc`, run `sudo zypper install gcc` This installs the following: `gcc`, `gcc43`, `glibc-devel`, `linux-kernel-headers`. • To install kernel-source, run `sudo zypper install kernel-source` • SELinux must be disabled. To disable SELinux, modify `/etc/selinux/config` by changing SELINUX=enforcing to SELINUX=disabled. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.

Platform	Notes
SLES 10 SP4 (kernel 2.6.16) (32 and 64-bit)	• SELinux must be disabled. To disable SELinux, modify `/etc/selinux/config` by changing SELINUX=enforcing to SELINUX=disabled. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.

Ubuntu Platforms

Platform	Notes
Ubuntu 12.10 (kernel 3.5) Ubuntu 12.04 (kernel 3.2) (32 and 64-bit)	• For 64-bit systems, you must install 32-bit libraries before invoking the installer. • Run sudo apt-get install ia32-libs to install the 32-bit libraries. • Extended attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an ENOTSUPP ("Operation not supported") error. • For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported. • Be advised of a known Linux kernel issue.
Ubuntu 11.10 (kernel 3.0) Ubuntu 11.04 (kernel 2.6.38) (32 and 64-bit)	
Ubuntu 10.10 (kernel 2.6.35) Ubuntu 10.04.1 (kernel 2.6.32) (32 and 64-bit)	

Known Linux Kernel Issue and ElectricAccelerator Performance

Affected Kernel Versions

- RHEL versions later than 2.6.18-194.32 and earlier than 2.6.32-131.
- Ubuntu versions 2.6.31, 32, 33, and 34

Symptoms

Affected systems may encounter reduced performance on both ext3 and ext4 filesystems. Symptoms may include:

- hung_task_timeout_secs messages in system dmesg logs
- widely variable agent availability (entering and exiting agent "penalty" status frequently)
- contention over the ecagent.state file
- slower builds (with unexplained variances)

To help determine if your environment has this kernel issue, run

```
dmesg | grep hung_task_timeout
```

If hung_task_timeout errors are present, this indicates that a known Linux kernel issue is present. Contact your kernel provider to obtain a different version of the precompiled kernel.

Corrective Actions

For systems running RHEL 5.6, 5.7, 5.8, and 6.0

Consider upgrading to 2.6.32-131 (RHEL 6.1), or downgrading to 2.6.18-194.32 (RHEL 5.5).

For systems running Ubuntu 10.04

Consider upgrading to kernel version 2.6.35 or later.

To install the upstream kernel, do the following, for example (replace * with the version you want to install):

```
sudo add-apt-repository ppa:kernel-ppa/ppa
sudo aptitude update
sudo apt-get install linux-image-2.6.35-*-generic
sudo apt-get install linux-headers-2.6.35-*-generic
sudo apt-get linux-maverick-source-2.6.35 linux-maverick-headers-2.6 (optional)
```

Upgrading the kernel may require you to reinstall various kernel modules such as video drivers, efs, and vm-tools. You can do so by running: apt-get dist-upgrade and then rebooting.

Supported Windows Platforms

Refer to the Release Notes for the latest updates to this information.

ElectricAccelerator Developer Edition currently runs on:

Platform	Notes

Windows 8
(32 or 64-bit)
	• NTFS is required for all Windows machines.
	• For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.
	• To run Electric Make with Visual Studio on Windows, one of the following is required:
	  • Visual Studio 2005 SP1
	  • Visual Studio 2005 SP1 Redistributable Package (if you use Visual Studio .NET 2002 or 2003). Links are provided for your convenience:
	  32-bit
	  64-bit

Platform	Notes

Windows 7
(32 or 64-bit)
	• NTFS is required.
	• For 32-bit, only x86 is supported. For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.
	• To run Electric Make with Visual Studio on Windows, one of the following is required:
	  • Visual Studio 2005 SP1
	  • Visual Studio 2005 SP1 Redistributable Package (if you use Visual Studio .NET 2002 or 2003). Links are provided for your convenience:
	  32-bit
	  64-bit

Platform	Notes

Windows Server 2008 R2
(64-bit only)
	• NTFS is required.
	• For 64-bit, only x86-64 is supported. IA-64 (Itanium) is not supported.
	• 64-bit registry mirroring is supported only if you use a 64-bit Agent/EFS (running on 64-bit Server 2008 R2) with 64-bit eMake (running on any 64-bit Windows platform).
	• To run Electric Make with Visual Studio on Windows, one of the following is required:
	  • Visual Studio 2005 SP1
	  • Visual Studio 2005 SP1 Redistributable Package (if you use Visual Studio .NET 2002 or 2003). Links are provided for your convenience:
	  32-bit
	  64-bit

Windows Server 2003 R2
(32-bit only)
	• SP1 and SP2 required
	• NTFS is required.
	• For 32-bit, only x86 is supported.
	• To run Electric Make with Visual Studio on Windows, one of the following is required:
	  • Visual Studio 2005 SP1
	  • Visual Studio 2005 SP1 Redistributable Package (if you use Visual Studio .NET 2002 or 2003). Links are provided for your convenience:
	  32-bit
	  64-bit

Windows XP SP3
(32-bit only)
	• NTFS is required.
	• For 32-bit, only x86 is supported.
	• To run Electric Make with Visual Studio on Windows, one of the following is required:
	  • Visual Studio 2005 SP1
	  • Visual Studio 2005 SP1 Redistributable Package (if you use Visual Studio .NET 2002 or 2003). Links are provided for your convenience:
	  32-bit
	  64-bit

Supported Third-Party Build Tools

See System Requirements as well.

- GNU Make 3.80 and 3.81
- Microsoft NMAKE 7.x and 8.x
- Symbian Make (Windows)
- Visual Studio .NET 2002, 2003, 2005, 2008, 2010, 2012

  Note: Accelerator can support Visual Studio 6 workspaces if they are exported to NMAKE files. However, every time something changes in the workspace, an export must be done again.

- Rational ClearCase 7.1.1, 7.1.2, and 8.0 (exceptions detailed below)

  Note: ClearCase versions 7.1.2.9 and 8.0.0.5 contained a known missing lib issue. IBM fixed this issue in ClearCase versions 7.1.2.10 and 8.0.0.6 or 8.0.0.7. Electric Cloud recommends using one of those ClearCase versions with Accelerator.

  Accelerator supports building within ClearCase dynamic views and provides the Ledger feature, which can track files that change as a result of a change to the view's configuration specification. (See "ElectricAccelerator Ledger File" in the *ElectricAccelerator Electric Make Users Guide*.)

  Accelerator 7.1 in GNU Make 3.81 emulation mode was tested against the following ClearCase environment:

ClearCase server
RHEL 4.6
ClearCase version 7.0.1 (Wed May 30 17:04:58 EDT 2007) @(#) MVFS version 7.0.1.0 (Wed Apr 11 21:19:21 2007) built at $Date: 2008-10-24.19:30:48 (UTC) $

ClearCase client
RHEL 5.6
ClearCase version 8.0.0.06 (Fri Mar 08 10:35:58 EST 2013) (8.0.0.06.00_2013A.D130307) @(#) MVFS version 8.0.0.6 (Thu Feb 21 05:02:58 2013) built at $Date: 2013-02-21.10:13:08 (UTC) $

ClearCase client
RHEL 5.6
ClearCase version 7.1.2.10 (Fri Mar 08 11:01:39 EST 2013) (7.1.2.10.00_2013A.D130307) @(#) MVFS version 7.1.2.10 (Thu Feb 21 00:53:03 2013) built at $Date: 2013-05-16.22:01:13 (UTC) $

System Requirements

See Supported Third-Party Build Tools as well.

Topics:

Hardware Requirements

The following are ***minimum*** hardware requirements for ElectricAccelerator Developer Edition:

- **Processor** - Pentium 4 (Linux and Windows)
- **Installer disk space** - 300 MB, additional component disk space usage varies and depends on the size of your builds.

Agents use system memory to cache small or unmodified files. For optimal performance, the host machine must have enough memory for your link step, which is typically the largest single build step, plus another 200 MB.

	Memory	Disk Space
Recommended	2 - 3 GB per agent (if your build contains very large build steps)	Free disk space should be at least 3 - 4 times the size of a complete build (input and output).
Minimum	1 GB per agent (machine minimum of 2 GB)	

Cygwin

If you run builds on Windows in Cygwin environments, ensure you have a supported cygwin1.dll version installed:

- 1.7.9
- 1.7.7
- 1.5.25

Install the same version of Cygwin on all agent hosts and eMake machines (if you plan to use an existing Cluster Manager with ElectricAccelerator Developer Edition). Mixing different Cygwin versions (for example, running v1.5 on an eMake machine and v1.7 on agents) is not supported. (In particular, Cygwin versions 1.5 and 1.7 default to different incompatible representations for symbolic links.)

Note: Cygwin version 1.7.x is supported for x = 7 or 9 only. There are known problems for other versions of 1.7.x.

By default, Cygwin 1.7.7 applies overly restrictive permissions to most directories. The permissions prevent the Administrators group from creating new subdirectories and may prevent the agent software from creating new directories to serve as mount points in order to reflect eMake client mount points.

On all agent hosts modify the permissions for the Cygwin installation directory and any other directories under which you want the agent software to dynamically create Cygwin mount points. For agent installations that use standard ECloudInternalUser* accounts, grant the “Administrators” group permission to “create folders / append data.” For custom agent users, grant permission for subdirectory creation to those agent users.

Checksum Utility

An MD5 checksum file is available on the Electric Cloud FTP site. If you choose to verify that the install files are intact and unaltered from their original form and content after you download them, download the corresponding MD5 checksum file also.

MD5 utilities are available for supported operating systems.

- On Linux, verify with `md5sum --check md5.txt`
- Most Linux installations provide an md5sum command for calculating MD5 message digests.
- An MD5 utility for Windows can be downloaded at http://www.fourmilab.ch/md5/.

Chapter 3: Installation

You can install the software using a GUI or an interactive command-line interface (for Linux), or by using a "silent" installation. Not all instructions are the same for each platform. Follow the instructions carefully for your particular platform. Read the procedures thoroughly ***before*** attempting to install, configure, or uninstall Accelerator.

Topics:

Before You Install ElectricAccelerator Developer Edition

Install Location Limitation

Electric Cloud does not support installation on the following:

- NFS
- CIFS
- Samba shares

32-bit vs. 64-bit Information

- The installer for Windows and Linux provides both 32-bit and 64-bit versions of Electric Make.
- The agent installer automatically determines whether to install the 32-bit or 64-bit agent based on the machine architecture. No user action is required.
- The Accelerator installer automatically sets the path to 32-bit. To use 64-bit, you must edit the environment variable path to include the 64-bit bin location *before* the 32-bit location.
- 64-bit executables are installed in a subdirectory of the install location, `<installDir>/64/bin`.

  For example:
  If you install into the Windows `C:\ECLoud\i686_win32` directory, the 64-bit executables are in the `C:\ECloud\i686_win32\64\bin` directory.
  For Linux, the directory is `/opt/ecloud/i686_Linux/64/bin`.

Installing ElectricAccelerator Developer Edition on Linux

Topics:

Additional Linux Install Information

Antivirus Software

Some antivirus software may affect the installer. Turn antivirus software off during installation. If antivirus software is running when you start the installer, you may receive an error dialog. The antivirus software may have reported Accelerator files as a virus and removed them from the temp location. As a workaround, turn off the antivirus software and rerun the installer.

If Symantec AntiVirus software is installed, disable it before installation to avoid serious filesystem conflicts. If Symantec AntiVirus cannot be disabled, put the ECloud directory in an exclusion list or disable the AutoProtect feature. If you need more information, contact Electric Cloud technical support.

umask

Electric Cloud recommends umask 0022. Do not set a different umask during installation.

Using the GUI Installation Method

1. Log in as root.
2. Double-click the `ElectricAccelerator Developer Edition-<version>` installer file to start installation.

   Note: It may take a few minutes to extract the installation packages to your machine before you see the installation wizard.

3. When the Welcome screen appears, click **Next** to continue.
4. If using the ElectricAccelerator installer, select ElectricAccelerator Developer Edition on the Setup Type screen and click **Next**..
5. On the Choose Destination Location screen, accept the default installation directory, or browse to select an alternative directory.

   Note: Avoid selecting an alternative directory that includes spaces in the name. Spaces can create problems when configuring connections with other command-line-based components.

   Click **Next**.

6. If the installer cannot locate a PDF viewer, it will prompt you to browse for one. If you wish, browse for a PDF viewer to use. Click **Next**.
7. On the Electric Agent screen, enter options for the locally installed Electric Agent.

- Enter the number of agents to run—the installer calculates the default number of agents based on the effective number of CPUs on the machine. Default=1, if one CPU is present. If more than one CPU is present, the default value is the number of effective CPUs -1, up to the maximum specified by your license file. If you configure more than the licensed number of agents, the configured number of agents will start, but only the licensed number of agents can participate in requested builds

  Electric Cloud recommends installing no more than (*n* - 1) agents, where n is the number of cores on your machine. If you also have ElectricAccelerator, eMake uses resources communicating with remote agents as well as local ones, so overall performance may be better with less than this number.
- Accept the default agent temporary directory, or choose an alternative.
- Accept the default for the Secure Agent Console Port checkbox if you do not plan to use the secure port, or select the checkbox if you want to use the secure port.
- Accept the default to keep all existing log files, or select the checkbox to Remove old Electric Agent logs.
- Select the checkbox to Reboot host if needed.

Click **Next**.

8. On the Choose ElectricInsight License screen, browse to your ElectricInsight license (if applicable) and click **Next**.
   If you do not supply a license, or you do not have a license installed, a time-limited license is automatically installed. If you have a license already installed, it will continue to be used.
9. On the Choose ElectricAccelerator Developer Edition License screen, browse to your ElectricAccelerator Developer Edition License (if applicable) and click **Next**.
   If you do not supply a license, or you do not have a license installed, a time-limited 4-agent license is automatically installed. If you have a license already installed, it will continue to be used.

   Later, you can supply a different license file if needed. To supply a license, rename the license file "license.xml" and place it in the root install directory:

   - Windows - `C:\ECloud`
   - Linux - `/opt/ecloud`
10. On Windows, select the user accounts you will use to run agents. Accept the default ECloud Internal User or select Existing User. If you select Existing User, click Add User to add a user account. Click **Next**.
11. When the Start Copying Files screen appears, click **Next**.
12. The Installing screen displays while the installation finishes. When installation is finished, the Complete screen displays. Click **Finish**.

    Installation is complete. The installation log file is in the install directory's root.

Using the Interactive Command-Line Installation Method

Note: Agent/EFS builds a kernel module during installation, so you may need to take this into consideration.

IMPORTANT: For RHEL 5, you must install the `kernel-devel` package version that matches the Linux kernel where the modules will be loaded.

1. Log in as root.
2. Run `chmod +x` on the installer (`ElectricAccelerator Developer Edition-<version>`) to ensure it is executable.
3. Run `./<installer filename> --mode` console to start installation.
4. When the welcome message displays, press Enter.
5. Provide configuration information. Accept the defaults or type-in alternatives.
   - Select the location to install ElectricAccelerator Developer Edition. `/opt/ecloud` is the default location.
   - If the installer cannot locate a PDF viewer, it will prompt you to provide the path to one. If you wish, provide the path to a PDF viewer.
   - Select the number of agents to run.
     The installer calculates the default number of agents based on the effective number of CPUs on the machine. Default=1, if one CPU is present. If more than one CPU is present, the default value is the number of effective CPUs -1, up to the maximum specified by your license file.
   - Specify the agent temporary directory. The default is `/tmp`. If you specify a different directory, it must already exist, otherwise the temporary directory defaults to `/tmp`.
   - Decide if you plan to use the Secure Agent Console port.
   - Decide if you want to remove old Electric Agent logs.
   - Decide if you want to reboot the host if needed.
   - Supply the location of the ElectricInsight license file (if applicable).
     If you do not supply a license, or you do not have a license installed, a time-limited license is automatically installed. If a license is already installed, it will continue to be used.
   - Supply the location of the ElectricAccelerator Developer Edition license file.
     If you do not supply a license, or you do not have a license installed, a time-limited 4-agent license is automatically installed. If a license is already installed, it will continue to be used.

   The installer installs ElectricAccelerator Developer Edition using the configuration details you provided, followed by “Installation complete” when the install completes.

   The installation log file is in the install directory’s root, `/opt/ecloud` by default.

Path Settings

ElectricAccelerator Developer Edition is installed in the `/opt/ecloud` directory.

Scripts to add the necessary environment variables are installed in `/opt/ecloud/i686_Linux/conf`.

The scripts are called `ecloud.bash.profile` (for bash shells) or `ecloud.csh.profile` (for csh). You can source the appropriate file in your shell to ensure your PATH includes ElectricAccelerator Developer Edition libraries.

Installing ElectricAccelerator Developer Edition on Windows

Topics:

Additional Windows Install Information

Installer Activity

- For Windows Server 2008 R2, the installer automatically does the following:
  - Disables the Windows error reporting service.
  - Sets HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\NtfsDisableLastAccessUpdate to 0. The default value for Windows Server 2008 R2 is 1.
  - Disables User Account Control (UAC) for 64-bit versions. Disabling UAC avoids popup windows for applications that require administrator privileges. If UAC is enabled, application registry access is redirected to each user's virtual store, even if it runs under the Administrator account.
- If you invoke the installer from a network drive, you may receive an Unknown Publisher security warning. You can disregard this warning and proceed with installation.

Installing from the Cygwin Shell

If you choose to run the installer from the Cygwin shell, be advised of the following:

- Before running the installer, disable UAC, or start the Cygwin shell using the Run as Administrator menu item (right-click the Cygwin desktop icon). Running the installer with UAC enabled may result in a "permission denied" error. This is applicable for all Windows versions that use UAC (Windows 7 and Windows Server 2008 R2).
- You may encounter issues when running the installer from the Cygwin `/tmp` directory. Electric Cloud recommends running the installer from a different directory. This is applicable for all Windows versions.

Antivirus Software

Some antivirus software may affect the installer. Turn antivirus software off during installation. If antivirus software is running when you start the installer, you may receive an error dialog. The antivirus software may have reported Accelerator files as a virus and removed them from the temp location. As a workaround, turn off the antivirus software and rerun the installer.

If Symantec AntiVirus software is installed, disable it before installation to avoid serious filesystem conflicts. If Symantec AntiVirus cannot be disabled, put the ECloud directory in an exclusion list or disable the AutoProtect feature. If you need more information, contact Electric Cloud technical support.

umask

Electric Cloud recommends umask 0022. Do not set a different umask during installation.

Using the GUI Installation Method

1. Log in as Administrator. (You *must* be a member of the Administrator group—Administrator privileges are not sufficient.)
   If you are running *rdp* on this host, ensure *rdp* is in installation mode: `change user/install`.
2. Double-click the `ElectricAccelerator Developer Edition-<version>` installer file to start installation. (For Windows systems running Windows 2008 or later, the administrator user must right-click the installer and select Run as administrator.)

Note: It may take a few minutes to extract the installation packages to your machine before you see the installation wizard. During installation, if you see a Windows security alert pop-up, click **unblock** and continue.

ElectricAccelerator Developer Edition requires the Microsoft Visual C++ 2005 SP1 Redistributable. If it is already installed, select the checkbox. Click **Next** to continue.

3. When the Welcome screen appears, click **Next** to continue.
4. If using the ElectricAccelerator installer, select ElectricAccelerator Developer Edition on the Setup Type screen and click **Next**..
5. On the Choose Destination Location screen, accept the default installation directory, or browse to select an alternative directory.

   Note: Avoid selecting an alternative directory that includes spaces in the name. Spaces can create problems when configuring connections with other command-line-based components.

   Click **Next**.
6. If the installer cannot locate a PDF viewer, it will prompt you to browse for one. If you wish, browse for a PDF viewer to use. Click **Next**.
7. On the Electric Agent screen, enter options for the locally installed Electric Agent.
   - Enter the number of agents to run—the installer calculates the default number of agents based on the effective number of CPUs on the machine. Default=1, if one CPU is present. If more than one CPU is present, the default value is the number of effective CPUs -1, up to the maximum specified by your license file. If you configure more than the licensed number of agents, the configured number of agents will start, but only the licensed number of agents can participate in requested builds

     Electric Cloud recommends installing no more than (*n* - 1) agents, where n is the number of cores on your machine. If you also have ElectricAccelerator, eMake uses resources communicating with remote agents as well as local ones, so overall performance may be better with less than this number.
   - Accept the default agent temporary directory, or choose an alternative.
   - Accept the default for the Secure Agent Console Port checkbox if you do not plan to use the secure port, or select the checkbox if you want to use the secure port.
   - Accept the default to keep all existing log files, or select the checkbox to Remove old Electric Agent logs.
   - Select the checkbox to Reboot host if needed.

   Click **Next**.
8. On the Choose ElectricInsight License screen, browse to your ElectricInsight license (if applicable) and click **Next**.
   If you do not supply a license, or you do not have a license installed, a time-limited license is automatically installed. If you have a license already installed, it will continue to be used.
9. On the Choose ElectricAccelerator Developer Edition License screen, browse to your ElectricAccelerator Developer Edition License (if applicable) and click **Next**.
   If you do not supply a license, or you do not have a license installed, a time-limited 4-agent license is automatically installed. If you have a license already installed, it will continue to be used.

Later, you can supply a different license file if needed. To supply a license, rename the license file "license.xml" and place it in the root install directory:

- Windows - `C:\ECloud`
- Linux - `/opt/ecloud`

10. On Windows, select the user accounts you will use to run agents. Accept the default ECloud Internal User or select Existing User. If you select Existing User, click Add User to add a user account. Click **Next**.
11. When the Start Copying Files screen appears, click **Next**.
12. The Installing screen displays while the installation finishes. When installation is finished, the Complete screen displays. Click **Finish**.

    Installation is complete. The installation log file is in the install directory's root.

Note: Installing ElectricAccelerator Developer Edition may unset the environment variable JAVA_HOME. Reset JAVA_HOME manually.

Silent Install

If you are installing on a series of identical machines, you can use the "silent install" method, which installs components automatically, without user interaction.

Note: On Windows, if you invoke the installer from the command line, you may receive an "Unknown Publisher" security warning. You can disregard this warning and proceed with installation.

Installer Command-Line Options

Use the following command-line options when performing a silent install. The options are the same values a user would normally set through the installer interface. Use this format:

```
<installer filename> [options]
```

This table lists each command-line option's equivalent in the installer UI and the variable that is set in the installer properties file. You can use the resulting properties file for running silent installs.

Note: You can use the values "yes", "y", "1" and "no", "n", "0" interchangeably within installer command-line options.

Command-line option Equivalent installer UI field	Variable set in the installer	Description
`--agentallowreboot <y or n>` Reboot host if needed	`EC_AGENT_REBOOT=y or n`	Indicates if you want to reboot after installing Agent/EFS. Default: n For Windows, if you use n, the installer does not restart the Agent service; reboot the host to ensure EFS works properly. Windows may prompt before the host is rebooted. For UNIX, the machine does not reboot unless required, even if you specify `EC_AGENT_REBOOT=y`.
`--agentnumber [ARG]` Number of Agents	`EC_AGENT_AGENT_NUMBER=1 to n`	Sets the number of Agents to set up on the host. The maximum is the number allowed by your license.
`--agentpassword [ARG]` ECloud Internal User Shared Password	`EC_AGENT_WINUSER_PASSWORD=`	Sets the password for ECloudInternalUser (Windows only).
`--agentremovelogs <y or n>` Remove old Electric Agent logs	`EC_AGENT_REMOVE_LOGS=y or n`	Removes old agent log files. If not, the install appends to them. Default: n
`--agentsecureconsole <y or n>` Secure Agent Console port	`EC_AGENT_SECURE_CONSOLE_PORT= y or n`	y requires an agent-generated key to be entered before commands will run on the agent console port. Default: n
`--agentskipefs <1 or 0>` n/a	n/a	Prevents (y) the installer from installing EFS. Default: 0 (n)
`--agentskiplofs <1 or 0>` n/a	n/a	Prevents (y) the installer from installing LOFS. Default: 0 (n)
`--agenttempdir [ARG]` Agent temporary directory	`EC_AGENT_TEMPDIR=<dir>`	Sets the temporary directory for all agent files. Windows default: C:\WINDOWS\temp UNIX default: /tmp

Command-line option Equivalent installer UI field	Variable set in the installer	Description
`--agentuserlist [ARG]` Existing User	n/a	Sets a list of username/password pairs to be used as login accounts for agents (Windows only).
`--debug` n/a	n/a	Runs the installer in debug mode.
`--debugconsole` n/a	n/a	Runs the installer with the debug console open.
`--finalprefix [ARG]` n/a	n/a	Sets the location where the installed directory tree will be located. Use this option when --prefix is a temporary location that is not the final destination for the product.
`--ignoreoldconf <y or n>` n/a	n/a	Ignores the previous configuration.
`--localagentagentlicensefile` *<browse to location>*	n/a	Sets the location of the ElectricAccelerator Developer Edition license file.
`--mode <console, silent, or standard>` n/a	n/a	Sets the mode in which to run the installer. For a console login, standard and console are identical. For a GUI machine, standard brings up the UI. You can use console in a Unix X Window environment to force the use of console mode.
`--noredist <y or n>` Do not install the redistributables	`EC_BASE_NOREDIST=y or n`	Does not install the Microsoft Visual C++ 2005 SP1 Redistributable (Windows only). Default: n
`--pdfreader [ARG]` *<browse to location>*	`UnixPDFReader=<path to PDF reader>`	Sets the PDF reader to use.

Command-line option Equivalent installer UI field	Variable set in the installer	Description
`--prefix [ARG]` Destination Folder	`InstallDir=<dir>`	Sets the installation directory. Windows default: C:\ECloud UNIX default: /opt/ecloud
`--propertyfile [ARG]` n/a	n/a	Sets the property file from which to read installer options.
`--removezips <1 or 0>` n/a	n/a	Removes (1) zip files after installation.
`--rootrun <1 or 0>` n/a	n/a	Allows the installer to run when root privileges are not present and disables the execution of installer steps that would require root privileges. This option does not change access privileges. 1 (y) or 0 (n).
`--rwprefix [ARG]` n/a	n/a	Specifies the location for read/write files. When using this option for Developer Edition installation, place the license in `<rwprefixdir>/<arch>/tmp` manually if you did not import it during install.
`--skiprun <1 or 0>` n/a	n/a	Prevents (1) the installer from starting the agent
`--temp [ARG]` n/a	n/a	Sets the temporary directory used by this program.
`--test` n/a	n/a	Runs the installer without installing any files.
`--version` n/a	n/a	Displays installer version information.

Creating an Installer Properties File

An installer properties file is a text file that defines installation parameters. These parameters are the same values a user would normally set through the installer interface or command line.

To create an installer properties file:

1. Run an installation with your desired settings.

   This creates a properties file (`install.props`) in the top-level install directory.

2. Use the resulting properties file for subsequent silent installs of the same component type.

The table beginning on page 3-8 details the parameters within installer properties files.

Note: Some variables are unused by ElectricAccelerator Developer Edition. Those variables are not editable and/or may remain empty after installation.

Automating a Linux Silent Install

1. Make sure you already have a properties file (`install.props`) created by a successful installation.
2. Log in to the remote machine as root.
3. Invoke the installer:

   ```
   # ./<installer filename> --mode silent --propertyfile <properties file>
   ```

Automating a Windows Silent Install

1. Make sure you already have a properties file (`install.props`) created by a successful installation.
2. Log in to the remote machine as Administrator.
3. Invoke the installer in a DOS shell:

   ```
   <installer filename> /mode silent /propertyfile <full path\properties file>
   ```

If you are performing a silent upgrade by running the install on the host itself, you may be prompted before the machine is rebooted. This prompt occurs if others are logged in to the machine when you run the agent upgrade.

Chapter 4: Configuration

Configuring ElectricAccelerator Developer Edition properly is important to using the software successfully. The following configuration information will help get you started.

Topics:

Configuring Linux

RHEL 6 and Accelerator Services Started from a Shell

On RHEL 6, if Accelerator services are started from a shell (as opposed to at boot time), you may encounter errors such as

```
java.lang.OutOfMemoryError: unable to create new native thread
```

This type of error occurs because RHEL 6 greatly decreased the default maximum number of threads per user.

As a workaround, comment out the `nproc` line in the `/etc/security/limits.d/90-nproc.conf` file.

RHEL 5 and the kernel-devel Package

On RHEL 5, you must install the `kernel-devel` package version that matches the Linux kernel where the modules will be loaded.

Configuring Windows

All Windows Versions

Disable the Windows error reporting service (if applicable to your Windows version). This avoids popup windows for crashed applications.

Windows 8

On Windows 8, you must disable Admin Approval Mode. Follow these steps:

1. Type `secpol.msc` in the Start Menu and then press Enter.
2. Double-click **Local Policies**, then double-click **Security Options**.
3. Scroll to the bottom of the entry. Locate and double-click

   User Account Control: Run all administrators in Admin Approval Mode

4. Set it to disabled and click **OK**.
5. Reboot the machine.

Note: This configuration is required because of a change in the Windows 8 Admin Approval Mode defaults. Leaving Admin Approval Mode enabled may result in performance degradation.

Applications

It is important that applications are properly initialized for the users that run the agent processes. By default, the users ECloudInternalUser1, ECloudInternalUser2, and so on, own the processes run by agents. Agents can be run as any user, but each agent must have a unique user, which is a requirement for agents to function properly (contact technical support for additional information).

Note: The users ECloudInternalUser1, ECloudInternalUser2, and so on (or the users you choose to run agent processes), must be local administrators on the agent machines.

This requirement imposes additional setup steps because some applications require per user setup (WinZip, Visual Studio 2005 and 2008, Microsoft Office, and so on). Setup is particularly important for applications that display a dialog if they are not properly initialized because this may result in stalled jobs (the application remains in interactive mode until the agent times it out for lack of activity) during the use of eMake.

To initialize applications, do one of the following:

- Use the `psexec`tool available from Microsoft (http://technet.microsoft.com/en-us/sysinternals/bb897553.aspx) to run the application as different users. This requires that you have the password for the account (for ECloudInternalUser1, and so on, contact technical support for the user password).
- Log in as each of the configured agent users and run the relevant application to initialize it.
- Identify which files/registry keys must be set for a user by using Microsoft's procmon tool (http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx), creating a .reg file, and copying the user files to create those entities for all users.

Visual Studio

If you intend to use Accelerator to build Microsoft Visual Studio projects...

Running eccheckvsinst

Use the `eccheckvsinst` utility (located in `C:\ECloud\i686_win32\unsupported`) to check agent installations that will be involved in Visual Studio-based builds.

Before building the Visual Studio project

Before you can use Accelerator to build your Visual Studio project, you must install Visual Studio and then log in and run `devenv` as the user that owns the respective agent processes (usually `ECloudInternalUser1`, `ECloudInternalUser2`, and so on). You can use `psexec` to eliminate the need to log in and log out multiple times. Using this tool is more efficient because Visual Studio stores user settings in the registry and creates files in "My Documents."

If you install Visual Studio ***after*** installing Accelerator, register the Visual Studio add-in by running `install_ecaddin<N>.bat` where N is 70, 71, 80, 90, 100, or 110, depending on your Visual Studio version.

If you are using Visual Studio 2005 or later, reduce the number of parallel builds Visual Studio performs:

1. In Visual Studio, select Tools > Options.
2. In the Options dialog, open Projects and Solutions > Build and Run.
3. Set maximum number of parallel project builds to 1.

The *ElectricAccelerator Visual Studio Integration Guide* contains additional information about using Accelerator with Visual Studio.

Initializing Visual Studio

Use the `psexec` method to initialize Visual Studio as shown:

```
psexec -u ECloudInternalUser1 "C:\Program Files\Microsoft Visual Studio
8\Common7\IDE\devenv.exe"
```

As an alternative, disable profiles for Visual Studio by running this regedit script:

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\8.0\Profile]
"AppidSupportsProfiles"="0"
```

Uninstalling the Visual Studio add-in

Uninstall the add-in by running `uninstall_ecaddin<N>.bat` where <*N*> is 70, 71, 80, 90, 100, or 110, depending on your Visual Studio version. These bat files are in the Accelerator `bin` directory.

Microsoft Office

You must run Microsoft Office by using `psexec` (or logging in directly) because there is no registry setting to initialize Microsoft Office easily.

Ensure that the Visual Basic setting for Security is set to Medium or Lower (assuming the build tries to run VB scripts). Find this under Tools > Options > Security > Macro Security.

WinZip

You must run WinZip by using `psexec` (or logging in directly) because there is no registry setting to initialize WinZip easily.

MSBuild

Specify configuration information for MSBuild under `C:\Program Files\MSBuild`.

Windows Notes

Accelerator may unset the JAVA_HOME environment variable

Installing Accelerator may unset the environment variable JAVA_HOME. Reset JAVA_HOME manually.

Automatic installer actions on Windows Server 2008 R2

- Disables the Windows error reporting service.
- Sets HKLM\SYSTEM\CurrentControlSet\Control\FileSystem\NtfsDisableLastAccessUpdate to 0. The default value for Windows Server 2008 R2 is 1.
- Disables User Account Control (UAC) for 64-bit versions. Disabling UAC avoids popup windows for applications that require administrator privileges. If UAC is enabled, application registry access is redirected to each user's virtual store, even if it runs under the Administrator account.

Windows kernel non-paged pool memory may become depleted

Under high-volume network traffic, the non-paged pool memory in the Windows kernel has the potential to become depleted. This issue in the Windows kernel can be triggered by applications such as MySQL, Java server applications, and so on. Over a period of time, this results in a machine crash.

The workaround is to use the 64-bit version of Windows. Though this does not completely resolve the issue, it increases the available memory to a point where crashes are unlikely and infrequent.

FileInfo and Superfetch services may affect EFS driver performance

The FileInfo and Superfetch services run on Windows 7 (Microsoft officially turned them off in Windows Server 2008 R2).

Because the FileInfo (used by Superfetch) filter driver issues a couple of calls for each file operation in the EFS driver, it has the potential to slow down the EFS driver.

Accelerator turns off the two services by default. You can choose to leave them running by removing the following two lines from runagent (located in `<ECloud install>\<arch>\bin`):

```
catch {service stop sysmain}
catch {service stop fileinfo}
```

and rebooting the machine.

Avoid real-time antivirus scans on agent machines

Real-time scans can slow down builds and can lead to unexpected failures due to issues with the antivirus software's dll injection. Generally, scans do not find anything relevant because all results are also checked on the eMake machine.

Terminating stale processes

Certain processes may continue to run on Windows Agent machines. You can choose to terminate all "stale" processes by adding the following line to runagent (located in <*ECloud install*>\<*arch*>\bin):

```
[efs connect] set terminateStaleProcess 1
```

Support for managed code/.NET

There are no known limitations with respect to building managed code or .NET code. There are, however, areas to keep in mind:

- Agents must have the same version of .NET installed.
- Agents must be on the same service pack level and have the same OS and tool hotfixes installed.
- The language bar must be enabled on all agent machines, or disabled on all agent machines.
- Including the Global Assembly Cache in the eMake root is *not* recommended. Contact technical support for more details.

Registry Information

To allow parallel building of Windows code, Accelerator virtualizes the registry and the file system. The following sections discuss important registry information.

Registry use under Windows

There are two relevant areas of registry use during an Accelerator build. By default, Accelerator virtualizes HKEY_CLASSES_ROOT (except HKEY_CLASSES_ROOT\Installer and HKEY_CLASSES_ROOT\Licenses).

- HKEY_CLASSES_ROOT
- All other keys

HKEY_CLASSES_ROOT

This key contains file name extensions and the COM class registration (http://msdn.microsoft.com/en-us/library/ms724475.aspx and http://technet2.microsoft.com/windowsserver/en/library/dd670c1d-2501-4f32-885b-0c6a1ae662f41033.mspx?mfr=true). Configuration data is stored under the program ids, CLSID, Interface, TypeLib, AppId, and so on.

For entities created during the build, this information must be virtualized to all involved agents.

The following information is registered for a type library:

http://msdn2.microsoft.com/en-us/library/ms221610(VS.85).aspx

```
\TypeLib\{libUUID}
\TypeLib\{libUUID}\major.minor = human_readable_string
\TypeLib\{libUUID}\major.minor\HELPDIR = [helpfile_path]
```

```
\TypeLib\{libUUID}\major.minor\Flags = typelib_flags
\TypeLib\{libUUID}\major.minor\lcid\platform = localized_typelib_filename
```

Other entities that are registered by UUID are registered in different places:

http://msdn2.microsoft.com/en-us/library/ms221150(VS.85).aspx

A ProgID("ApplicationName") maps to and from a CLSID(GUID). The CLSID maps to the actual ActiveX component ("APP.EXE"). The type library is available from the CLSID:

```
\CLSID\TypeLib = {UUID of type library}
\CLSID\{UUID} = human_readable_string
\CLSID\{UUID}\ProgID = AppName.ObjectName.VersionNumber
\CLSID\{UUID}\VersionIndependentProgID = AppName.ObjectName
\CLSID\{UUID}\LocalServer[32] = filepath[/Automation]
\CLSID\{UUID}\InProcServer[32] = filepath[/Automation]
```

http://msdn2.microsoft.com/en-us/library/ms221645(VS.85).aspx

Applications that add interfaces must register the interfaces so OLE can find the appropriate remoting code for interprocess communication. By default, Automation registers dispinterfaces that appear in the .odl file. It also registers remote Automation-compatible interfaces that are not registered elsewhere in the system registry under the label ProxyStubClsid32 (or ProxyStubClsid on 16-bit systems).

The syntax of the information registered for an interface is as follows:

```
\Interface\{UUID} = InterfaceName
\Interface\{UUID}\Typelib = LIBID
\Interface\{UUID}\ProxyStubClsid[32] = CLSID
```

All other keys

Other keys are likely not relevant to the build. HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER, HKEY_USERS, and HKEY_CURRENT_USER are machine specific. If other areas must be virtualized, it is recommended that you add them to the `emake-reg-root` option.

Registry underlay

When a process in the build requests information from the registry, the EFS first checks if the requested key is already present in its cache. If the key is not present, the EFS relays the request to the agent, which in turn sends the request to eMake. After receiving the response from eMake, the agent loads the key into the EFS cache, subject to the following conditions:

- If the key does not exist at all in the local registry on the agent host, the value from the eMake response is used unconditionally.
- If the key exists in the local registry, the value from the local registry is given precedence over the initial value from eMake, but not any value set by prior commands in the build. That is, if the key is modified during the course of the build, the modified value is used in preference of any value from the local registry.

  The order of precedence is (lowest to highest):

  - Value from eMake host registry prior to the start of the build
  - Value from the agent host registry, if any
  - Value set by an earlier job in the build

  The additional checking of precedence enables Accelerator to interoperate with tools that store host-specific licensing information in the registry. If the agent simply used the value from eMake unconditionally in all cases, such tools would fail to operate correctly.

Electric Cloud STRONGLY RECOMMENDS AGAINST running builds locally on agent host machines. Running builds locally on agent machines may add relevant keys to the local machine, which take precedence over the eMake machine's keys. If a key that should come from the eMake machine (such as the typelib information for a lib generated during the build) is already present on the agent due to a locally performed build, the wrong information is used, possibly causing a broken build.

If an agent machine has locally created keys, remove the typelibs that are created during the build from the registry. Any typelib that has an invalid path name associated with it is a likely candidate for an "underlayed" lookup.

Ideally, typelibs created by a build are known. At this point, it is recommended to check for their existence on the host. If an error occurs that indicates the direction of this problem (for example, a library/typelib cannot be found), investigate the failing agent's registry.

ExcludeProcessList registry entry

You can add a multi-string registry value to the agent host inside HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ElectricFS to exclude processes from interfering with EFS and causing a crash. The ExcludeProcessList entry can list processes from McAfee antivirus (for example, Mcshield.exe and mfevtps.exe) or other antivirus software.

Note: Make these registry changes only if the system crashed previously.

Using the accelerator.properties File

The `accelerator.properties` file allows you to modify parameters that affect a range of Accelerator behaviors. The file resides in the following default location:

- Linux: `/opt/ecloud/i686_Linux/conf`
- Windows: `C:\ECloud\i686_win32\conf`

The following topics discuss parameters configurable through the `accelerator.properties` file.

- Changing Log Locations

Restart the Cluster Manager service after modifying the `accelerator.properties` file.

Changing Log Locations

Modify the following parameters to change the location of build logs and accelerator logs:

- `ACCELERATOR_BUILD_LOGS_PATH=build_logs`
- `ACCELERATOR_LOG=logs/accelerator.log`

Changing the Disk Cache Directory and Agent Temporary Storage Location

The same location is used for the disk cache directory and agent temporary storage.

- Use the agent's `ecconfig` command to change one agent:

  `ecconfig -tempdir <newtempdir>`

  Where *<newtempdir>* is the new directory. Specify a full `PATH` to the directory you want to use. Each agent on the host creates a unique subdirectory within the disk cache/temporary storage location, so they do not conflict with each other.

After specifying a different disk cache/temporary storage location, you can switch back to the default location whenever you choose. To do this, use `ecconfig -tempdir` with an empty string [""] as the location.

Default Directories

Installation Location

Platform	Location
Linux	`/opt/ecloud/i686_Linux`
Windows	`C:\ECloud\i686_win32`

Log Files

Platform	Install Log	Agent Log
Linux	`/opt/ecloud/install_$timestamp.log`	`/var/log/ecagent?.log`
Windows	`C:\ECloud\install_ $timestamp.log`	*<ECloud Install>*`\ecagent?.log`

Where:

- ? is the agent number
- *<ECloud Install>* is the installation directory

Disk Cache Directory and Agent Temporary Storage Location

The same location is used for the disk cache directory and agent temporary storage.

Platform	Location
Linux	`/tmp`
Windows	`C:\WINDOWS\Temp`

The most common reason to change this location is due to insufficient disk cache space. To change the location, see Changing the Disk Cache Directory and Agent Temporary Storage Location.

Documentation Roadmap

The following list is an overview of more product information and help.

- *ElectricAccelerator Visual Studio Integration Guide* includes:
  - Building Visual Studio solutions and projects from within the Visual Studio IDE using Electric Make
- *ElectricAccelerator Release Notes* includes:
  - Latest feature changes, fixes, and known issues
  - Install/upgrade notes

- Information on the Electric Cloud Support Web Site (https://electriccloud.zendesk.com/home) includes:
  - Knowledge base articles
  - User forums

Chapter 5: Upgrade

You can upgrade the software using a GUI or an interactive command-line interface (for Linux), or by using a "silent" installation. Not all instructions are the same for each platform. Follow the instructions carefully for your particular platform. Ensure a build is not running ***before*** beginning an upgrade.

Topics:

Upgrading ElectricAccelerator Developer Edition on Linux

Use the GUI Upgrade Method

IMPORTANT: When upgrading, do ***not*** run the installer from a directory path that contains spaces.

The upgrade process is similar to a new installation.

1. Invoke the installer and click **Next** to see the next screen—Electric Agent—which is filled-in with your previous configuration information.
2. Click **Next** to continue to the next screen.
3. When the Start Copying Files screen appears, click **Next**.

   The Installing screen displays while the upgrade proceeds. When the upgrade is finished, the Complete screen displays.
4. Click **Finish**.
5. You may receive a message to reboot the machine after upgrading—rebooting may not be required.

Note: The installer dynamically builds the EFS kernel module if it detects it does not have a prebuilt version matching your Linux kernel version.

The installation log file is in the install directory's root, `/opt/ecloud` by default.

Use the Interactive Command-Line Upgrade Method

1. Log in as root.
2. Run chmod +x on the installer to ensure it is executable.
3. Run `./<installer filename> --mode console` to start the upgrade.
4. Enter configuration details for the upgrade.

   The installer upgrades ElectricAccelerator Developer Edition using the configuration details you entered, followed by "Installation complete" when the upgrade completes.
5. You may receive a message to reboot the machine after upgrading—rebooting may not be required.

Note: The installer dynamically builds the EFS kernel module if it detects it does not have a prebuilt version matching your Linux kernel version.

The installation log file is in the install directory's root, `/opt/ecloud` by default.

Upgrading ElectricAccelerator Developer Edition on Windows

IMPORTANT: When upgrading, do ***not*** run the installer from a directory path that contains spaces.

The upgrade process is similar to a new installation.

1. After invoking the installer, click **Next** to see the next screen—Electric Agent—which is filled-in with your previous configuration information.
2. Click **Next** to continue to the next screen.

If you previously specified an Agent to run as a specific user, the selection is filled-in as you specified.

3. Change the user that the Agent runs as if needed, and click **Next**.
4. When the Start Copying Files screen appears, click **Next**.

   The Installing screen displays while the upgrade proceeds. When the upgrade is finished, the Complete screen displays.
5. Click **Finish**.
6. Reboot the machine after the upgrade.

The installation log file is in the install directory's root, `C:\ECloud` by default.

Chapter 6: Uninstall

If you wish to uninstall the software, follow the procedures described in the following topics.

Topics:

Uninstalling Accelerator on Linux

1. Log in as root.
2. Change to the `tmp` directory by entering `# cd /tmp`
3. Copy the uninstaller `uninstall-accelerator` to the `/tmp` directory by entering:
   `# cp /opt/ecloud/uninstall-accelerator /tmp`
4. Invoke the uninstaller in console mode by entering `./uninstall-accelerator /mode console`

   The system displays the following:

   ```
   This will completely remove ElectricAccelerator from your system.  Are you sure
   you want to do this? [n/Y]
   ```

5. Enter `y` to confirm the uninstall.

   The system displays the following:

   ```
   A full uninstall will remove all leftover files, including other packages such
   as ElectricInsight.

   Perform a full uninstall [y/N]
   ```

6. Enter `y` or `n`.

   No second opportunity to confirm the uninstall request is displayed. The uninstall begins immediately.

   You will see an "uninstall complete" message when Accelerator software is removed.

7. Check the `/etc/sysconfig/ecagent.conf` directory and remove the file `ecagent.conf` if you want to delete *all* Accelerator files.

Uninstalling Accelerator on Windows

1. Go to the Electric Cloud installation directory and run `uninstall-accelerator.exe`.

   Note: For Windows systems running Windows Server 2008 or later, the user must right-click the uninstaller and select Run as administrator.

2. You can choose to perform a full uninstall during the uninstall process. A full uninstall removes all leftover files, including other packages such as Visual Studio Integration and ElectricInsight.

   You will see an "uninstall complete" message when Accelerator software is removed.

Chapter 7: Electric Make Overview

Electric Make (eMake), the main build application in ElectricAccelerator®, is a new Make version invoked interactively or through build scripts. It reads makefiles in several different formats, including GNU Make and Microsoft NMAKE. Electric Make distributes commands to the cluster for remote execution and services file requests.

Topics:

Understanding Component Interactions

To a user, Accelerator may appear identical to other Make versions—reading makefiles in several different formats and producing identical results. Using multiple agents for builds is transparent to the Accelerator user.

Important differences in Accelerator build processing versus other distributed systems:

- ElectricAccelerator Developer Edition components work together to achieve faster, more efficient builds. Instead of running a sequential build on a single processor, ElectricAccelerator Developer Edition executes build steps in parallel using multiple local agents.
- For fault tolerance, job results are isolated until the job completes. If an Agent fails during a job, Accelerator discards any partial results it might have produced and reruns the job on a different Agent.
- Missing dependencies discovered at runtime are collected in a history file that updates each time a build is invoked. Accelerator uses this collected data to improve performance of subsequent builds.

Electric Make and EFS

High concurrency levels in Accelerator are enabled by the Electric File System (EFS). When a job such as a compilation runs on a host, it accesses files such as source files and headers through EFS. EFS records detailed file access data for the build and returns that data to Electric Make.

Electric Make acts as a file server for Agents, reading the correct file version from file systems on its machine and passing that information back to the Agents. Agents retain different file version information and do not rely on Electric Make's file sequencing ability to provide the correct version for a job. The Agent receives file data, downloads it into the kernel, notifying EFS, which then completes the original request. At the end of a job, Electric Agent returns any file modifications to Electric Make so it can apply changes to its local file systems.

ElectricAccelerator Developer Edition Virtualization

ElectricAccelerator Developer Edition is designed to virtualize parts of the build setup so cluster hosts can be configured correctly for the specific build they are executing. Specifically, ElectricAccelerator Developer Edition dynamically mirrors the following host system properties on the Agent:

- Electric File System
- System registry (Windows only)
- User ID (UNIX only)
- Environment variables

Electric File System (EFS)

Files are the most important resource to distribute to hosts. When Electric Make starts, it is given a directory (or list of directories) called `EMAKE_ROOT`. All files in eMake root(s) are automatically mirrored for the build duration. This powerful feature means you simply specify the `EMAKE_ROOT` and the local agents can access files on the host build system.

Almost any file visible to the host build system can be mirrored—regardless of how the host system accesses that file (local disk, network mount, and so on). Both sources and tools can be mirrored, but there are important flexibility/performance "trade-offs" to consider when you include tools in the eMake root. See Configuring Your Build.

When the build completes, EFS is unmounted and the files are no longer visible—this ensures builds do not interfere with host configuration.

Generally, EMAKE_ROOT can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands run by agents. The exceptions are detailed in Setting the Electric Make Root Directory.

System Registry (Windows Only)

A side effect of running builds on Windows: Windows may execute tools that modify the system registry. For example, a build job step may install a program that includes registry modifications that are executed by a subsequent step. If job steps are run on different Agents without registry virtualization, the build may fail because registry modifications by one Agent are not visible by another.

ElectricAccelerator Developer Edition solves this situation by mirroring the Windows registry in addition to the file system. You can specify a registry root using the command line `--emake-reg-roots=<path>`. Just as EMAKE_ROOT specifies a host file system subset to mirror, the registry root specifies which registry keys should be virtualized to Agents (for example, HKEY_LOCAL_MACHINE\Software\My Product). Access and modifications to keys and values under the registry root on Agents are then sent back to the host build system to ensure correct values are propagated to other Agents.

As with the file system, registry mirroring is active during the build only. After the build completes, the registry returns to its original configuration. Processes not running under an Agent on a host cannot see mirrored files and registry entries—they will see only the usual view of the machine.

User Accounts

The same user ID is used by the Electric Agent and the Electric Make process. Using the same user ID ensures processes have the same permission and obtain the same results from system calls (such as getuid) as they would running serially on the user machine. On Windows, processes are executed by the Electric Agent service user.

Environment Variables

When Electric Make sends commands for execution to the Agent, it also sends environment variables. In most cases, this automatic environment virtualization is sufficient. Typically, variables that specify output locations, target architecture, debug-vs.-optimized, and so on, are propagated to tools. By default, environment variables sent by Electric Make override system-wide settings on the host.

For certain variables, override behavior is not desirable. For the build to operate correctly, tools must see the value from the local Agent, not the host build system. For example, the Windows SYSTEMROOT variable depends on the directory where Windows was installed on that machine—values may vary in different Window releases. The SYSTEMROOT value from the host build system is frequently different than the value from the local Agent.

To accommodate these instances, Electric Make allows you to exclude specific environment variables from virtualization. In addition to variables you may explicitly choose to exclude, Electric Make automatically excludes the TMP, TEMP, and TMPDIR variables on all platforms; and the COMSPEC and SYSTEMROOT variables on Windows. For a complete description of environment variables, see Electric Make Command-Line Options and Environment Variables

Understanding Build Parts

Software builds are often complex systems with many inputs that must be carefully set up and configured. Correctly distributing system processes over multiple agents requires efficient replication of relevant parts of the build setup.

ElectricAccelerator Developer Edition software is designed to help virtualize your build environment for each local agent so distributed jobs display the same behavior as compared to how those jobs run serially.

Because input configurations can vary from one build to the next, ElectricAccelerator Developer Edition virtualization is active only while the build is running. When the build completes, the host returns to its original state.

Chapter 8: Setting Up ElectricAccelerator Developer Edition

Before invoking Electric Make, you need to set up ElectricAccelerator Developer Edition to ensure accurate, reliable builds.

Topics:

Defining Your Build

Before using Accelerator, precisely define which components go into your build. Generally, a build has three input types:

SOURCES + TOOLS + ENVIRONMENT = BUILD

Build Sources

Build sources include all compiled or packaged files to build your product. For example, these sources include:

- Intermediate files generated during the build (for example, headers or IDL files)
- Makefiles and scripts that control the build
- Third-party source files read during the build
- Any file read by the build in source control

Build Tools

Tools used to create build output include make, compilers, linkers, and anything else operating on the ources during the build:

- Executable post-process tools (for example, *strip*)
- Static analyzers (for example, *lint*) if they run as part of the build
- Packaging tools (for example, *tar* or *zip*)

Sometimes the distinction between sources and tools is blurred. Consider these examples:

- A utility executable compiled from your sources during the build, run during later steps, but not part of final output
- Header source files that are part of the compiler (for example, `stdio.h)` or a third-party package, but not under source control

In this context, *sources* are those files that may change from one build to the next. Thus, a utility executable compiled from your sources as part of the build is considered a source.

By contrast, *tools* change infrequently—tools are often configured once and served from a central location (for example, an NFS share). A standard header such as `stdio.h` is usually considered part of the tool suite.

The distinction between inputs that can change between builds (sources) and inputs that can safely be assumed to be constant (tools) becomes important when configuring Accelerator virtualization.

Build Environment

Your operating system environment is an essential part of your build. The operating system is easy to overlook because the environment usually is configured once per host and then ignored as long as builds function normally. It is important to identify which parts of the operating system could affect the build. Some inputs to consider include:

- User environment variables
- System registry (Windows only)
- Operating system version (including patches or service packs)

- User account and user permissions
- Host-specific attributes (for example, machine name, network configuration)

For each of these inputs, consider what impact (if any) they will have on the build.

Generally, some environment variables require correct settings for a build (for example, `PATH` or variables that specify output architecture or source/output file locations).

Another common example of how the operating system can affect the build occurs with the use of tools that require license management. If a tool license (for example, a compiler) is host-locked or it requires contacting a license server to operate, ensure the compiler on the host can acquire the license also.

Configuring Your Build

After defining your build environment and identifying all of its inputs, configure your host and Electric Make so the system correctly virtualizes:

1. Set the Electric Make root directory.
2. Determine if you need to do additional configuration for tools.
3. Make additional configurations regarding the registry. (Windows only)
4. Configure environment variables if needed.
5. Set Electric Make emulation.
6. Enable local agents.

Some cases of virtualization and/or distribution of specific job steps are not desirable. For these cases you can configure Accelerator to:

- Run an individual command from the Agent back on the host system, using the “proxy command” function. See Using the Proxy Command.
- Prevent remote job execution by using the `#pragma runlocal` function. See Running a Local Job on the Make Machine.

Chapter 9: Electric Make Basics

The following topics discuss basic Electric Make information to get you up and running.

Topics:

Invoking eMake

The Electric Make executable is called `emake`. The most important change to your build process is to ensure this executable is invoked in place of the existing Make.

For interactive command-line use, ensure that:

- the ElectricAccelerator Developer Edition `bin` directory is in your `PATH` environment variable:
  - For Linux: `/opt/ecloud/i686_Linux/bin` or `/opt/ecloud/i686_Linux/64/bin`
  - For Windows: `C:\ecloud\i686_win32\bin` or `C:\ecloud\i686_win32\64\bin`
- you type `emake` in place of gmake or nmake.

You can rename the eMake executable to either gmake or nmake because Electric Make checks the executable to determine which emulation to use. If the name of the submake is hard-coded in many places within your makefiles, a simple solution would be to rename gmake or nmake to *gmake.old* or *nmake.old*, and rename eMake to either *gmake* or *nmake*. In this way, you can maintain access to your existing make, but all submakes from an Accelerator build will correctly use Electric Make.

Note: Electric Cloud does ***not*** recommend running builds in /tmp.

To ensure eMake is called for recursive submake invocations in makefiles, use the `$(MAKE)` macro for specifying submakes instead of hard-coding references to the tool. For example, instead of using:

```
libs:
	make -C lib
```

use the following `$(MAKE)` macro:

```
libs:
	$(MAKE) -C lib
```

Single Make Invocation

It is important to keep the build in a single Make invocation. At many sites, Make is not directly invoked to do a build. Instead, a wrapper script or harness is used to invoke Make, and users (or other scripts) invoke this wrapper. The wrapper script may take its own arguments and may perform both special set up or tear down (checking out sources, setting environment variables, post-processing errors, and so on). Because Electric Make behaves almost exactly like native Make tools, usually it can directly replace the existing makefile in wrapper scripts.

Sometimes, however, the script may invoke more than one Make instance. For example, the script could iterate over project subdirectories or build different product variants. In this case, each of these builds becomes a separate Accelerator build, with its own build ID, history file, and so on.

It is much more efficient for Make instances that are logically part of one build to be grouped under the control of a single parent Make invocation. In this way, Electric Make can track dependencies between submakes, ensure maximal parallelization and file caching, and manage the build as a single, cohesive unit.

If your build script invokes more than one submake, consider reorganizing makefile targets so a single Make is invoked that in turn calls Make recursively for submakes.

If a lot of the setup for each instance occurs within the build script, another possible solution is to use a simple top level makefile to wrap the build script; for example,

```
all:
	my-build-harness ...
```

In this instance, `my-build-harness` runs on the Agent much like any other command and sends commands discovered by submake stubs back to the host build machine. This approach works only if each submake's

output is not directly read by the script between Make invocations. Otherwise, it may be susceptible to submake stub output problems. See Invoking eMake for more information.

Setting the Electric Make Root Directory

The `--emake-root` option (or the `EMAKE_ROOT` environment variable) specifies the EFS root directory [or directories] location. All files under the Electric Make root directory are automatically mirrored on each Agent.

Electric Make uses the current directory as the default if no other root directory is specified. You must specify the correct root directory [or directories] or the build may fail because Electric Make cannot find the necessary files to complete the build or resolve dependencies.

For best results and performance, be specific when setting the Electric Make root location. Be sure to include:

- All files created or modified by the build.
- All source files.
- The location where build output files will go during the build, for example, object files, linker output, and so on.
- Other files read by the build such as third-party tools and system headers, or other files not modified if you need to. Be aware, however, that including these files can slow performance. See Configuring Your Build.

  If necessary, specify more than one directory or subdirectory. Separate each location using standard `PATH` variable syntax (a colon for UNIX, a semicolon for Windows).

  UNIX example:

  ```
  --emake-root=/src/foo:/src/baz
  ```

  In this example, you have streamlined the root path by excluding other `/src` subdirectories not necessary to the build.

  Windows example:

  ```
  --emake-root=C:\Build2;C:\Build4_test
  ```

Note: Any files used by the build, but not included under an Electric Make root directory, must be preloaded onto all hosts and identical to corresponding files on the system running Electric Make. If these files are not identical, Electric Make could find and use the wrong files for the build. This approach is appropriate for system compilers, libraries, header files, and other files that change infrequently and are used for all builds.

Generally, `EMAKE_ROOT` can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands running on agents. However, there are a few exceptions:

- `EMAKE_ROOT` cannot be set to the system root directory (for example, "`/`" on UNIX or `C:/` on Windows). It may be tempting to try this to specify "mirror everything," but in practice, this is not desirable because mirroring system directories such as `/var` or `/etc` on UNIX or `C:/Windows` on Windows can lead to unpredictable behavior. eMake will not allow you to specify the root directory as `EMAKE_ROOT`.
- `/tmp` and the Windows temp directory cannot be included in the eMake root.
- On Windows, another operating system restriction is imposed:
  `EMAKE_ROOT` is not a UNC path specification—it must be a drive letter specification or a path relative to a drive letter. It must also be a minimum of three characters.

Configuring Tools

Agents must be able to execute all tools (compilers, linkers, and so on) required during the build. Tools that are available on the host build system are available to agents at the same locations in the virtualized environment. On Windows, tools must be globally accessible because agents run as a different user than the one running the build. On UNIX, the agents impersonate the user running the build.

In some special cases, it may be necessary to use tools outside of the virtualized file system. To support running such tools on the local build machine during a cluster build, see Using the Proxy Command.

IMPORTANT: Electric Cloud does not recommend Proxy Command for frequent use; use it with caution.

Tools that Access or Modify the System Registry

In addition to files, tools on Windows may access or modify the system registry. Use the `--emake-reg-roots` command-line option to specify a key to mirror. You can specify more than one key by separating multiple entries with semicolons:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
HKEY_LOCAL_MACHINE\Software\Bar
```

In addition, you can specify exception keys to not mirror the system registry by prefixing the key with a "–" character. For example:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
-HKEY_LOCAL_MACHINE\Software\Foo\Base
```

which means "mirror all keys and values under `Foo` except the keys in `Foo\Base`.

To ensure compatibility with Microsoft Visual Studio, the registry root specification automatically includes:

```
HKEY_CLASSES_ROOT;-HKEY_CLASSES_ROOT\Installer;
-HKEY_CLASSES_ROOT\Licenses
```

Configuring Environment Variables

If your build environment is fairly constant, you may want to use *environment variables* to avoid respecifying values for each build. Environment variables function the same way as command-line options, but command-line options take precedence over environment variables.

Because Electric Make environment variables are propagated automatically to Agents, most commands running on an Agent will run with the correct environment without modifications.

However, two important exceptions exist.

- As described in Configuring Tools, if tools for the agents are installed in different locations from the host system, the `PATH` variable (and other variables that reference tool locations) on the build system must be modified to include the locations for the agent tools.
- Generally, differences between the build system and agents may indicate it is undesirable to override environment variables with eMake values. In this case, use the `--emake-exclude-env` command-line option or the `EMAKE_EXCLUDE_ENV` environment variable.

  For example, consider a build system environment variable called `LICENSE_SERVER` that normally contains the license server name that the system should contact to obtain a tools license. If this variable is machine-specific, eMake overrides the correct machine-specific value on the cluster hosts with the value from the build system. To ensure eMake does not override `LICENSE_SERVER` with the value from the build system, use the option, `--emake-exclude-env`, when running eMake:

  ```
  --emake-exclude-env=LICENSE_SERVER
  ```

You can specify more than one value by separating them with commas:

```
--emake-exclude-env=LICENSE_SERVER,TOOLS_SERVER
```

Some variables are almost always used to describe the local machine state, so Electric Make always excludes them from mirroring. These variables are:

```
TMP
TEMP
TMPDIR
```

For Windows, this list also includes:

```
COMSPEC
SYSTEMROOT
```

Configuring ccache

The only configuration required to use ccache with Accelerator is to set the `CCACHE_NOSTATS` environment variable. If you do not set this environment variable, the entire build becomes serialized because ccache continuously writes to a statistics file throughout the build. To learn more about ccache, refer to http://ccache.samba.org/.

Setting Electric Make Emulation

Electric Make can emulate different make variants: GNU Make (gmake), gmake in a Cygwin environment, and Microsoft NMAKE (nmake).

By default, the `--emake-emulation=<mode>` is set to `gmake`, which supports of a subset of GNU Make 3.81 (see Unsupported GNU Make Options and Features).

Available modes:

- `gmake`
- `gmake3.82`
- `gmake3.81`
- `gmake3.80`
- `nmake`
- `nmake8`
- `nmake7`
- `cygwin`
- `symbian`

Note: You can rename `emake.exe` to `nmake.exe` or `gmake.exe` to change the emulation type for all builds automatically. See the `--emake-emulation-table` option in Electric Make Command-Line Options and Environment Variables.

Windows

If you are using nmake, override the default setting from `gmake` to `nmake`:

```
emake --emake-emulation=nmake
```

This is a Windows-specific setting that must be set prior to invoking a build.

Cygwin

If you are using Cygwin, override the default setting from `gmake` to `cygwin`:

```
emake --emake-emulation=cygwin.
```

Enabling ElectricAccelerator Developer Edition Agents

An ElectricAccelerator Developer Edition (local) Agent is a regular agent that advertises itself to eMake processes on the same machine through a named pipe, instead of talking to a Cluster Manager.

Enabling Local Agents

To enable eMake to use local agents:

- Add the `--emake-localagents=y` option to the `emake` invocation.

or

- Add the `--emake-localagents=y` to the `EMAKEFLAGS` environment variable.

Running Multiple eMakes

By default, the first build request gets all agents and no other build request can proceed.

To change this behavior, you must explicitly instruct the first build's eMake to:

- Limit its use of local agents (with `--emake-maxlocalagents`)

or

- Release local agents occasionally (with `--emake-yield-localagents`) so they can be used by another eMake that is looking for local agents, or by the current eMake if there are no other such eMakes running

If you want to run multiple simultaneous eMakes on your machine, note that no load balancing occurs.

Using Local Agents with Cluster Agents (Hybrid Mode)

If you have ElectricAccelerator Developer Edition and an ElectricAccelerator Cluster Manager, you can specify `--emake-localagents=y` ***and*** `--emake-cm`. This is referred to as hybrid mode. Hybrid mode instructs eMake to use local agents and cluster agents, if available.

Local Agent eMake Command-line Arguments

Argument	Description
`--emake-localagents=y`	Instructs eMake to use any available local agents. Without an ElectricAccelerator Cluster Manager, eMake will use local agents only. If you specify `--emake-localagents=y` but not `--emake-cm`, eMake uses local agents only. If you specify `--emake-cm` but not `--emake-localagents=y`, eMake uses cluster agents only. Specifying both options instructs eMake to use both, if available.

Argument	Description
`--emake-maxlocalagents=<N>`	Limits the number of local agents used. N=0 uses all available agents (default 0).
`--emake-yield-localagents=<N,T>`	If using more than `N` local agents, then eMake releases the number of agents over `N` every `T` seconds so they can be used by another eMake that is requesting local agents.

Electric Make Command-Line Options and Environment Variables

You can configure Electric Make options from the command line for a specific build and/or use Electric Make environment variables to set persistent options.

A few caveats for using these option types:

- The environment variable `EMAKEFLAGS` can be used to set any command-line option. For example, this emake invocation:

  ```
  % emake --emake-root=/home/joe
  ```

  is equivalent to the following in `csh`:

  ```
  % setenv EMAKEFLAGS " --emake-root=/home/joe"% emake
  ```

  The `bash` equivalent is:

  ```
  $ export "EMAKEFLAGS=--emake-root=/home/joe"$ emake
  ```

  and in a Windows command shell:

  ```
  C:\> set EMAKEFLAGS=--emake-root=C:\home\joe
  ```

- The hierarchy or precedence for setting an Electric Make option is:
  - Command-line options
  - `EMAKEFLAGS`
  - Environment variables

Using command-line options to set Electric Make values overrides values set using both `EMAKEFLAGS` and environment variables. Using `EMAKEFLAGS` to set options overrides the use of environment variables.

Command-line options are listed in alphabetical order except for platform-specific options that are listed *after* platform-independent options. Debug options are listed at the end of the table. Some options and environment variables are applicable to ElectricAccelerator only, not ElectricAccelerator Developer Edition.

Note: The `--emake-volatile` command-line option is deprecated and no longer has any effect. If the option is specified, it is ignored.

Command-line Options	Environment Variables	Description
	EMAKE_BUILD_MODE	Always set to local. Specifies that an individual emake invocation on the Agent does not enter stub mode, but instead behaves like a local (non-cluster) make. Electric Make automatically uses local mode when the -n switch is specified.
--emake-annodetail=*var1[,var2[,...]]*		Specifies the level of detail to include in annotation output—a comma separated list for any of the following values: basic: Basic annotation history: Serialization details file: Files read or written lookup: All file names accessed waiting: Jobs that waited registry: updates to registry env: enhanced environment variables
--emake-annofile=<*file*>		Specifies the name of the XML-formatted log output file. By default, the annotation file, emake.xml, is created in the directory where eMake is run. If specified, implies at least “basic” annotation details. The following macros are available: @ECLOUD_BUILD_ID@ expands into the unique eMake build ID. @ECLOUD_BUILD_DATE@ expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD. @ECLOUD_BUILD_TIME@ expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS. Example: --emake-annofile=annofile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml results in: annofile-4-20090220-184128.xml
--emake-autodepend=<*0|1*>		Enables (1) or disables (0=default) the eDepend feature.
--emake-big-file-size=<N>		Sets the minimum file size (in bytes) to send through Agent-to-Agent transfers for direct file sharing between hosts. Default=10KB.

Command-line Options	Environment Variables	Description
`--emake-build-label=<label>`	`ECLOUD_BUILD_LABEL`	Sets a customized build label. These labels are literal strings and do not use available tags when defining labels for build classes.
`--emake-clearcase=var1[,var2[,...]]`	`EMAKE_CLEARCASE`	Turns on support for specified ClearCase features—a comma separated list of any of the following values: `symlink` : symbolic links `vobs` : per-VOB caching (for speed) `rofs` : read-only file system
`--emake-collapse=<0|1>`		Turns history collapsing on or off. When collapsing is enabled, dependencies between a single or several jobs in another *make* instance are replaced with a serialization between the job and the other Make instance. This action typically results in significant history file size reduction, but may cause some over-serialization. In most builds, this has little or no impact on build time. In some builds, disabling collapsing improves performance at the cost of increased history file size. Default=1 (on)
`--emake-disable-pragma=var1[,var2[,...]]`		Comma separated list of pragma directives to ignore—can be one or more of: `allserial`, `runlocal`, `noautodep`, or `all` to disable all pragmas.
`--emake-disable-variable-pruning=<0|1>`		Disables variable table pruning. Default=`0` (off)
`--emake-emulation=<mode>`	`EMAKE_EMULATION`	Sets Make-type emulation to mode. Default emulation type is `gmake`. You can rename `emake.exe` to `nmake.exe` or `gmake.exe` to change the emulation type for all builds automatically. Available modes: `gmake`, `gmake3.80`, `gmake3.81`, `gmake3.82`, `symbian`, `nmake`, `nmake7`, `nmake8`, or `cygwin`. If specifying `gmake3.82`, read the GNU Make 3.82 Note.
`--emake-emulation-table=<table>`		Configures default emulation modes for Make programs. TABLE is a comma separated list of NAME=MODE, where NAME is the name of a Make executable and MODE is the emulation mode to use if `emake` is invoked as NAME.
`--emake-exclude-env=var1[,var2[,...]]`	`EMAKE_EXCLUDE_ENV`	Specifies which environment variables must not be replicated to the hosts.

Command-line Options	Environment Variables	Description
`--emake-hide-warning=<list>`	`EMAKE_HIDE_WARNING`	Hides one or more Accelerator-generated warning numbers. List is a comma-separated list of numbers you want to hide.
`--emake-history=<read|create|merge>`		Specifies the history mode creation model. Default=`merge`.
`--emake-history-force=<0|1>`		Honors history mode even if the build fails. Default=`1` (on)

`--emake-historyfile=<path/file>`		Specifies which history file to use for a specific build. Allows you to change the default name and path for the history file `emake.data` set automatically by Electric Make.
		The following macros are available:
		`@ECLOUD_BUILD_ID@` expands into the unique eMake build ID.
		`@ECLOUD_BUILD_DATE@` expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD.
		`@ECLOUD_BUILD_TIME@` expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS.
		Example:
		`--emake-historyfile=historyfile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml`
		results in:
		`historyfile-4-20090220-184128.xml`

`--emake-job-limit=<N>`		Limits the maximum number of uncommitted jobs to `N` where `0` means unlimited. Default=`0`
`--emake-ledger=<valuelist>`	`EMAKE_LEDGER`	Enables the ledger capability. `Valuelist` is a comma-separated list that includes one or more of: `timestamp`, `size`, `command`, and `nobackup`.
`--emake-ledgerfile=<path/file>`	`EMAKE_LEDGERFILE`	The name of the ledger file. Default=`emake.ledger`
`--emake-mem-limit=<N>`		Controls the amount of memory Electric Make devotes to uncommitted jobs. When the limit is exceeded, Electric Make stops parsing new Make instances. Default=`1,000,000,000` (1 GB).

Command-line Options	Environment Variables	Description
`--emake-mergestreams=<0/1>`	`EMAKE_MERGE_STREAMS`	Indicates whether to merge the `stdout/stderr` output streams, yes (`1`) or no (`0`). The default is merge the streams (`1`). For most situations, this is the correct value. If you re-direct standard output and standard error separately, specify no (`0`) for this option.
`--emake-monitor <hostname/IP>:<port>`		Sets the hostname/IP and port of the system from where you want to view the ElectricInsight live monitor data. To monitor live build data, you must launch the ElectricInsight live monitor *before* you start the build.
`--emake-pedantic=<0|1>`		Turns *pedantic mode* on (`1`) or off (`0`=default). When pedantic mode is on, warnings appear when invalid switches are used, or potential problems are identified (for example, rules with no targets or reading from a variable that was not written). When pedantic mode is off, Electric Make ignores irrelevant switches or exits without warning if it encounters unresolvable errors.
`--emake-read-only=<path>`	`EMAKE_READ_ONLY`	All paths starting at the directories specified in `--emake-read-only` will be marked as read-only file systems when they are accessed on the agent. On UNIX, any attempt to create new files or write to existing files under those directories will fail with EROFS, “Read-only file system”. On Windows, it will fail with ERROR_ACCESS_DENIED, “Access is denied”.
`--emake-readdir-conflicts=<0|1>`		Explicitly enables conflict detection on directory read operations (commonly called “glob conflicts,” which is but one manifestation of the problem). Allowed values are `0` (disabled, the default value) and `1` (enabled). If your build is susceptible to readdir conflict failures, you can enable these checks and get a correct build even if you do not conduct a single-agent build. The resulting history file is identical to a single-agent build result. Though the initial run with this feature may be over-serialized (a consequence of readdir conflicts), a good history file allows builds to go full speed, without conflicts, the next time. You do ***not*** want to enable this option all the time. Correct usage: Enable it for one run if you suspect a globbing problem, and then disable it, but use the history file generated by the previous run. Another possible strategy to use if you are not familiar with the build you are building, is to enable the option until you get a successful build, and then disable it after you have a complete, good history file.

Command-line Options	Environment Variables	Description
`--emake-remake-limit=<N>`		This option defaults to `10`. If set to `0`, makefiles are not added as goals at all and no remaking occurs. Setting the value to `1` is equivalent to the deprecated `--emake-multiemake=0`.
`--emake-resource=<resource>`	`EMAKE_RESOURCE`	The resource requirement for this build. This option overrides the build class resource request setting set on the Cluster Manager.
`--emake-root=<path>`	`EMAKE_ROOT`	Specifies the Electric Make root directory(s) location. Particularly on Windows, this parameter should not be used to virtualize your tool chain. The semi-colon is the delimiter between drives. Example: `build@winbuild-cm$ emake --emake-cm=winbuild-cm --emake-emulation=cygwin --emake-root=/c/cygwin/tmp;/c/tmp` `Starting build: 867` `make: Nothing to be done for `foo'.` `Finished build: 867 Duration: 0:00 (m:s)` In this example, the C: drive is mounted on /c
`--emake-showinfo=<0|1>`		Turns build information reporting on (`1`=default) or off (`0`). Information includes build time to completion.
`--emake-tmpdir=<path>`	`EMAKE_TMPDIR`	Sets the Electric Make file temporary directory.
Dependency Optimization and Parse Avoidance Commands		
`--emake-assetdir=<path>`		Use the specified directory for assets such as saved dependency information and cached parse results. The default directory is `.emake`.
`--emake-optimize-deps=<0|1>`		Use the saved dependency information file for a makefile when dependencies are the same and save new dependency information when appropriate.
`--emake-parse-avoidance=<0|1>`		Avoid parsing makefiles when prior parse results remain up-to-date and cache new parse results when appropriate.

Command-line Options	Environment Variables	Description
`--emake-parse-avoidance-ignore-env=<var>`		Ignore the named environment variable when searching for applicable cached parse results. To ignore more than one variable, use this option multiple times.
`--emake-parse-avoidance-ignore-path=<path>`		Ignore this file or directory when checking whether cached parse results are up-to-date. Append % for prefix matching. To ignore more than one path or prefix, use this option multiple times.
`--emake-suppress-include=<pattern>`		Skip matching makefile includes (such as generated dependencies). Generally, you should not suppress makefile includes unless they are generated dependency files, and you have enabled automatic dependencies as an alternative way of handling dependencies. **Note:** If the pattern does not have a directory separator, then the pattern is compared to the include's filename component only. If the pattern has a directory separator, then the pattern is taken relative to the same working directory that applies to the include directive and compared to the included file's entire path.
UNIX-specific Commands		
	`ECLOUD_ICONV_LOCALE`	Allows you to set the iconv locale. Use `iconv -l` to list the available locales. **Usage Note:** If you receive an emake assertion failure that contains information similar to: `emake: ../util/StringUtilities.h:406: std::string to_utf8(const std::string&): Assertion `c != iconv_t(-1)' failed.` This could mean that your system is missing an internationalization package, or the locale package has a different name for ISO 8859-1. (This particular issue is due to the conversion between 8-bit strings and UTF-8.) Example: Your system recognizes “ISO8859-1”, “ISO_8859-1”, and “ISO-8859-1” only. Set `ECLOUD_ICONV_LOCALE` to one of those valid locale names.
Windows-specific Commands		

Command-line Options	Environment Variables	Description
`--emake-case-sensitive=<0|1>`		Sets case sensitivity for target and pattern name matching. This is inherited by all submakes in the build. The option applies when using gmake/cygwin emulation modes only; nmake and symbian modes are not affected by this option. Default for gmake and cygwin is on (`1`). This option replaces `--case-sensitive`.
`--emake-cygwin=<Y|N|A>`	`EMAKE_CYGWIN`	`Y`=requires *cygwin1.dll* `N`=ignore *cygwin1.dll* `A`=use *cygwin1.dll* if available Default=`A`, if launched from a Cygwin shell, if not, then `N`. Default=`Y`, if eMake `emulation=cygwin` was set.
`--emake-ignore-cygwin-mounts=<mounts>`	`EMAKE_IGNORE_CYGWIN_MOUNTS`	Comma separated list of Cygwin mounts to ignore. Unless listed, Cygwin mount points are replicated on the Agent.
`--emake-reg-limit=<N>`		Limits the number of registry keys sent automatically for each key. Default=`50`.
`--emake-reg-roots=<path>`		Sets the registry virtualization path on Windows machines. The syntax is `--emake-reg-roots=path [;path]`. Do not use this parameter to virtualize your tool chain.
Debug Commands		
`--emake-debug=<value>`	`EMAKE_DEBUG`	Sets the local debug log level(s). For a list of possible values, see the `emake --help` message.

Command-line Options	Environment Variables	Description

`--emake-logfile=<file>`	`EMAKE_LOGFILE`	Sets the debug log file name.
		Default is `stderr`.
		The following macros are available:
		`@ECLOUD_BUILD_ID@` expands into the unique eMake build ID.
		`@ECLOUD_BUILD_DATE@` expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD.
		`@ECLOUD_BUILD_TIME@` expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS.
		Example:
		`--emake-logfile=logfile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml`
		results in:
		`logfile-4-20090220-184128.xml`

`--emake-rdebug=<value>`	`EMAKE_RDEBUG`	Sets the remote debug log level(s). For a list of possible values, see the `emake --help` message. Enabling this option disables parse avoidance.
`--emake-rlogdir=<dir>`	`EMAKE_RLOGDIR`	Sets the directory for remote debug logs.
Local Agent Commands		
`--emake-localagents=y`		Instructs eMake to use any available local agents. Without an ElectricAccelerator Cluster Manager, eMake will use local agents only. If you specify `--emake-localagents=y` but not `--emake-cm`, eMake uses local agents only. If you specify `--emake-cm` but not `--emake-localagents=y`, eMake uses cluster agents only. If you specify both, eMake uses both, if available.
`--emake-maxlocalagents=<N>`		Limits the number of local agents used. N=0 uses all available agents (default 0).
`--emake-yield-localagents=<N,T>`		If using more than *N* local agents, eMake releases the number of agents over *N* every *T* seconds.

Hybrid Mode Sample Build

After all ElectricAccelerator components and ElectricAccelerator Developer Edition are installed and you are familiar with the concepts, try a test build. Using a text editor, create a makefile with the following content:

UNIX

```
all: aa bb cc
aa:
       @echo building aa
       @sleep 10
bb:
       @echo building bb
       @sleep 10
cc:
       @echo building cc
       @sleep 10
```

Windows

```
SLEEP=ping -n 10 -w 1000 localhost>NUL
all: aa bb cc
aa:
        @echo building aa
        -$(SLEEP)
bb:
        @echo building bb
        -$(SLEEP)
cc:
        @echo building cc
        -$(SLEEP)
```

Note: “ping” is used in the Windows example because Windows does not have a SLEEP utility.

If you were to run this file with GNU Make, you would expect it to finish in approximately 30 seconds—allowing for each 10-second command to run serially. Running against an ElectricAccelerator cluster and ElectricAccelerator Developer Edition Agents, the commands run in parallel allowing the build to complete much faster.

To start this sample build:

1. Specify the Cluster Manager by using the `--emake-cm=<host>` option. The Cluster Manager is responsible for assigning Agents to Electric Make for processing jobs. The example uses “linuxbuilder” as the Cluster Manager host.
2. Make sure the Electric Make root directory [or directories] specification includes all directories that contain source or input files required by the build. In the example, the only source file is the makefile, which is in the same directory where Electric Make is invoked. Because the default emake root is the current directory, `--emake-root=<path>` is not needed.
3. Enable ElectricAcclerator Developer Edition (local) Agents by adding the `--emake-localagents=y` option to the `emake` invocation.

```
% emake --emake-cm=linuxbuilder --emake-localagents=y

      Starting build: 1
      building aa
      building bb
```

```
building cc
Finished build: 1  Duration: 0:11(m.s) Cluster availability: 100%
```

Cluster availability: 100% indicates the cluster was fully available for the build duration. For more information on cluster sharing and the cluster availability metric, see Annotation.

ElectricAccelerator Developer Edition Sample Build

After ElectricAccelerator Developer Edition is installed and you are familiar with the concepts, try a test build. Using a text editor, create a makefile with the following content:

UNIX

```
all: aa bb cc
aa:
       @echo building aa
       @sleep 10
bb:
       @echo building bb
       @sleep 10
cc:
       @echo building cc
       @sleep 10
```

Windows

```
SLEEP=ping -n 10 -w 1000 localhost>NUL
all: aa bb cc
aa:
        @echo building aa
        -$(SLEEP)
bb:
        @echo building bb
        -$(SLEEP)
cc:
        @echo building cc
        -$(SLEEP)
```

Note: “ping” is used in the Windows example because Windows does not have a SLEEP utility.

If you were to run this file with GNU Make, you would expect it to finish in approximately 30 seconds—allowing for each 10-second command to run serially. Running with at least three Agents, the commands run in parallel allowing the build to complete much faster.

To start this sample build:

1. Make sure the Electric Make root directory [or directories] specification includes all directories that contain source or input files required by the build. In the example, the only source file is the makefile, which is in the same directory where Electric Make is invoked. Because the default emake root is the current directory, `--emake-root=<path>` is not needed.
2. Enable ElectricAcclerator Developer Edition (local) Agents by adding the `--emake-localagents=y` option to the `emake` invocation.

```
% emake --emake-localagents=y
```

```
Starting build: 1
building aa
building bb
```

```
building cc
Finished build: 1  Duration: 0:11(m.s)
```

Chapter 10: Additional Electric Make Settings and Features

The following topics discuss additional eMake settings and features.

Topics:

Using the Proxy Command

Normally, Electric Make sends commands to the Agent for execution. In some cases, however, it may not be desirable or possible to execute a particular command on the Agent.

Proxy Command Location

During installation, the `proxyCmd` binary is installed on every host:

- Linux: `/opt/ecloud/i686_Linux/bin/proxyCmd`
- Windows: `C:\ECloud\i686_win32\bin\proxyCmd.exe`

When invoked by the Agent, it is: `proxyCmd <program> <arg1> ...`

ElectricAccelerator Developer Edition executes <*program*> on the host build system and proxies the result back to the Agent so it can continue remote execution.

Determine If You Can Use the Proxy Command

You can use the “proxy command” feature to run a command locally on the Electric Make machine *if both of the following are true*:

- You have a command that cannot run on the Agent, either because it returns incorrect results or because it is not available.
- The command in question *does not read or write build sources or output*—it only makes external [outside of the build] read-only queries.

The second item is particularly important because the command runs on the host build machine, outside of the virtualized file system. Because ElectricAccelerator Developer Edition cannot track the activity of this process, the dependency and conflict resolution mechanisms that prevent build output corruption are circumvented. It is also important that the process is read-only [that the command make no changes to whatever system it is querying] because in parallel builds with conflicts, Electric Make could rerun the job producing unintended side effects.

Note: For these reasons, it is important to use the “proxy command” only when necessary.

Proxy Command Example 1

The simplest *safe* use of the `proxyCmd` is a source control system query. For example, a particular build step queries the source control system for branch identification using a tool called `getbranch` that it embeds in a version string:

```
foo.o:
     gcc -c -DBRANCH=`getbranch` foo.c
```

It is preferable to avoid installing and configuring a full deployment of the source control system on the host when only this simple query command is needed.

In the following example, the `proxyCmd` provides an efficient solution. By replacing `getbranch` with `proxyCmd getbranch`, you avoid having to install the `getbranch` tool and its associated components on the host:

```
foo.o:
    gcc -c -DBRANCH=`proxyCmd getbranch` foo.c
```

Proxy Command Example 2

A less invasive implementation that does not require makefile modifications and allows compatibility with non-ElectricAccelerator Developer Edition builds is to create a link on all agent machines using the name of the tool [for example, 'getbranch'] found on the eMake host to `proxyCmd`. For Windows operating systems that do not

support `symlinks`, use the `copy` command. When invoked under a different name, `proxyCmd` knows to treat the linked name as the <*program*> to execute:

```
# ln -s /opt/ecloud/i686_Linux/bin/proxyCmd /usr/bin/getbranch
```

Using Subbuilds

The Electric Make subbuild feature is designed to help speed up component builds through intelligent build avoidance. Currently, the subbuild feature's scope includes the following use case:

```
Makefile:
-------------
.PHONY: a b
all: a b
       @echo all
a b:
       $(MAKE) -C $@

a/Makefile
-------------
all: a.lib
       @echo a

a.lib:
       @touch $@

b/Makefile
-------------
all: ../a/a.lib
       @echo b
```

Explanation: If something from 'a' changes, and you are building from 'b', the only way to pick up the new `a.lib` is to build from the top level directory. With subbuilds, you know `b`'s dependencies so you can build those dependencies directly without having to build *everything* from the top level directory.

The subbuild database must be built beforehand to make the dependency list available without having to parse any Makefiles that are not in the current directory.

The following sections describe how to use subbuilds. Refer to Subbuild Limitations for additional information about subbuild limitations.

Subbuild Database Generation

The following command runs your build as normal and also generates a subbuild database with the name `emake.subbuild.db`.

```
emake --emake-gen-subbuild-db=1 --emake-root=<path> --emake-subbuild-
db=emake.subbuild.db
```

Where <`path`> is the Electric Make root directory setting.

`--emake-root` is required for agent builds.

`--emake-subbuild-db` is optional. If it is missing, the default `emake.subbuild.db` name is used.

Run a Build Using Subbuild

The following command runs a build with subbuild information:

```
emake --emake-subbuild-db=emake.subbuild.db
```

Specify `--emake-subbuild-db=<file>` to run a build with subbuild information. When you invoke eMake with the `--emake-subbuild-db` option, it uses the dependencies extracted from the makefile and the subbuild database to determine which build components are prerequisites of the desired current make, then rebuilds those components before proceeding as normal.

When you specify `--emake-subbuild-db=<file>`, do ***not*** specify `--emake-gen-subbuild-db`, otherwise eMake regenerates the database.

Subbuild Limitations

- There is no incremental building of the database. Each time you change something in a makefile in your build, you must rebuild the database by doing a full build.
- The database is not currently optimized for size. This may result in an extremely large database for very large builds.
- Subbuilds do not provide additional gains in non-recursive make builds.
- Because of the manner in which subbuilds are currently scheduled, there is interleaving output for the "Entering directory..." and "Leaving directory..." messages.

  For example: If a subbuild database was built for the following build:

```
Makefile:
-------------
.PHONY: a b
all: a b
a b:
       $(MAKE) -C $@

a/Makefile
----------------
all: a.lib
a.lib:
       echo aaa > $@

b/Makefile:
----------------
all: ../a/a.lib
       echo b
```

When you proceed to build just 'b' (maybe with "`emake -C b`") and `a/a.lib` is missing, you receive "entering directory a" after "entering directory b", even though 'a' is supposed to be built before 'b'.

```
make: Entering directory 'b'
make -C a
make[1]: Entering directory 'a'
echo aaa > a.lib
make[1]: Leaving directory 'a'
echo b
b
make: Leaving directory 'b'
```

Information Applying to Local Builds Only

- Rules to build a sub-directory's output files must not overlap.
  For example: The rule to build `sub1/foo.o` must appear in `sub1/Makefile` only and not `sub2/Makefile`. Default suffix rules can cause eMake to find a way to build `sub1/foo.o` while trying to

build `sub2`. In this situation, adding "`.SUFFIXES:`" to `sub2/Makefile` can resolve the issue.

- Subbuilds require that the build be componentized to some degree.
- Subbuilds require that you have practiced "good hygiene" in your build tree—there must be explicit dependencies mentioned in the component makefiles.

  For example: If a build has two components, `foo` and `bar`, where `foo` produces a library `foo.dll` and `bar` uses that library, the rule might be written to produce `bar.exe` such as this in `bar/Makefile`:

  ```
  bar.exe: $(BAR_OBJS)
         link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
  ```

  For subbuilds to work (in local mode), it must be modified as in the following:

  ```
  bar.exe: $(BAR_OBJS) $(OUTDIR)/foo/foo.dll
         link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
  ```

  Note that it is explicitly stated that `bar.exe` requires `foo.dll`. Also note that it is NOT required to have a rule to build `foo.dll` in `bar/Makefile`.

  There cannot be ANY rule at all to build `$(OUTDIR)/foo/foo.dll` in `bar/Makefile`, explicit or implicit, otherwise you will get the wrong information for building `foo/foo.dll` in the subbuilds database. The subbuilds database currently allows updates to existing entries while building the database.

Extension for Building Multiple Targets Simultaneously (GNU Make emulation only)

Sometimes you may want one rule that creates multiple outputs simultaneously, and using a pattern rule may not be a suitable solution.

ElectricAccelerator provides the `#pragma multi` directive to enable you to create non-pattern rules that have multiple outputs.

`#pragma multi` causes the targets of the immediately following rule or dependency specification to be treated as updated together if/when they are updated. For example, the following produces one rule and one rule job only, rather than three of each:

```
#pragma multi
a b c: ; @echo building a b and c
```

Important Notes

- If you apply `#pragma multi` to a target list, then you must apply it to all overlapping target lists, and those lists must specify the same set of targets (though they may do so in a different order).
- Target-specific and pattern-specific variable assignments for the targets of a `#pragma multi` rule must agree; otherwise it is unspecified whether eMake chooses the assignments for just one target, or somehow combines them all.
- You may not apply `#pragma multi` to static patterns, double-colon rules, or pattern targets that follow non-pattern targets on the same line.
- If you apply `#pragma multi` to a non-static pattern, a warning will issue.
- A `#pragma multi` rule with commands may not override or be overridden by other commands for the same targets.

- A `#pragma multi` dependency specification must correspond to a `#pragma multi` rule with commands having the same set of targets. Otherwise, eMake will error out. Implicit rules are not searched to find such missing commands.
- `$@` has the same meaning as it does in multiple-target patterns: the target that first caused the rule to be needed.
- Setting either of the following disables `#pragma multi`: `--emake-disable-pragma=multi` or `--emake-disable-pragma=all`.

Stopping a Build

You can terminate an in-progress build by pressing **Ctrl-C**.

Chapter 11: Make Compatibility

ElectricAccelerator Developer Edition is designed to be completely compatible with existing Make variants it emulates. There are, however, some differences in behavior that may require changes to makefiles or scripts included in the build.

Almost all GNU Make and NMAKE options are valid for use with ElectricAccelerator Developer Edition. However, ElectricAccelerator Developer Edition does not support some GNU Make and NMAKE options.

The following topics documents those differences and what actions to take to ensure your build is compatible with ElectricAccelerator Developer Edition.

Topics:

Unsupported GNU Make Options and Features

Unsupported GNU Make Options

Option	eMake response if specified
`-d (debug)`	error message
`-j (run parallel)`	ignored
`-l (load limit)`	ignored
`-o (old file)`	error message
`-p (print database)`	error message
`-q (question)`	error message
`-t (touch)`	error message

Unsupported GNU Make 3.81 Features

Electric Make does not support the following GNU Make 3.81 features:

- Though `$(eval)` is allowed in rule bodies, any variables created by the `$(eval)` exist only in the scope of the rule being processed, ***not*** at the global scope as in GNU Make. For example, the following is not supported:

```
all:
        $(eval foo: bar)
```

- Using `$*` for an archive member target

GNU Make 3.82 Note

Support for multi-line variable definition types when GNU Make 3.82 emulation is enabled; other GNU Make 3.82 features are not yet supported

Unsupported NMAKE Options

```
/C (suppress output)
/Q (check timestamps, don't build)
/T (update timestamps, don't build)
/NOLOGO (suppresses NMAKE copyright message)
/ERRORREPORT (sends information to Microsoft)
```

Commands that Read from the Console

When GNU Make or NMAKE invokes a makefile target command, that process inherits the standard I/O streams of the Make process. It is then possible to invoke commands that expect input during a build, either from the terminal or passed into the Make standard input stream. For example, a makefile such as:

```
all:
        @cat
```

could be invoked like this:

```
% echo hello | gmake
hello
```

More commonly, a command in a makefile might prompt the user for some input, particularly if the command encounters an error or warning condition:

```
all:
        @rm -i destroy
```

```
% gmake
rm: remove regular file `destroy'? y
```

Neither of these constructs is generally recommended because systems that require run-time user input are tedious to invoke and extremely difficult to automate.

Makefiles (such as the examples above) with commands expecting interactive input are not supported because processes do not inherit the I/O streams and console of the parent eMake.

In the majority of cases, tools that prompt for console input contain options to disable interactive prompting and proceed automatically. For example, invoking "`rm`" without the "`i`" enables this behavior. For those that do not, explicitly feeding expected input (either from a file or directly by shell redirection or piping) will suffice. For example:

```
all:
echo y | rm -i destroy
```

Finally, tools such as Expect (see http://expect.nist.gov) that automate an interactive session can be used for commands that insist on reading from a console.

Transactional Command Output

Electric Make simulates a serial GNU Make or NMAKE build. Though Electric Make runs many commands in parallel, the command output (including text written to standard streams and changes to the file system, such as creating or updating files) appears serially in the original order without overlapping. This feature is called *transactional output* (or "serial order execution") and is unique to Accelerator among parallel build systems. This feature ensures standard output streams and underlying file systems always reflect a consistent build execution state, regardless of how many jobs Electric Make is actually running concurrently.

Transactional output is achieved by buffering the results of every command until the output from all preceding commands is written. Buffering means that while the output *contents* on the standard streams matches GNU Make or NMAKE exactly, the *timing* of its appearance may be a little unexpected. For example:

- **"Bursty" output** – One of the first things you notice when running a build with Accelerator is that it appears to proceed in bursts, with many jobs finishing in quick succession followed by pauses. This type of output is normal during a highly parallel build because many later jobs may have completed and output is ready to be written as soon as longer, earlier jobs complete. The system remains busy, continuously running jobs throughout the build duration, even if the output appears to have paused momentarily.
- **Output follows job completion** – GNU Make and NMAKE print commands they are executing before they are invoked. Because Electric Make is running many commands in parallel and buffering results to ensure transactional output, command-line text appears with the output from the command *after* the command has completed. For example, the last command printed on standard output is the job that just completed, not the one currently running.

- **Batch output** – As a way to provide feedback to the user during a long-running execution, some commands may write to standard output continuously during their execution. Typically, these commands may print a series of ellipses or hash marks to indicate progress or may write status messages to standard error as they run. More commonly, a job may have several long-running commands separated with echo statements to report on progress during build execution:

For example, consider a rule that uses `rsync` to deploy output:

```
install:
        @echo "Copying output into destination"
        rsync -vr $(OUTPUT) $(DESTINATION)
        @echo "done"
```

With GNU Make, users first see the `Copying output` echo, then the state information from `rsync` as it builds the file list, copies files, and finally, they see the `done` echo as the job completes.

With Electric Make, all output from this job step appears instantaneously in one burst when the job completes. By the time any output from `echo` or `rsync` is visible, the entire job has completed.

Stubbed Submake Output

Recursive Makes (also called *submakes* because they are invoked by a parent or *top-level* Make instance) are often used to partition a build into smaller modules. While submakes simplify the build system by allowing individual components to be built as autonomous units, they can introduce new problems because they fragment the dependency tree. Because the top-level Make does not coordinate with the submake—it is just another command—it is unable to track targets across Make instance boundaries. For a discussion of submake problems, see “Recursive Make Considered Harmful” by Peter Miller (http://miller.emu.id.au/pmiller/books/rmch/)

Submakes are particularly problematic for parallel builds because a build composed of separate Make instances is unable to control target serialization and concurrency between them. For example, consider a build divided into two phases: a `libs` submake that creates shared libraries followed by `apps` that builds executables that link against those libraries. A typical top-level makefile that controls this type of build might look like this:

```
all: makelibs makeapps
makelibs:
       $(MAKE) -C libs
makeapps:
       $(MAKE) -C apps
```

This type of makefile works fine for a serialized make, but running in parallel it can quickly become trouble:

- If `makelibs` and `makeapps` run concurrently (as the makefile “all” rule implies), link steps in the `apps` Make instance may fail if they prematurely attempt to read `libs` generated libraries. Worse, they may link against existing, out-of-date library copies, producing incorrect output without error. This is a failure to correctly serialize dependent targets.
- Alternatively, if `apps` is forced to wait until `libs` completes, even `apps` targets that do not depend on `libs` (for example, all the compilation steps, which are likely the bulk of the build) are serialized unnecessarily. This is a failure to maximize concurrency.

Also important to note: Submakes are often spawned indirectly from a script instead of by makefile commands,

```
makelibs:
       # 'do-libs' is a script that will invoke 'make'
       do-libs
```

which can make it difficult for a Make system to identify submake invocations, let alone attempt to ensure their correct, concurrent execution.

These problems are exacerbated with distributed parallel builds because each make invocation is running on anAgent.

Correct, highly concurrent parallel builds require a single, global dependency tree. Short of re-architecting a build with submakes into a single Make instance, this is very difficult to achieve with existing Make tools.

An ideal solution to parallelizing submakes has the following properties:

- maximizes concurrency, even across make instances
- serializes jobs that depend on output from other jobs
- minimizes changes to the existing system (in particular, does not require eliminating submakes or prohibit their invocation from scripts)

Submake Stubs

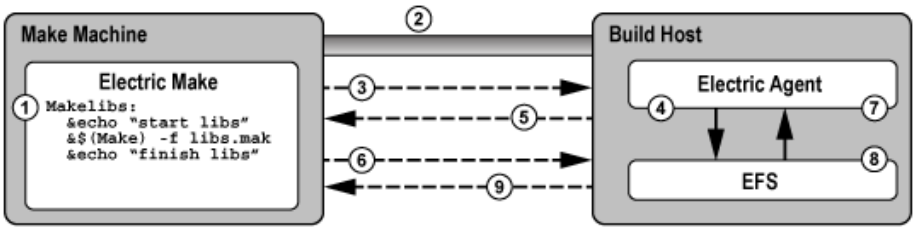
ElectricAccelerator Developer Edition solves the parallel submake problem by introducing submake stubs — Electric Make dispatches *all* commands, regardless of tool (compiler, packager, submake, script, and so on) to Agents. After the Agent executes a command, it sends the results (output to standard streams and exit status) back to Electric Make, which then sends the next job command.

If the command run by the Agent invokes Electric Make (either directly by the expanded `$(MAKE)` variable or indirectly through a script that calls `emake`), a new top-level build is *not* started. Instead, an Electric Make process started on the Agent enters *stub* mode and it simply records details of its invocation (current working directory, command-line, environment, and so on) and immediately exits with status **0** (success) without writing output or reading any makefiles. The Agent then passes invocation details recorded by the stub back to the main Electric Make process on the host build machine, which starts a new Make instance and integrates its targets (which run in parallel just like any other job) into the global dependency tree. Commands that follow a submake invocation are logically in a separate job serialized after the last job in the submake. In the illustrations, the build host and the Make machine are the same machine.

The following example illustrates this process:

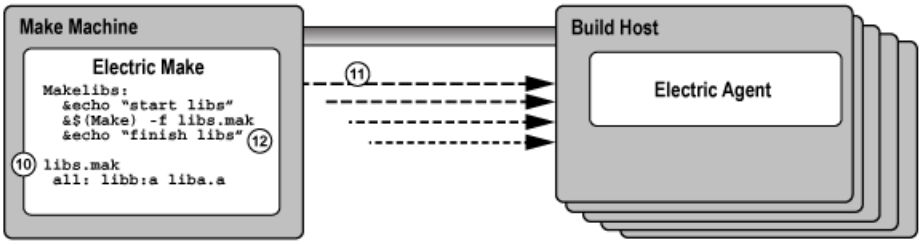
```
...

Makelibs:
        @echo "start libs"
        @$(MAKE) -C libs
        @echo "finish libs"
```

This example is diagramed in steps as shown in the following illustrations.

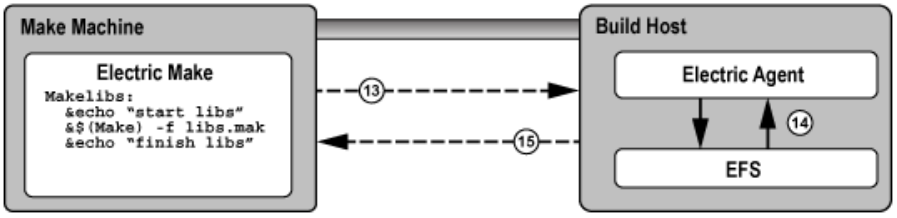


1. Electric Make determines that `Makelibs` target needs to be built.
2. Electric Make connects to Electric Agent.
3. Electric Make sends the first command, `echo "start libs"`.
4. The Agent invokes the command, `echo "start libs"`, and captures the result.

5. The Agent returns the result, "`start libs`", to Electric Make.
6. Electric Make sends the second command, `emake -f libs.mak`
7. The Agent runs the second command, `emake -f libs.mak`
8. `emake` enters stub mode and records the current working directory, command, and environment, then exits with no output and status code **0**
9. Agent returns the stub mode result and a special message stating a new Make was invoked with recorded properties (Electric Make).



10. Electric Make starts a new Make instance, reads the makefile, `libs.mak`, and integrates the file into the dependency tree.
11. New jobs are created and run to build all targets in the `libs.mak` makefile.
12. Electric Make splits the last command in the `Makelibs` target, `echo "finish libs"`, into a continuation job that is defined to have a serial build order later than the last job in the submake, but there is no explicit dependency created that requires it to run later than any of the jobs in the submake. This means that it may run in parallel (or even before) the jobs in the submake, but if for some reason that is not safe, Electric Make will be able to detect a conflict and rerun the continuation job.



13. Electric Make sends the last command, `echo "finish libs"`.
14. Agent runs the command, `echo "finish libs"`, and captures the result.
15. Agent returns the result, "`finish libs`", to Electric Make.

The Electric Make Stub solution addresses three basic parallel submake problems by:

- **Running parallel submakes in stub mode** – Stubs finish instantaneously and discover jobs as quickly as possible so the build is as concurrent as possible.
- **Creating a single logical Make instance** – New Make instances are started by the top-level Electric Make process only and their targets are integrated into the global dependency tree. Electric Make can then track dependencies across Make instances and use its conflict resolution technology to re-order jobs that may have run too soon.

- **Capturing submake invocations** – By capturing submake invocations as they occur, the submake stub works with your makefiles and builds scripts for the majority of cases “as-is.” However, the stub introduces a behavior change that may require some makefile changes. See Submake Stub Compatibility.

Submake Stub Compatibility

ElectricAccelerator submake stubs allow your existing recursive builds to behave like a single logical Make instance to ensure fast, correct parallel builds. Stubs do introduce new behavior, however, that may appear as build failures. This section describes what constructs are not supported and what must be changed to make ElectricAccelerator stubs compatible with submake stubs.

At this point, it is useful to revisit the relationship between commands and rules:

```
all:
        echo "this is a command"
        echo "another command that includes a copy" ; \
        cp aa bb
        echo "so does this command" ; \
        cp bb cc
        cp cc dd
```

The rule [above] contains four commands: (1) echo, (2) echo-and-copy, (3) another echo-and-copy, and (4) a copy. Note how semicolons, backslashes, and new lines delimit (or continue) commands. The rule becomes a job when Make schedules it because it is needed to build the “all” target.

The most important features of ElectricAccelerator submake stubs are:

- A submake stub never has any output and always exits successfully.
- The Agent sends stub output back (if any) after each command.
- Commands that follow a stub are invoked after the submake in the serial order.

Because a submake stub is really just a way of marking the Make invocation and does not actually do anything, *you cannot rely on its output (stdout/stderr, exit status, or file system changes) in the same command*.

In the following three examples, incompatible Accelerator commands are described and examples for fixing the incompatibility are provided.

Example 1: A command that reads submake file system changes

```
makelibs:
        $(MAKE) -C libs libcore.aa ; cp libs/libcore.aa /lib
```

In this example, a single command spawns a submake that builds `libcore.a` and then copies the new file into the `/lib` directory. If you run this rule as-is with Accelerator, the following error may appear:

```
cp: cannot stat 'libs/libcore.aa': No such file or directory
make: *** [all] Error 1
```

The submake stub exited immediately and the `cp` begins execution right after it in the same command. Electric Make was not notified of the new Make instance yet, so no jobs to build `libcore.a` even exist. The `cp` fails because the expected file is not present.

An easy fix for this example is to remove the semicolon and make the `cp` a separate command:

```
makelibs:
        $(MAKE) -C libs libcore.aa
        cp libs/libcore.aa /lib
```

Now the `cp` is in a command after the submake stub sends its results back to the build machine. Electric Make forces the `cp` to wait until the submake jobs have completed, thus allowing the copy to succeed because the file is present. Note that this change has no effect on other Make variants so it will not prevent building with existing tools.

Note: In general, Accelerator build failures that manifest themselves as apparent re-ordering or missing executions are usually because of commands reading the output of submake stubs. In most cases, the fix is simply to split the rule into multiple commands so the submake results are not read until after the submake completes.

Example 2: A command that reads submake *stdout*

```
makelibs:
        $(MAKE) -C libs mkinstall > installer.sh
```

The output is captured in a script that could be replayed later. Running this makefile with Accelerator always produces an empty `installer.sh` because submake stubs do not write output. When Electric Make does invoke this Make instance, the output goes to standard output, as though no redirection was specified.

Commands that read from a Make *stdout* are relatively unusual. Those that do often read from a Make that does very little actual execution either because it is invoked with `-n` or because it runs a target that writes to *stdout* only. In these cases, it is not necessary to use a submake stub. The Make instance being spawned is small and fast, and running it directly on the Agent in its entirety does not significantly impact performance.

You can specify that an individual `emake` invocation on the Agent does not enter stub mode, but instead behaves like a local (non-cluster) Make simply by setting the `EMAKE_BUILD_MODE` environment variable for that instance:

```
makelibs:
       EMAKE_BUILD_MODE=local \
$(MAKE) -C libs mkinstall >
       installer.sh
```

For Windows:

```
makelibs:
       set EMAKE_BUILD_MODE=local && $(MAKE) -C libs mkinstall >
               installer
```

Electric Make automatically uses local mode when the `-n` switch is specified.

Example 3: A command that reads submake exit status

```
makelibs:
        $(MAKE) -C libs || echo "failure building libs"
```

This example is a common idiom for reporting errors. The `||` tells the shell to evaluate the second half of the expression *only* if Make exits with non-zero status. Again, because a submake stub always exits with **0**, this clause will never be invoked with Accelerator, even if it would be invoked with GNU Make. If you need this type of fail-over handling, consider post-processing the output log in the event of a build failure. Also see Annotation for more information.

Another common idiom in makefiles where exit status is read in loop constructs such as:

```
all:
       for i in dir1 dir2 dir3 ; do \
               $(MAKE) -C $$i || exit 1;\
       done
```

This is a single command: a “for” loop that spawns three submakes. The `|| exit 1` is present to prevent GNU Make from continuing to start the next submake if the current one fails. Without the `exit 1` clause, the command exit status is the exit status from the last submake, regardless of whether the preceding submakes

succeeded or failed, or regardless of which error handling setting (for example, `-i`, `-k`) was used in the Make. The `|| exit 1` idiom is used to force the submakes to better approximate the behavior of other Make targets, which stops the build on failure.

On first inspection, this looks like an unsupported construct for submake stubs because exit status is read from a stub. Accelerator never evaluates the `|| exit 1` because the stub always exits with status code **0**. However, because the submakes really are reintegrated as targets in the top-level Make, a failure in one of them halts the build as intended. Explained another way, Accelerator already treats a submakes loop as a series of independent targets, and the presence or absence of the GNU Make `|| exit 1` hint does not change this behavior. These constructs should be left as-is.

Hidden Targets

Electric Make differs from other Make variants in the way it searches for files needed by pattern rules (also called suffix or implicit rules) in a build.

- At the beginning of each Make instance, Electric Make searches for matching files for all pattern rules *before it runs any commands.* After eMake has rules for every target that needs updating, it schedules the rules [creating jobs] and then runs those jobs in parallel for maximum concurrency.
- Microsoft NMAKE and GNU Make match pattern rules *as they run commands*, interleaving execution and pattern search.

Because of the difference in the way Electric Make and NMAKE match pattern rules, NMAKE and Electric Make can produce different makefile output with *hidden targets*. A hidden target (also known as a “hidden dependency”) is a file that is:

- created as an undeclared side-effect of updating another target
- required by a pattern to build a rule

Consider the following makefile example:

```
all: bar.lib foo.obj

bar.lib:
        touch bar.lib foo.c

.c.obj:
        touch $@
```

Notice that `foo.c` is created as a side-effect of updating the `bar.lib` target. Until `bar.lib` is updated, no rule is available to update `foo.obj` because nothing matches the `.c.obj` suffix rule.

NMAKE accepts this construct because it checks for `foo.c` existence before it attempts to update `foo.obj`. NMAKE produces the following result for this makefile:

```
touch bar.lib foo.c
touch foo.obj
```

Electric Make, however, performs the search for files that match the suffix rule once so it can schedule all jobs immediately and maximize concurrency. Electric Make will not *notice* the existence of `foo.c` by the time it attempts to update `foo.obj`, even if `foo.c` was created. Electric Make fails with:

```
NMAKE : fatal error U1073: don't know how to make 'foo.obj'
Stop.
```

The fix is simply to identify `foo.c` as a product for updating the `bar.lib` target, so it is no longer a hidden target. For the example above, adding a line such as `foo.c: bar.lib` is sufficient for Electric Make to understand that `.c.obj` suffix rule matches the `foo.obj` target if `bar.lib` is built first. Adding this line is more accurate and has no effect on NMAKE.

GNU Make is similarly incompatible with Electric Make, but the incompatibility is sometimes masked by the GNU Make directory cache. GNU Make attempts to cache the directory contents on first access to improve performance. Unfortunately, because the time of first directory access can vary widely depending on which targets reference the directory and when they execute, GNU Make can appear to fail or succeed randomly in the presence of hidden targets.

For example, in this makefile, the file `$(DIR)/foo.yy` is a hidden target created as a side-effect of updating `aa` and needed by the pattern rule for `foo.xx`:

```
all: aa bb
aa:
	touch $(DIR)/foo.yy
bb: foo.xx

%.xx: $(DIR)/%.yy

	@echo $@
```

Depending on the value of `DIR`, this build may or may not work with GNU Make:

```
% mkdir sub; gmake DIR=sub
touch sub/foo.yy
foo.xx

% gmake DIR=.
touch ./foo.yy
gmake: *** No rule to make target 'foo.xx', needed by 'bb'.  Stop.
```

Electric Make does not attempt to emulate this behavior. Instead, it consistently refuses to schedule `foo.xx` because it depends on a hidden target (just as it did in the NMAKE emulation mode in the earlier example). In this case, adding a single line declaring the target: `$(DIR)/foo.yy: aa` is sufficient to ensure it always matches the `%.xx` pattern rule.

Note: If a build completes successfully with Microsoft NMAKE or GNU Make, but fails with "don't know how to make `<x>`" with Electric Make, look for rules that create `<x>` as a side-effect of updating another target. If `<x>` is required by a suffix rule also, it is a hidden target and needs to be declared as explicit output to be compatible with Electric Make.

There are many other reasons why hidden targets are problematic for all Make-based systems and why eliminating them is good practice in general. For more information, see:

- "Paul's Rules of Makefiles" by Paul Smith at http://www.make.paulandlesley.org/rules.html. Among other useful guidelines for writing makefiles, the primary author of GNU Make writes, "Every non-`.PHONY` rule **must** update a file with the *exact* name of its target. [...] That way you and GNU Make always agree."
- "The Trouble with Hidden Targets" by John Graham-Cumming at http://www.cmcrossroads.com/content/view/6519/120/.

Note: In a limited number of cases, eMake may conclude that a matching pattern rule for an output target does not exist. This occurs because eMake's strict string equality matching for prerequisites determines that the prerequisites are different (even though the paths refer to the same file) and that there is no rule to build it.

Wildcard Sort Order

A number of differences exist between GNU Make and Electric Make regarding the use of `$(wildcard)` and prerequisite wildcard sort order functions. When using the `$(wildcard)` function or using a wildcard in the rule prerequisite list, the resultant wildcard sort order may be different for GNU Make and Electric Make.

Different GNU Make versions are not consistent and exhibit permuted file lists. Even a GNU Make version using different system libraries versions will exhibit inconsistencies in the wildcard sort order.

No difference exists in the file list returned, other than the order. If the sort order is important, you may wrap `$(wildcard)` with `$(sort)`.

For example:

```
$(sort $(wildcard *.foo))
```

Do not rely on the order of rule prerequisites generated with a wildcard. For example, using `target: *.foo`.

Relying on the order of `*.foo` can be dangerous for both GNU Make and Electric Make. Neither GNU Make nor Electric Make guarantees the order in which those prerequisites are executed.

Delayed Existence Checks

All Make variants process makefiles by looking for rules to build targets in the dependency tree. If no target rule is present, Make looks for a file on disk with the same name as the target. If this *existence check* fails, Make notes it has no rule to build the target.

Electric Make also exhibits this behavior, but for performance reasons it delays the existence check for a target without a makefile rule until just before that target is needed. The effect of this optimization is that Electric Make may run more commands to update targets [than GNU Make or NMAKE] *before* it discovers it has no rule to make a target.

For example, consider the following makefile:

```
all: nonexistent aa bb
aa:
        @echo $@
bb:
        @echo $@
```

GNU Make begins by looking for a rule for `nonexistent` and, when it does not find the rule, it does a file existence check. When that fails, GNU Make terminates immediately with:

```
make: *** No rule to make target 'nonexistent', needed by 'all'.  Stop.
```

Similarly, NMAKE fails with:

```
NMAKE : fatal error U1073: don't know how to make 'nonexistent' Stop.
```

Electric Make delays the existence check for `nonexistent` until it is ready to run the `all` target. First, Electric Make finishes running commands to update the `aa` and `bb` prerequisites. Electric Make fails in the same way, but executes more targets first:

```
aa
bb
make: *** No rule to make target 'nonexistent', needed by 'all'.  Stop.
```

Of course, when the existence check succeeds [as it does in any successful build], there is no behavioral difference between Electric Make and GNU Make or Microsoft NMAKE.

Multiple Remakes (GNU Make only)

GNU Make has an advanced feature called Makefile Remaking, which is documented in the GNU Manual, "How Makefiles are Remade," and available at:
http://www.gnu.org/software/make/manual/make.html#Remaking-Makefiles

To quote from the GNU Make description:

> "Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want make to get an up-to-date version of the makefile to read in.

> "To this end, after reading in all makefiles, make will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see section Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, make starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)"

This feature can be very useful for writing makefiles that automatically generate and read dependency information with each build. However, this feature can cause GNU Make to loop infinitely if the rule to generate a makefile is always out-of-date:

```
all:
	@echo $@
makefile: force
	@echo "# last updated: 'date'" >> $@
force:
```

In practice, a well-written makefile will not have out-of-date rules that cause it to regenerate. The same problem, however, can occur when Make detects a clock skew—most commonly due to clock drift between the system running Make and the file server hosting the current directory. In this case, Make continues to loop until the rule to rebuild the makefile is no longer out-of-date.

In the example below, `DIR1` and `DIR2` are both part of the source tree:

```
-include $(DIR1)/foo.dd
all:
	@echo $@
$(DIR1)/foo.dd: $(DIR2)/bar.dd
%.d:
	touch $@
```

If two directories are served by different file servers and the clock on the system hosting `DIR2` is slightly faster than `DIR1`, then even though `foo.dd` is updated after `bar.dd`, it may appear to be older. On remaking, GNU Make will again see `foo.dd` as out-of-date and restart, continuing until the drift is unnoticeable.

Electric Make fully supports makefile remaking and can be configured to behave exactly as GNU Make. However, by default, to ensure builds do not loop unnecessarily while remaking, Electric Make limits the number of times it restarts a make instance to 10. If your build is looping unnecessarily, you may want to lower this value or disable remaking entirely by setting:

```
--emake-remake-limit=0
```

NMAKE Inline File Locations (Windows only)

NMAKE contains a feature to create *inline files* with temporary file names. For example, the following makefile creates a temporary inline file containing the word "pass" and then uses "type" to output it.

```
all:
		type <<
	pass
	<<
```

With Microsoft NMAKE, the file is created in the directory where `%TMP%` points. Electric Make does not respect the `%TMP%` setting and creates the inline file in the rule working directory that needs the file.

How eMake Processes MAKEFLAGS

eMake uses the following process:

1. Similar to GNU Make, eMake condenses no-value options into one block.
2. When eMake encounters an option with a value, it does what GNU Make does, it appends the value and starts the next option with its own -
3. Certain options are ignored/not created. This changes the layout of the options in `MAKEFLAGS` (for example `-j`, `-l`).
4. eMake-specific options are not added to `MAKEFLAGS`, but are handled through `EMAKEFLAGS`.
5. Passing down environment variables as `TEST=test` renders the same result as in GNU Make (an extra - at the end, followed by the `variable=value`).
6. On Windows, eMake prepends the `--unix` or `--win32` flag explicitly.

Chapter 12: Performance Optimization

The following topics discuss how to use ElectricAccelerator Developer Edition's performance optimization features and performance tuning techniques.

Topics:

Optimizing Android Build Performance

Recommended Deployment/Sizing

For optimal Android build performance, make sure you have at least 48 ElectricAccelerator agents available. Electric Cloud internal tests showed that utilizing local cores and agents using ElectricAccelerator Developer Edition combined with remote agents from a back-end cluster is a cost-efficient approach to accumulate a large number of cores, while still making the infrastructure centrally available for a large community of developers.

Recommended Steps

Electric Cloud recommends the following actions:

1. Upgrade your ElectricAccelerator deployment to v7.0 or later (procedure available in the *ElectricAccelerator Installation and Configuration Guide*).
2. Configure a clean build environment.
3. To generate a correct history-file and record all the actual dependencies in the build, ElectricAccelerator needs to run a learning build.

   Sample command:

   ```
   emake \
     --emake-cm=$YOUR_CLUSTER_MANAGER \
     --emake-root=$YOUR_ROOTS \
     --emake-annofile=$YOUR_ANNOTATION_FILE \
     --emake-annodetail=basic \
     --emake-logfile=$YOUR_LOG_FILE_NAME \
     --emake-debug=g \
     --emake-optimize-deps=1 \
     --emake-parse-avoidance=1 \
     --emake-autodepend=1 \
     --emake-suppress-include=*.d \
     --emake-suppress-include=*.P \
     -f Makefile \
     -f noautodep.mk
   ```

   `noautodep.mk` contains "hints" used to tailor ElectricAccelerator dependency management for better performance with Android builds. The contents are as follows:

   ```
   #pragma noautodep */.git/*
   $(local-intermediates-dir)/libbcc-stamp.c :

   #pragma noautodep */out/target/product/generic/system/bin/cat
   $(linked_module) :

   #pragma nolookup noproguard.classes-with-local.dex
   #pragma nolookup noproguard.classes.dex
   #pragma nolookup javalib.jar
   #pragma nolookup classes-full-debug.jar
   out/target/common/docs/doc-comment-check-timestamp :
   ```

4. Make "clean" and then re-run the build. ElectricAccelerator will use the previous learning build to optimize performance.

   Run the same sample command from step 3.

Note: If you want to find out why a cached result was not used, add "P" debugging to `--emake-debug`.

Dependency Optimization

ElectricAccelerator includes a dependency optimization feature to improve performance. By learning which dependencies are actually needed for a build, Accelerator can use that information to improve performance in subsequent builds.

When dependency optimization is enabled, emake maintains a dependency information file for each makefile. If a build's dependencies have not changed from its previous build, emake can use that stored dependency information file for subsequent builds.

Enabling Dependency Optimization

You must first run a "learning" build with the dependency optimization feature enabled. To enable dependency optimization, set the following:
`--emake-optimize-deps=1`

For the learning build, (because there is no stored dependency information file for that specific makefile), the argument only saves a new result. For subsequent builds, the argument enables the reuse of stored dependency information and saves a new file as appropriate. If you do not specify `--emake-optimize-deps=1`, then dependency information is not saved or accessed.

Note: It is recommended that you turn on `--emake-autodepend=1` to ensure accurate incremental builds.

The following table describes dependency optimization-related options.

eMake Option	Description
`--emake-assetdir=<`*`path`*`>`	Use the specified directory for assets such as saved dependency information. The default directory is `.emake`. This option also affects parse avoidance.
`--emake-optimize-deps=` `<`*`0/1`*`>`	Use the saved dependency information file for a makefile when dependencies are the same and save new dependency information when appropriate.

Dependency Optimization File Location and Naming

Dependency information files are saved in the working directory where emake was invoked under `<`*`assetdir`*`>/deps`. (The default asset directory is `.emake`.)

The file is named from the md5 hash of the root-relative path of the "main" makefile for the make instance that the dependency information file belongs to.

Examples

If your emake root is /tmp/src, and your makefile is /tmp/src/Makefile, then the dependency information file is named after the md5 hash of the string "Makefile".

If your root is /tmp/src and your makefile is /tmp/src/foo/Makefile, the file is named after the md5 hash of the string "foo/Makefile".

Parse Avoidance

ElectricAccelerator includes a parse avoidance feature that can almost eliminate makefile parse time. By caching and reusing parse result files, Accelerator can speed up both full builds and incremental builds.

Parse avoidance works when emulating GNU Make with clusters and/or local agents (ElectricAccelerator Developer Edition).

Parse avoidance utilizes the concept of cache "slots." When parse avoidance is enabled, emake maintains a slot for each combination of non-ignored command line options, non-ignored environment variable assignments, and current working directory. (Ignored arguments and environment variables are listed here) A slot can be empty or hold a previously cached result. If the appropriate slot holds an up-to-date result, parsing is avoided.

A cached result becomes obsolete if emake detects filesystem changes that might have caused a different result (with the same command line options, environment variable assignments, and current working directory). Such filesystem changes include any file read by the parse job, which means all makefiles, all programs invoked by $(shell) during the parse (as opposed to during rule execution), the files they read, and so on.

Console Output

When a cached parse result is reused, eMake replays the filesystem modifications made during the parse job and any standard output or standard error that the original parse job produced. For example, if the makefile includes

```
VARIABLE1 := $(shell echo Text > myfile)

VARIABLE2 := $(warning Warning: updated myfile)
```

then when using a cached parse result emake will create "myfile" and will print "Warning: updated myfile".

Enabling Parse Avoidance

You must first run a "learning" build with the parse avoidance feature enabled. To enable parse avoidance, set the following: `--emake-parse-avoidance=1`

For the learning build, (because the parse cache is empty), the argument only saves a new result to the cache. For subsequent builds, the argument enables the reuse of cached parse results and saves a new result to the cache as appropriate. If you do not specify `--emake-parse-avoidance=1`, then the parse avoidance cache is not accessed at all.

Note: The following actions are also recommended:

- Turn on `--emake-autodepend=1`
- Disable generated dependencies (either by modifying your makefiles or by using `--emake-suppress-include`; see below)

The following table describes parse avoidance-related options.

eMake Option	Description
`--emake-assetdir=<`*`path`*`>`	Use the specified directory for assets such as cached parse results. The default directory is `.emake`. This option also affects dependency optimization.
`--emake-parse-avoidance=<`*`0/1`*`>`	Avoid parsing makefiles when prior parse results remain up-to-date and cache new parse results when appropriate.
`--emake-parse-avoidance-ignore-env=<`*`var`*`>`	Ignore the named environment variable when searching for applicable cached parse results. To ignore more than one variable, use this option multiple times.

eMake Option	Description
`--emake-parse-avoidance-ignore-path=<path>`	Ignore this file or directory when checking whether cached parse results are up-to-date. **Append** % for prefix matching (the % must be the last character). To ignore more than one path or prefix, use this option multiple times. Incorrect placement of `%` will result in an error. Correct: `--emake-parse-avoidance-ignore-path=.foo%` Incorrect: `--emake-parse-avoidance-ignore-path=.%foo`
`--emake-suppress-include=<pattern>`	Skip matching makefile includes (such as generated dependencies). Generally, you should not suppress makefile includes unless they are generated dependency files, and you have enabled automatic dependencies as an alternative way of handling dependencies. **Note:** If the pattern does not have a directory separator, then the pattern is compared to the include's filename component only. If the pattern has a directory separator, then the pattern is taken relative to the same working directory that applies to the include directive and compared to the included file's entire path.

If a file is read during a parse but changes before eMake attempts to reuse that parse's results, the cached parse result is normally considered to be obsolete. You can, however, temporarily override this decision with `--emake-parse-avoidance-ignore-path`.

eMake permanently ignores the effect on a given parse result of certain special files that may have existed when it was created, in all cases using the names they would have had at that time:

- Anything in ".emake" or whatever alternative directory you specified using `--emake-assetdir`
- Client-side eMake debug log (as specified by `--emake-logfile`)
- Annotation file (as specified by `--emake-annofile`)
- History file (as specified by `--emake-historyfile`)

Notes:

- It is recommend that your makefiles avoid any $(shell) expansion that uses these four special files and directories in a meaningful way, because parse avoidance will ignore changes to them.
- Enabling remote parse debugging (the `--emake-rdebug` option) disables parse avoidance.

Parse Avoidance Example

`noautodep.mk` is used for this example. The contents are:

```
#pragma noautodep */.git/*
$(local-intermediates-dir)/libbcc-stamp.c :

#pragma noautodep */out/target/product/generic/system/bin/cat
$(linked_module) :
```

The following parse avoidance example provides command line arguments for Android 4.1.1.

Note: You may need additional options such as `--emake-cm`

```
--emake-parse-avoidance=1 --emake-autodepend=1 --emake-suppress-include=*.d --
emake-suppress-include=*.P --emake-debug=P -f Makefile -f noautodep.mk
```

Deleting the Cache

To delete the cache, delete `<assetdir>/cache.*`. (The default asset directory is `.emake`.)

For example: `rm -r .emake/cache.*`

Limitations

- Windows is currently not supported.
- The parse avoidance feature does not detect the need to reparse in these cases:
  - Changes to input files and programs that are used during the parse (such as makefile includes) but that Accelerator does not virtualize (because they are not located under `--emake-root=...`).
  - Changes to non-filesystem, non-registry aspects of the environment, for example, the current time of day
- If a parse job checks the existence or timestamp of a file without reading it, parse avoidance may not invalidate its cached result when the file is created, destroyed, or modified.
  - Using `--emake-autodepend=1` and `--emake-suppress-include=<pattern>` in conjunction with parse avoidance helps to avoid this limitation. If a generated dependency file did not exist when a parse result was saved into the cache, emake may reuse that cached parse result even after that dependency file was created. Of course, you also benefit from eDepend's performance and reliability gains.

Debugging

To log information about parse avoidance, use "P" debug logging (set in `--emake-debug=<value>`).

Debugging Log Example

You can look in "P" debug logging for an explanation of why a cached result was found to be obsolete:

```
WK01: 0.062173 Input changed: job:J08345218 slot:b6189634a793fee2fe1929fbf47cc4e4
path:/home/aeolus/t/Makefile thenSize:175 nowSize:176 ignore:0
WK01: 0.062228 Input changed: job:J08345218 slot:b6189634a793fee2fe1929fbf47cc4e4
path:/home/aeolus/t/fog.mk thenSize:0 nowSize:1 ignore:0
...
WK01: 0.062284 Cache slot is obsolete: job:J08345218
slot:b6189634a793fee2fe1929fbf47cc4e4
```

Notice that "`Makefile`" and "`fog.mk`" were both bigger; either of those changes would have triggered a reparse. The "ignore:0" comments indicate that `--emake-parse-avoidance-ignore-path` was not used to ignore the changes.

Using Key Files for Debugging

To discover why a new cache slot was used, look in "P" debug logging for the name of the "key" file for that new cache slot; for example:

```
WK01: 11.853035 Saved slot definition: .emake/cache.11/i686_
Linux/d9/0f/79/4fb975e8ff94dfa2569a303e62.new.2NPR8J/key
```

Note: The ".new.2NPR8J" portion of the slot directory name should have already been removed automatically by eMake before you need to access the key file.

Then diff that key file against other key files in sibling directories to see what parameters changed. To determine which slot directories to compare, grep for the parse job IDs in "P" debug logging. You can get parse job IDs from annotation using ElectricInsight. Each key file contains an artificial "cd" command to express the working directory, all non-ignored environment variables, and the non-ignored command-line arguments. Ignored command-line arguments and environment variables are omitted from key files. The key file is stored in a directory whose pathname is derived from the md5sum of the file itself. Also, the command line may have been normalized to one that is equivalent to the original one. Currently, emake quotes special characters according to UNIX/Linux/Cygwin "sh" shell rules.

Ignored Arguments and Environment Variables

The following arguments do not affect which cache slot is chosen:

```
--emake-annodetail                --emake-ledger
--emake-annofile                  --emake-ledgerfile
--emake-annoupload                --emake-localagents
--emake-assetdir                  --emake-logfile
--emake-big-file-size             --emake-logfile-mode
--emake-build-label               --emake-maxagents
--emake-class                     --emake-maxlocalagents
--emake-cluster-timeout           --emake-mem-limit
--emake-cm                        --emake-parse-avoidance-ignore-path
--emake-debug                     --emake-pedantic
--emake-hide-warning              --emake-priority
--emake-history                   --emake-readdir-conflicts
--emake-history-force             --emake-resource
--emake-historyfile               --emake-showinfo
--emake-idle-time                 --emake-tmpdir
--emake-job-limit                 --emake-yield-localagents
```

The following default set of environment variables do not affect which cache slot is chosen. You can specify additional environment variables with `--emake-parse-avoidance-ignore-env`. You must use `--emake-parse-avoidance-ignore-env` once for each variable to ignore.

```
EMAKE_PARSEFILE
EMAKE_RLOGFILE
ECLOUD_AGENT_NUM
ECLOUD_BUILD_CLASS
ECLOUD_BUILD_COUNTER
ECLOUD_BUILD_ID
ECLOUD_BUILD_TAG
ECLOUD_BUILD_TYPE
EMAKE_APP_VERSION
EMAKE_BUILD_MODE
ECLOUD_RECURSIVE_COMMAND_FILE
EMAKE_LEDGER_CONTEXT
EMAKE_MAKE_ID
TMP
TEMP
TMPDIR
_
OLDPWD
SHLVL
SSH_AGENT_PID
SSH_AUTH_SOCK
SSH_CLIENT
SSH_CONNECTION
SSH_TTY
GPG_AGENT_INFO
COLUMNS
LINES
TERM
TERMCAP
COLORFGBG
LS_COLORS
DISPLAY
DESKTOP_SESSION
SHELL_SESSION_ID
SESSION_MANAGER
DBUS_SESSION_BUS_ADDRESS
KDE_FULL_SESSION
KDE_MULTIHEAD
KDE_SESSION_UID
KDE_SESSION_VERSION
KONSOLE_DBUS_SERVICE
KONSOLE_DBUS_SESSION
WINDOWID
WINDOWPATH
XCURSOR_THEME
XDG_SESSION_COOKIE
XDM_MANAGED
```

Javadoc Caching

The ElectricAccelerator Javadoc caching feature can improve build performance by caching and reusing Javadoc files.

Enabling Javadoc Caching

To enable Javadoc caching for a specific rule, add the following before that rule:

```
#pragma cache javadoc
```

Alternatively, you can mention the target again in a separate makefile:

```
#pragma cache javadoc
mytarget :
```

By default, the cache becomes obsolete when `*.java` files are added to or removed from directories that are read by the rule body. You can override this default wildcard pattern by invoking:

```
#pragma cache javadoc -readdir pattern
```

pattern is the pattern that you want to ignore.

eMake permanently ignores the effect on a given result of certain special files that may have existed when it was created, in all cases using the names they would have had at that time:

- Anything in ".emake" or whatever alternative directory you specified using `--emake-assetdir`
- Client-side eMake debug log (as specified by `--emake-logfile`)
- Annotation file (as specified by `--emake-annofile`)
- History file (as specified by `--emake-historyfile`)

Note: -hdf arguments are removed from consideration for which cache slot to use.

If you want to delete the cache, see Deleting the Cache.

For debugging information, see Debugging.

Limitations

- The following Javadoc invocations are currently not supported:
  - Javadoc is invoked indirectly, such as through a wrapper script
  - The rule body recursively invokes eMake
  - If Javadoc is named something other than "javadoc"
- Javadoc caching is sensitive to changes in `*.java` wildcard outcomes in directories that are actually read by the job, and any new subdirectories thereof.
- Currently, `#pragma cache ...` cannot be applied to patterns as such, only to explicitly specified targets (though the commands may be provided by a pattern).
- Windows is currently not supported.
- The Javadoc caching feature does not detect the need to rebuild Javadoc files in these cases:
  - Changes to input files and programs that are used by a rule body but that Accelerator does not virtualize (because they are not located under `--emake-root=...`).
  - Changes to non-filesystem, non-registry aspects of the environment, for example, the current time of day
- If a rule job checks the existence or timestamp of a file without reading it, Javadoc caching may not invalidate its cached result when the file is created, destroyed, or modified. However, Javadoc caching will detect when files matching "*.java" (or the pattern specified by "-readdir") are added or removed from directories read by the rule body, or any new subdirectories thereof. In that case eMake will rerun Javadoc.

Schedule Optimization

How it Works

Schedule optimization uses performance and dependency information from previous builds in order to optimize the runtime ordering of jobs in subsequent builds.

This approach can improve performance on the same number of agents (compared to v7.0, or compared to v7.1 when the requisite information is not available), and enables eMake to realize the 'best possible' build performance for a given build with fewer agents.

Using Schedule Optimization

Schedule optimization is enabled by default, but it has no effect if scheduling data files are not available.

Emake automatically generates the scheduling data file in the asset directory (`.emake` by default) in the `sched` subdirectory under a platform-specific directory such as `Linux`, `SunOS`, or `win32`. For example, on Linux the scheduling data file is found in `.emake/sched/Linux/emake.sched`.

The next eMake build that uses the scheduling data file will have improved performance.

Disabling Schedule Optimization

To disable schedule optimization, add the following to the command-line:

```
--emake-optimize-schedule=0
```

Running a Local Job on the Make Machine

Normally, ElectricAccelerator executes all build commands on the Agents. Any file access by commands is served by in-memory EFS cache or fetched from Electric Make and monitored to ensure dependency information is captured, thus ensuring correct results.

Sometimes it is not desirable to run a command on an Agent. In these instances, ElectricAccelerator Developer Edition also provides the `#pragma runlocal` directive. In a makefile, the `#pragma runlocal` directive allows you to specify a job to run locally on the host build machine.

A run-local job executes on the host build machine where no Electric File System exists to virtualize and monitor file access. Therefore, special care must be taken for build success. The following are two important properties of run-local jobs:

- **Forced serial mode** – Electric Make executes a run-local job when all previous commands are complete, and does not allow other commands to run until the run-local job is complete. In other words, while the run-local job is running, no other build steps are executed.
- **Current directory changes only** – The run-local job must not make any changes outside of its current working directory. If changes occur, subsequent jobs will not see the output outside of the current working directory and could fail—precise interaction is dependent on the Electric Make file system state at the time the job is invoked.

  IMPORTANT: After a run-local job, eMake reloads the filesystem state for the current working directory of the Make, not the job.

  Note: Marking a job that does not meet this requirement with `#pragma runlocal` is not a supported configuration.

Because of these restrictions, run-local jobs are intended only for jobs that:

- Perform heavy I/O [consequently, very inefficient to run remotely]
- Execute near the end of the build (for example, forcing the build into serial mode would have minimal performance impact)
- Logically produce a final output not consumed by steps later in the build

Typically, final link or packaging steps are the only commands that should be marked *runlocal*.

If a run-local job modifies the registry, these modifications will not be visible to later jobs in the build. Ensure you do not mark a job as *runlocal* if that job modifies the registry in a way that later parts of the build depend on.

To mark a job runlocal, precede its rule in the makefile with `#pragma runlocal`. For example:

```
#pragma runlocal
myexecutable:
	$(CC) -o $@ $(OBJS) ...
```

Also, you can specify `#pragma runlocal` for a pattern rule. In this case, all jobs instantiated from that rule are marked run-local:

```
#pragma runlocal
%.exe: %.obj
	$(CC) -o $@ $*
```

You can cause all jobs after a certain point in the build to run locally by using `#pragma runlocal sticky`. This variant means all jobs that occur later in the serial order than the job specifically marked as “runlocal sticky”

are run locally, as if “runlocal” were specified for all of them. This is NOT the same as “all jobs declared after this job in the makefile”! The order of declaration of jobs in a makefile has no relationship to the serial order of those jobs, so the difference is significant.

Serializing All Make Instance Jobs

Normally, Electric Make runs all jobs in a Make instance in parallel on multiple distinct Agents. For most builds, this process ensures the best possible performance.

In some cases, however, all jobs in a Make instance are interdependent and must be run serially, for example, a set of jobs updating a shared file. In particular, Microsoft Visual C/C++ compilers exhibit this behavior when they create a common program database (PDB) file to store symbol and debug information to all object files.

Example

The following makefile invokes `cl` with the `/Zi` flag to specify the program database file be created with type and symbolic debugging information:

```
all:
        $(MAKE) my.exe
        $(MAKE) other.exe

my.exe: a.obj b.obj c.obj my.obj
                cl /nologo /Zi $^ /Fe'my.exe'

%.obj: %.c

                cl /nologo /Zi /c /Fo$@ $^
```

In this build, the `a.obj, b.obj, c.obj` and `my.obj` jobs are implicitly serialized because they all write to the shared PDB file (by default, `vc70.pdb`). In this case, jobs run in parallel, and running them on separate Agents only introduces unnecessary network overhead. This job type needs to run serially so it can correctly update the PDB file.

The Electric Make special directive, `#pragma allserial` used in the makefile, allows you to disable a parallel build in a Make instance and run the job serially on a single Agent. By inserting the `#pragma allserial` directive at the beginning of a line anywhere in the makefile, the directive specifies that all jobs in that make instance be serialized. This process maximizes network and file cache efficiency.

In the example above, by prefixing the `%.obj` pattern rule with the `#pragma allserial` directive:

```
#pragma allserial
%.obj: %.c
        cl /nologo /Zi /c /Fo$@ $^
```

Electric Make runs compiles and links for the `my.exe` Make instance in serial on the same Agent.

Splitting PDBs Using hashstr.exe

The `hashstr.exe` utility creates a hash of the filename given a modulus (maximum number of PDBs that will be produced). A given file must always produce the same PDB or history would constantly change. The hash should only include the filename and not its full path. Precompiled headers (PCHs) must be turned off.

Example

Usage: `hashstr "mystring" [modulus]`

Where mystring is the string from which to generate the hash value, and modulus is the number of hash bins you want to use.

You can add this to a pattern rule for builds that suffer from performance degradation due to PDB serialization,

with something similar to the following:

```
%.o: %.c
$(CC) /c $(cflags) $(PCH_USE_FLAGS) $(cvars) $(cplus_flags) $(LOCAL_INCLUDE)
$(PCB_INCLUDE) $< /Fo$@ /Fd$(shell ${path-to-hashstr}/hashstr.exe "$@"
${hashstr-modulus}).pdb
```

Managing Temporary Files

During the build process, temporary files are created, then deleted automatically when a build completes. To maintain optimum efficiency, it is important to identify where these files are created in relation to the final output files.

Configuring the Electric Make Temporary Directory

Electric Make runs commands and collects output in parallel, but the results are written in order, serially. This feature ensures a build behaves exactly as if it were run locally. See Transactional Command Output.

The Electric Make *temporary directory* is used to store output files from commands (for example, object files) that finished running but are waiting their turn to be relocated to a final destination on disk. By default, Electric Make creates a temporary directory in the current working directory. The directory name will have the form:

`ecloud_tmp_<pid>_<n>`
where `pid` is the eMake process identifier.
where `n` is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

Electric Make removes the temporary directory on exit (including **Ctrl-C** user interrupt). If Electric Make is terminated unexpectedly (for example, by the operating system), the temporary directory may persist and need to be removed manually.

Important temporary directory requirements include:

- Must be writable by the eMake process
- Must not be on an NFS share
- Must have enough space to contain the build output
  - Ideally, the temporary directory should have enough space to hold the complete build output; in practice however, it may need only enough space for output from the largest commands.
  - The exact temporary directory space requirement varies greatly with the build size, the number of Agents used, and build system speed.

You can use the `--emake-tmpdir` command-line option or the `EMAKE_TMPDIR` environment variable to change the temporary directory default location:

```
% emake --emake-tmpdir=/var/tmp ...
```

Files are relocated from the temporary directory to their final location. Keeping the temporary directory on the same file system as the `EMAKE_ROOT` helps performance because files can simply be renamed in place, instead of copied.

Electric Make and agent machines reset the values for TEMP, TMP, and TMPDIR. This is necessary to avoid possible conflicts with multiple build agents/jobs running on the same machine.

If you have more than one `EMAKE_ROOT` that spans multiple file systems, you can specify more than one temporary directory to ensure Electric Make can always rename files in place, instead of copying them.

For example, if your `EMAKE_ROOT` contains three directories:

```
% setenv EMAKE_ROOT /home/alice:/local/output:/local/libs
```

that reside on two different file systems

```
% df /home/alice /local
Filesystem     1K-blocks  Used   Available Use% Mounted on
filer:/vol/    76376484 47234436 25262352   66%  /home
vol0/space
/dev/hdb1      76896316 51957460 21032656    2%  /local
```

to ensure better performance, specify two temporary directories—write permission at both locations is required:

```
% emake --emake-tmpdir=/home/alice:/local ...
```

When specifying multiple temporary directories, note the following:

- If a temporary directory was specified for a particular file system, Electric Make automatically uses that directory for any files destined to reside on that file system.
- If a temporary directory was not specified for a particular file system, Electric Make uses the *first* directory specified for files destined to reside on that file system.

Deleting Temporary Files

During a build, Electric Make creates a temporary directory inside the directory specified by the `EMAKE_TMPDIR` environment variable or in the directory specified by the `--emake-tmpdir` command-line option (or the current working directory if no Electric Make temporary directory is specified). If a specified directory does not exist, eMake creates it. All temporary directories created by eMake are automatically deleted when a build completes.

For example, if you set your temporary directory to /foo/bar/baz and only /foo/bar exists, eMake creates /foo/bar/baz and /foo/bar/baz/ecloud_tmp_<*pid*>_<*n*>, and then deletes both directories and all their contents when it exits. If /foo/bar/baz exists at the start of the build, only /foo/bar/baz/ecloud_tmp_<*pid*>_<*n*> is created and deleted.

However, if the build is aborted in-progress, eMake will not have the opportunity to remove the temporary directory.

Subsequent Electric Make invocations automatically delete temporary directories if they are more than 24 hours old. You can reclaim disk space more quickly by deleting temporary directories and their contents by hand. However, *do not delete the temporary directory while a build is in-progress*—this action causes the build to fail.

Temporary directory names are in this form:

`ecloud_tmp_<pid>_<n>`
where `pid` is the eMake process identifier.
where `n` is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

Chapter 13: Dependency Management

The following topics discuss ElectricAccelerator eDepend, the Ledger, and how ElectricAccelerator handles history data files.

Topics:

ElectricAccelerator eDepend

In its default configuration, Accelerator is designed to be a drop-in replacement for your existing Make tool—GNU Make or Microsoft NMAKE. Accelerator behaves exactly like your existing Make: it will rebuild (or declare up-to-date) the same targets following the same rules Make uses. The file dependency tracking technology in the Electric File System (EFS) and the Electric Make history feature is used to ensure the system reproduces exactly the same results in a parallel, distributed build as it would serially.

Because the system captures and records such detailed information about the relationships between build steps, it is uniquely capable of accomplishing much more than simply ensuring parallel builds are serially correct. In particular, by enabling ElectricAccelerator eDepend, you can wholly replace tools and techniques like **makedepend** or '**gcc -M**'—commonly used to generate makefiles too difficult to maintain by hand (for example, C-file header dependencies).

ElectricAccelerator eDepend is easier to configure, faster, more accurate, and applicable to a much wider range of dependencies than existing dependency generation solutions. If your current build does not employ dependency generation, you can enable eDepend and benefit from more accurate incremental builds without the overhead of configuring and integrating an external dependency generation tool.

The following sections describe the dependency generation challenge in more detail and how eDepend can improve your build speed and accuracy.

Dependency Generation

Consider a build tree that looks like this:

```
src/
   Makefile          <--- top-level makefile: recurses into mylib and then into main to build the program
   common/
      header1.h
      header2.h
   mylib/
      Makefile       <--- has rules to build mylib.o and create library mylib.a
      mylib.h
      mylib.c        <--- includes common/header1.h and mylib.h
   main/
      Makefile       <--- has rules to build main.o and then to link main using main.o and mylib.a
      main.c         <--- includes common/header1.h, common/header2.h, and lib/mylib.h
```

The Problem

Even in this simple example, the need for dependency generation is apparent: if you make a change to a header file, how do you ensure dependent objects are recompiled when you rebuild?

Makefiles could explicitly declare all header dependencies, but that quickly becomes too cumbersome: each change to an individual source file may or may not require an adjustment in the makefile. Worse, conditionally compiled code can create so many permutations that the problem becomes intractable.

Another possibility is to declare all headers as dependencies unilaterally, but then the build system becomes very inefficient: after a successful build, a modification to `header2.h` should trigger a rebuild only of the `main` module, not `mylib.a` as well.

Clearly, to get accurate, efficient builds, the system must have calculated dependencies automatically *before* it builds.

There are several ways to generate dependencies and update makefiles to reflect these dependencies (for example, **makedepend** and '**gcc -M**'), but they all have the drawbacks mentioned previously.

eDepend Benefits

ElectricAccelerator eDepend is an Electric Make feature that directly addresses all problems with existing dependency generation solutions. Specifically:

- It is part of Electric Make and requires no external tool to configure, no extra processing time, and it is faster than other solutions.
- It is easily enabled by setting a command-line parameter to Electric Make. No tools or changes to makefiles are required.
- Like ElectricAccelerator Developer Edition itself, it is completely tool and language independent. eDepend automatically records any and all kinds of dependencies, including implicit relationships such as executables on libraries during a link step.
- eDepend dependencies are recorded in Electric Make history files—transparently recorded and used without manifesting as makefile rules.
- eDepend is accurate because it uses file information discovered by the Electric File System at the kernel level as a job executes its commands.

How Does eDepend Work?

Internally, eDepend is a simple application of sophisticated file usage information returned by the Electric File System.

1. As a job runs, the Electric File System records all filenames it accesses inside `EMAKE_ROOT`.

   This function has two very important implications:

   - eDepend can track dependencies within `EMAKE_ROOT` only.
   - eDepend can track dependencies for a job only after it has run—this is why you must start with a complete build rather than an incremental build.

2. After a job completes, Electric Make saves the following eDepend information to the eMake history file:

   - the working directory for the Make instance
   - the target name
   - any files actually read (not just checked for existence) and/or explicitly listed as prerequisites in the makefile

   Note: eDepend information is stored in the history file along with serialization history information. Commands operating on the history file (for example, those specifying file location or that erase it) apply to eDepend information as well.

3. In a subsequent build, whenever Electric Make schedules a target in a directory for which it has eDepend information, it evaluates file dependencies recorded in the earlier run as it checks to see if the target is up-to-date.

   In the example above, the rule to update `mylib.o` may look like this:

   ```
   mylib.o: mylib.c
          $(CC) ...
   ```

`mylib.c` includes `common/header1.h`, which is not explicitly listed as a prerequisite of `mylib.o`, so eDepend records this implicit dependency in the history file.

Directory	Object	Dependency
src/mylib	mylib.o	common/header1.h

If a change is then made to `common/header1.h`, `src/mylib/mylib.o` it will be rebuilt.

Enabling eDepend

1. Start from a clean (unbuilt) source tree.
2. If your build system already has a dependency generation mechanism, turn it off if possible. If you cannot turn it off, you will still get eDepend's additional accuracy, but you will not be able to improve the performance or shortcomings of your existing system.
3. Build your whole source tree with eDepend enabled.

   Use the `--emake-autodepend=1` command-line switch:

   ```
   % emake --emake-autodepend=1 ...
   ```

   Alternatively, insert `--emake-autodepend=1` into the `EMAKEFLAGS` environment variable.

   ```
   % setenv EMAKEFLAGS --emake-autodepend=1
   % emake ...
   ```

4. Make a change to a file consumed by the build, but not listed explicitly in the makefiles.

   For example, touch a header file: `% touch someheader.h`

5. Rebuild, again making sure eDepend is enabled.

   ```
   % emake --emake-autodepend=1 ...
   ```

   Notice that without invoking a dependency generator, Electric Make detected the changed header and rebuilt accordingly.

Important Notes

- The eDepend list is consulted only if all other prerequisites in the makefile indicate the target is up-to-date.

  Explained another way: If a target is out-of-date because it does not exist or because it is older than one of the prerequisites listed in the makefile, eDepend costs nothing and has no effect.

  If the target is newer than all its listed prerequisites, then eDepend is the "11th hour" check to ensure it really is up-to-date, and that there is not a newer implicit dependency. This is the only place eDepend interacts with your build: it forces a target that incorrectly appears to be up-to-date to be rebuilt.

- eDepend information, unlike traditional Makedepend rules, does not in any way imply anything about needing to build or update the implicit prerequisite.

  In the example above, if `header1.h` is renamed or moved, Electric Make just ignores the stale eDepend information. When eMake next updates the `mylib.o` target, it will prune stale dependencies from the eDepend list. This change to the history file occurs regardless of the setting of the `--emake-history-force` parameter to eMake.

  Unlike Make, Electric Make does not complain if it does not have a rule to make `header1.h` because eDepend dependencies are not used to schedule targets.

- eDepend's information scope is bound by a directory name and the target name. This means you can build cleanly from the top of a tree, then run accurate incremental builds from individual subdirectories and eDepend information will be used and updated correctly.

  However, it does imply if you have a build that

  - performs multiple passes or variants over the same directories
  - with exactly the same target names, but
  - runs significantly different commands

  For example, a build that produces objects with the same name for a different architecture or configuration, eDepend information may be over-used unnecessarily. In this case, Electric Make may rebuild more than is necessary, but with no incorrect build results. In this situation, you can achieve fast, correct builds by using separate history files, or ideally, by changing to unique target names across build variants.

Using #pragma noautodep

Some build steps contain many implicit dependencies that may not make sense to check for up-to-dateness. Examples include symbol files consumed by a link step or archive packager input files (for example, *tar*). In both cases, any makefile explicit prerequisites are sufficient to determine if the target should be updated: eDepend information would just add overhead or cause unnecessary rebuilds.

You can selectively disable eDepend information for certain files from any step by supplying Electric Make with the makefile directive:

```
#pragma noautodep *.pdb
%.o: %.c
       $(CL) ...
```

The directive `#pragma noautodep` is applied to the next rule or pattern rule in the makefile. This directive specifies a class of files for eDepend to ignore. Note the following information about `#pragma autodep`:

1. Wildcards allowed for `#pragma autodep`:

   * matches 0 or more characters

   ? matches 1 character

   [ ] matches a range of characters

2. The `noautodep` filters are matched against absolute pathnames. To limit a filter to files in the current directory for the job, use '`./`':

   ```
   #pragma noautodep ./foo.h
   ```

   To specify "ignore foo.h" in any directory, use:

   ```
   #pragma noautodep */foo.h
   ```

3. If the supplied pattern has no wildcards and does not specify a path, it will never match.

   Electric Make ignores the directive and prints a warning as it parses the makefile:

   ```
   Makefile:2: ignoring autodep filter 'foo',
   does not match absolute path(s).
   ```

ElectricAccelerator Ledger File

Traditional Make facilities rely exclusively on a comparison of filesystem timestamps to determine if the target is up-to-date. More specifically, an existing target is considered out-of-date if its inputs have a “last-modified” timestamp later than the target output.

For typical interactive development, this scheme is adequate: As a developer makes changes to source files, their modification timestamps are updated, which signals Make that dependent targets must be rebuilt. There is, however, a class of workflow styles that cause file timestamps to move arbitrarily into the past or future, and therefore circumvent Make's ability to correctly rebuild targets.
Two common examples are:

- Using a version control system that preserves timestamps on checkout (also known as “sync” or “update”).

  The default mode for most source control systems is to set the last-modified timestamp of every file updated in a checkout or sync operation to the current day and time. If you change this behavior to preserve timestamps (or if your tool's default mode is *preserve*), then updating your source files can result in modified contents but with a timestamp in the past (typically, it is the time of the checkin).

- Using file or directory sychronization tools (even simple recursive directory copies) to keep files updated against some other repository.

  Here again, while it is easy to modify source file content, the timestamp for modifications may be any of several possibilities: time of copy, last-modified time of source, last-modified time of destination, and so on.

The Problem

In all modified source files cases, we would like the Make system to rebuild any dependent objects. However, because timestamps of modified files are not set reliably, Make may or may not force a target update. Here is an example Makefile:

```
foo.o: foo.c
        gcc -c foo.c

foo.o: foo.h
```

And a build is run without an existing `foo.o` object:

```
% make
gcc -c foo.c

% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 20 Apr 25 17:34 foo.h
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

The `foo.o` target is updated. Next, suppose we ask our source control system to update the working directory, and it responds by giving us a newer copy of `foo.h`, one that is several weeks newer than what we have, and *that* timestamp is preserved:

```
% <sync>
% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 29 May 17 11:21 foo.h <-- notice timestamp change
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

Traditional Make programs (here, GNU Make) will not notice the change because the timestamp is still in the past, and will incorrectly report that the target is up-to-date.

```
% make
make: `foo.o' is up to date.
```

Some Make facilities (notably, Rational 'clearmake' in conjunction with Rational ClearCase) have the ability to track timestamp information because they are integrated with the source control system.

The Electric Make Solution

Electric Make solves this problem at the file level, completely independent of the source control system, by keeping a separate database of inputs and outputs called a ***ledger***. To use the Ledger, specify which file aspects to check for changes when considering a rebuild. The (nonexclusive) choices are:

- `timestamp` – any timestamp changes to either the target or the explicitly declared dependency, regardless of how it relates to the last modified time of the target input file, triggers a target rebuild.
- `size` – any size change, regardless of the timestamp in the input file, triggers a target rebuild.
- `command` – records the text of the command used to create the target. If makefile or its variables change, using `command` rebuilds the target. **Important caveat:** If you initialize a variable using the $(*shell*) function, be extremely careful to use the $(*shell*) function with a ':=' assignment to avoid re-evaluating it every time the variable is referenced. ':=' simply expanded variables are expanded immediately upon reading the line.
- `nobackup` – suppresses the automatic backup of the ledger file before its use.
- `nonlocal` – instructs eMake to operate on the ledger file in its current location, even if it is on a network volume. By default, if the file specified by `--emake-ledgerfile` (emake.ledger in the current working directory, by default) is not on a local disk, eMake copies that file (if it already exists) to the system temporary directory and opens the copy, then copies it back to the specified location when the build is complete.

  Using `nonlocal` removes a safety and may cause problems if the non-local filesystem has issues with memory-mapped I/O (IBM Rational ClearCase MVFS is known to have issues with memory-mapped I/O). If you are confident that you will get efficient and reliable memory-mapped I/O performance from the non-local filesystem, you can remove the safety for improved efficiency because eMake does not spend time at startup and shutdown copying ledger files. Electric Cloud strongly recommends against using `nonlocal` with ClearCase dynamic views. Electric Cloud does not support Ledger-related problems that occur when `nonlocal` is used in conjunction with the MVFS.

Use the `--emake-ledger=<valuelist>` command-line switch (or the `EMAKE_LEDGER` environment variable) to specify one or more of the following: timestamp, size, command, nobackup, nonlocal (`timestamp`, `size`, `command`, `nobackup`, `nonlocal`). See Ledger options in Electric Make Command-Line Options and Environment Variables.

In the example above, the Ledger can detect if a rebuild is necessary as the timestamps change. If the original build was:

```
% emake --emake-ledger=timestamp
gcc -c foo.c
% <sync> <-- notice timestamp change
% emake --emake-ledger=timestamp
gcc -c foo.c
```

Electric Make consulted the Ledger and concluded the target needed to be rebuilt.

Important Notes for the Ledger Feature

- The Ledger feature works by comparing an earlier input state with the current state: if the Ledger has no information about a particular input (for example, during the first build after it was added to a makefile), it will not contribute in the up-to-dateness check.
- Only one Ledger is used per build.
- The default ledger file is called `emake.ledger`

  It can be adjusted by the `--emake-ledgerfile=<path>` command-line option or `EMAKE_LEDGER_FILE=<path>` environment variable.
- If you specify `--emake-ledgerfile=<path>` but not `--emake-ledger=<valuelist>`, the Ledger still hashes the filenames, so the Ledger is triggered when the filename order changes or a file is added or removed.
- The Ledger automatically backs up the ledger file before using it. This ensures a non-corrupt file is available. If the ledger file is large, copying it could take some time on incremental builds. The ledger option, `nobackup`, suppresses the backup.
- Ledger works for local builds, as well as local submakes in a runlocal job, see Managing Temporary Files.
- It is not possible, however, to share a Ledger between top-level make instances and local-mode submakes running on different Agents. See EMAKE_BUILD_MODE=local in Electric Make Command-Line Options and Environment Variables.
- eMake consults Ledger information to trigger a rebuild only when a target would otherwise be considered up-to-date. Information in the Ledger never prevents a target from being rebuilt.
- In a GNU Make emulation, the Ledger feature changes the meaning of the '`$?`' automatic variable to be synonymous with '`$^`' (all prerequisites, regardless of up-to-dateness).
- You cannot change Ledger options for a particular ledger file—you must use the same combination of timestamp, size, and command that was used to create the ledger file.
- If you turn on `--emake-ledger` and `--emake-autodepend` at the same time, the Ledger keeps track of both implicit and explicit dependencies. This feature is comparable to using ClearMake under ClearCase, but is independent of ClearCase information records.
- Order-only prerequisites, in keeping with their semantic meaning, never affect Ledger behavior.
- Because the Ledger automatically rebuilds a target when there is no existing entry in the ledger file, a build that is using the Ledger for the first time may take longer than expected.

Managing the History Data File

When Accelerator runs a build for the first time, it takes aggressive action to run all jobs as fast as possible in parallel. Jobs that run in the wrong order because of missing makefile dependencies are automatically re-run to

ensure correct output. (These failed steps are referred to also as *conflicts*.)

To avoid the cost of re-running jobs on subsequent builds, Electric Make saves the missing dependency information in a *history data file*. The history data file evolves with each new run and enables Accelerator to continually run builds at peak efficiency.

You can choose the location of the history file and how it is updated.

Setting the History File Location

By default, Electric Make creates the history file in the directory you use to invoke the build and names it `emake.data`.

The file location can be explicitly specified using the command-line option

```
--emake-historyfile=<pathname>
```

Selecting History File Options

The history file (by default, `emake.data`) is used for two separate operations during an Electric Make build:

- Input – Electric Make reads the history file as it starts a build to improve build performance.
- Output – Electric Make *writes* to the history file *as it completes a build* to improve performance of subsequent builds.

Input Rules

If the history file (`emake.data` or whatever was specified with `--emake-historyfile`) exists, it is *always* read and used to improve build performance at the time of build execution.

Output Rules

Data written to the history file when the build completes depends on the `--emake-history` option setting. Three options are available:

1. Merge – By default, Electric Make merges any new dependencies it discovers into the existing history file. In this way, the history file automatically evolves as your makefiles change, learning dependencies that accelerate the builds you run.
2. Create – If `--emake-history` is set to `create`, the old history file contents are completely overwritten by new dependencies discovered in the run that just completed. Use this setting to start a fresh history file, effectively eliminating any stale information from the file.
3. Read – If `--emake-history` is set to `read`, no data is written to the history file at build completion, and any new dependencies Electric Make discovered are discarded. Use this setting when you are sharing a single, static copy of the history file between developers.

By default, the history file is updated even if the build fails, regardless of the set value for `--emake-history`. You can override this behavior by setting the command-line option to `--emake-history-force=0`.

The history file directly impacts the number of conflicts the build can encounter. Ideally, an ElectricAccelerator Developer Edition build with good history should have close to “0” conflicts. If you see conflicts starting to rise, ensure you have a current history file for the build you are executing.

Important Notes for Creating and Using the History File

- One-Agent build to guarantee correct history – Some parallel builds will not succeed without a good history file. In particular, builds that use wildcard or globbing operators to produce build generated lists of files and operate on those lists may fail. For example, a makefile may use `ld *.o` as shorthand to

avoid enumerating all the `*.o` files in a directory. Running the build against one Agent (different from running in local mode—local builds do not use history) guarantees the build will succeed and a history file is created for use by subsequent parallel builds.

- Relative `EMAKE_ROOT` locations must match – The history file records target filenames relative to the `EMAKE_ROOT` specified during that run. For a subsequent build to correctly use the history file, target filenames must have the same path name relative to the eMake root.

  For example:

  If your eMake root is set to: `/home/alice/builds`
  and your build references a path name in that root: `/home/alice/builds/lib/foo.o`
  then the history file records it as `lib/foo.o`
  If a subsequent build sets the eMake root to: `/home/bob/builds`
  the history file will match correctly.

  If, however, the eMake root is set to: `/home/bob`
  then the file that exists on the disk as: `/home/bob/builds/lib/foo.o`
  gets the root-relative name of: `builds/lib/foo.o`
  which does not match the name `lib/foo.o` in the history file generated above.

  Because the history file does not match, performance can suffer.

  Note: Be sure your `EMAKE_ROOT` matches the same location relative to sources as the `EMAKE_ROOT` used to create the history file you are using.

- For builds with multiple roots, the roots must have the same alphabetical sorting order in each build in order for history to match.

Chapter 14: Annotation

As Electric Make runs a build, it discovers a large amount of information about the build structure. This information can be written to an “annotation” file for use after the build completes. Annotation information is represented as an XML document so it is easy to parse.

eMake collects many different types of information about the build depending on various command-line switches. The information eMake collects includes:

- makefile structure
- commands and command output
- list of file accesses by each job
- dependencies between jobs
- detailed timing measurements
- eMake invocation arguments and environment
- performance metrics

Configuring eMake to Generate an Annotation File

By default, Electric Make collects configuration information and performance metrics only.

Optionally, eMake can be configured to collect additional information. This extra annotation information is written to an XML file in the build directory (`emake.xml` by default). The `--emake-annodetail` command-line switch controls the amount of information eMake should collect.

Supported annotation detail flags:

- **basic -** The basic mode collects information about every command run by the build. Detailed information about each “job” in the build is recorded, including command arguments, output, exit code, timing, and source location. In addition, the build structure is represented as a tree where each recursive make level is represented in the XML output.
- **env** - The env mode adds information about environment variable modifications.
- **file -** The file mode adds information about files read or written by each job.
- **history -** The history mode adds information about missing serializations discovered by eMake. This includes information about which file caused two jobs to become serialized by the eMake history mechanism.

- **lookup -** The lookup mode adds information about files that were looked up by each job. **Note:***This mode can cause the annotation file to become quite large.*
- **md5** - The md5 mode computes MD5 checksums for files read and written by the build, and includes that information in annotation as an md5 attribute on appropriate `<op>` tags. The list of operation types that will include the checksum is read, create, and modify. No checksum is generated or emitted for operations on directories or symlinks, or for append operations. If a read file was appended to, and the read occurs before the appended update is committed, you will see a zero checksum on that read operation (by design because reading files that were appended to is a rare occurrence). The md5 mode implies “file” level annotation. This mode is configurable through the command-line only; it is not available on the web interface.
- **registry -** The registry mode adds information about registry operations.
- **waiting -** The waiting mode adds information about the complete dependency graph for the build.

All of the detail settings automatically enable “basic” annotation.

The `--emake-annoupload` command-line switch controls whether or not eMake sends a copy of the annotation file to the Cluster Manager as the build runs. By default, eMake sends minimal information to the Cluster Manager, even if more detailed annotation is enabled. eMake sends the full annotation file if annotation uploading is configured by the build class or by the eMake command line.

Note: You cannot disable `mergestreams` if you enable annotation. Enabling annotation automatically enables `mergestreams`, even if it was explicitly disabled on the command line.

Annotation File Splitting

Because of limitations in ElectricInsight (a 32-bit application), eMake automatically partitions annotation files into 1.6 GB “chunks.” The first chunk is named using the file name that you specify with the `--emake-annofile` option or with “emake.xml,” if `--emake-annofile` is not specified. The second chunk uses that name as the base but adds the suffix _1, the third chunk adds the suffix _2, and so on. For example, a four-part annotation file might consist of files named emake.xml, emake.xml_1, emake.xml_2, and emake.xml_3.

No special action is required to load a multipart annotation file into ElectricInsight. If all parts are present in the same directory, ElectricInsight automatically finds and loads the content of each file—simply specify the name of the first chunk when opening the file in ElectricInsight.

Working with Annotation Files

The simplest way to use an eMake annotation file is to load it into the Electric Cloud ElectricInsight® product. This program allows the user to see a graphical representation of the build, search the annotation file for interesting patterns, and perform sophisticated build analysis using its built-in reporting tools.

Also, you can write your own tools to perform simple tasks using annotation output. For example, reporting on failures in the build can be accomplished by looking for “failed” elements inside job elements and then reporting various details about the failed job such as the commands, their output, and the line of the makefile that contains the rule for the command. Refer to the DTD for the annotation file format below.

Annotation XML DTD

```
<!-- build.dtd -->
<!-- The DTD for Emake's annotated output. -->
<!-- -->
<!-- Copyright (c) 2004-2008 Electric Cloud, Inc. All rights reserved. -->

<!ENTITY % hexnum "CDATA">
<!ENTITY % job "(message*, job)">
<!ENTITY % valueName "name NMTOKEN #REQUIRED">
```

```
<!-- Can't use NMTOKEN because Windows has environment variables like
    "=D:". -->
<!ENTITY % envValueName "name CDATA #REQUIRED">

<!ELEMENT build
    (properties?, environment?, (message* | make)+, fs?, metrics? )
>
<!ATTLIST build
    id    CDATA #REQUIRED
    cm    CDATA #IMPLIED
    start CDATA #REQUIRED
>
<!-- Out of band build messages -->

<!ELEMENT message (#PCDATA) >
<!ATTLIST message
    thread    %hexnum;            #REQUIRED
    time      CDATA               #REQUIRED
    code      CDATA               #REQUIRED
    severity  ( warning | error ) #REQUIRED
>
<!-- Properties list -->

<!ELEMENT properties (property*) >
<!ELEMENT property   (#PCDATA) >
<!ATTLIST property
    %valueName;
>
<!-- Environment list -->

<!ELEMENT environment (var*) >
<!ELEMENT var   (#PCDATA) >
<!ATTLIST var
    %envValueName;
    op ( add | modify | delete ) "add"
>
<!-- File system dump -->

<!ELEMENT fs      (roots, symRoots, (content|name)*) >
<!ELEMENT roots   (root+) >
<!ELEMENT root    (#PCDATA) >
<!ATTLIST root
    nameid CDATA #REQUIRED
>
<!ELEMENT symRoots (symRoot*) >
<!ELEMENT symRoot (#PCDATA) >
<!ATTLIST symRoot
    symLinkPath CDATA #REQUIRED
>
<!ELEMENT content (contentver+)>
<!ATTLIST content
    contentid CDATA #REQUIRED
>
<!ELEMENT contentver EMPTY>
<!ATTLIST contentver
    job CDATA #REQUIRED
>
<!ELEMENT name (namever*) >
<!ATTLIST name
```

```
    nameid CDATA #REQUIRED
    dir    CDATA #REQUIRED
    name   CDATA #REQUIRED
>
<!ELEMENT namever EMPTY>
<!ATTLIST namever
    job       CDATA #REQUIRED
    contentid CDATA #REQUIRED
>
<!-- Metrics list -->

<!ELEMENT metrics (metric*) >
<!ELEMENT metric  (#PCDATA) >
<!ATTLIST metric
    %valueName;
>
<!-- Make subtree -->

<!ELEMENT make
    (environment?, ( message | job | make )*)
>
<!ATTLIST make
    level CDATA #REQUIRED
    cmd   CDATA #REQUIRED
    cwd   CDATA #REQUIRED
    mode  ( gmake | nmake | symbian ) #REQUIRED
>
<!-- Job -->

<!ELEMENT job
    (environment?,(output | command | conflict)*,depList?,opList?,
registryOpList?,timing+,failed?,waitingJobs?)
>
<!ATTLIST job
    thread    %hexnum;  #REQUIRED
    id        ID    #REQUIRED
    status    ( normal | rerun | conflict | reverted | skipped ) "normal"
    type      ( continuation | end | exist |
                follow | parse | remake | rule ) #REQUIRED
    name      CDATA #IMPLIED
    file      CDATA #IMPLIED
    line      CDATA #IMPLIED
    neededby  IDREF #IMPLIED
    partof    IDREF #IMPLIED
    node      CDATA #IMPLIED
>
<!-- Command and related output, output blocks can contain nested -->
<!-- make subtrees in local mode. -->

<!ELEMENT command
    (argv,inline*,(output | make)*)
>
<!ATTLIST command
    line  CDATA #IMPLIED
>
<!ELEMENT argv (#PCDATA)>
<!ELEMENT inline (#PCDATA)>
<!ATTLIST inline
```

```
    file CDATA #REQUIRED
>
'<!ELEMENT output (#PCDATA)>
<!ATTLIST output
    src   ( prog | make ) "make"
>
<!-- Conflict description -->

<!ELEMENT conflict EMPTY>
<!ATTLIST conflict
    type       ( file | cascade | name | key | value ) "cascade"
    writejob   IDREF #IMPLIED
    file       CDATA #IMPLIED
    rerunby    IDREF #IMPLIED
    hkey       CDATA #IMPLIED
    path       CDATA #IMPLIED
    value      CDATA #IMPLIED
>
<!-- Job failure code -->

<!ELEMENT failed EMPTY>
<!ATTLIST failed
    code CDATA #REQUIRED
>
<!-- List of jobs waiting for this job, local mode only -->

<!ELEMENT waitingJobs EMPTY>
<!ATTLIST waitingJobs
    idList IDREFS #IMPLIED
>
<!-- Start and stop times of this job -->

<!ELEMENT timing EMPTY>
<!ATTLIST timing
    invoked   CDATA #REQUIRED
    completed CDATA #REQUIRED
    node      CDATA #IMPLIED
>
<!-- Dependency list, only used when annoDetail includes 'history' -->

<!ELEMENT depList (dep*)>
<!ELEMENT dep     EMPTY>
<!ATTLIST dep
    writejob IDREF #REQUIRED
    file     CDATA #REQUIRED
>
<!-- Operation list, only present when annoDetail includes -->
<!-- 'file' or 'lookup' -->

<!ELEMENT opList (op*)>
<!ELEMENT op     EMPTY>
<!ATTLIST op
    type     ( lookup | read | create | modify | unlink | rename |
               link | modifyAttrs | append | blindcreate ) #REQUIRED
    file     CDATA #REQUIRED
    other    CDATA #IMPLIED
    found    ( 1 | 0 ) "1"
    isdir    ( 1 | 0 ) "0"
    filetype ( file | symlink | dir ) "file"
```

```
    atts     CDATA #IMPLIED
>
<!-- Registry operation list, only present when annoDetail includes -->
<!-- 'registry' -->

<!ELEMENT registryOpList (regop*)>
<!ELEMENT regop     (#PCDATA)>
<!ATTLIST regop
    type     ( createkey | deletekey | setvalue | deletevalue |
               lookupkey | readkey ) #REQUIRED
    hkey     CDATA #REQUIRED
    path     CDATA #REQUIRED
    name     CDATA #IMPLIED
    datatype ( none | sz | expandsz | binary | dword | dwordbe |
               link | multisz | resourcelist | resourcedesc |
               resourcereqs | qword ) "none"
>
```

Metrics in Annotation Files

Some metrics and timers may be applicable to ElectricAccelerator only, not ElectricAccelerator Developer Edition.

Metric	Description
chainLatestReads	Number of times the latest version of an FSChain was requested.
chainSerialReads	Number of times an FSChain was requested to match a particular spot in the serial order.
chainWrites	Number of new versions created, which occurs any time a new name is created, or the content of a file changes.
compressBytesIn	The number of bytes passed in for compression.
compressBytesOut	The number of bytes returned from compression.
compressTime	Time spent compressing data.
conflicts	Number of jobs that ran into conflicts and had to be re-run.
decompressBytesIn	The number of bytes passed in for decompression.
decompressBytesOut	The number of bytes returned from decompression.
decompressTime	Time spent decompressing data.
diskReadBytes	The number of bytes read from the local disk.
diskReadTime	Time spent reading data from the local disk.
diskReadWaitTime	Time spent waiting for data to be read from the local disk.

Metric	Description
diskWriteBytes	The number of bytes written to the local disk.
diskWriteTime	Time spent writing data to the local disk.
diskWriteWaitTime	Time spent waiting for data to be written to the local disk.
duration	Duration of build.
elapsed	The total elapsed time for the build.
emptyNameVersions	Total number of Name versions with no associated Content. Available with `--emake-debug=g`.
fitness	Indicates the "fitness" of the history, which is how closely the history used for a given build "matched" that build. The range of values is 0 (history did not match at all) to 1 (history already had information about all implicit dependencies). Available with `--emake-debug=g`.
freezeTime	Time the job queue was frozen, which means only high priority items are taken of the queue.
hiddenWarningCount	The number of warning messages hidden by the eMake client and all remote parse jobs, with one count for every message number for which at least one message was hidden. The count does not include messages hidden by eMake stubs or rlocal-mode emakes.
localAgentReadBytes	Bytes copied from local agents.
localAgentReadTime	Time spent copying data from local agents.
localAgentWriteBytes	Bytes copied to local agents.
localAgentWriteTime	Time spent copying data to local agents.
maxArenaCount	Peak active arenas during the build.
maxMakeCount	Peak active make instances during the build.
noAgentsWaitTime	Amount of time spent waiting with no agents allocated to the build.
runjobs	Number of jobs that did work.
symlinkReads	The number of times the application read symlinks from the local disk.

Metric	Description
termDiskCopiedBytes	The number of bytes committed by copying. This should be a small number if possible. If you are using ClearCase and this number is nonzero, look into the `--emake-clearcase=vob`s option.
termDiskCopiedFiles	The number of files committed by copying. This number should be small if possible.
termDiskMovedBytes	The number of bytes committed by moving. You want this to be big number if possible.
termDiskMovedFiles	The number of files committed by moving. You want this to be a big number if possible.
termDiskRemovedBytes	The number of bytes committed by remove (original file).
termDiskRemovedFiles	The number of files committed by remove (original file).
termDiskRemoved TmpBytes	The number of bytes committed by remove (temporary file).
termDiskRemoved TmpFiles	The number of files committed by remove (temporary file).
terminated	Number of jobs that ran to completion. These jobs may not necessarily have done anything.
totalChains	Total number of FSChains in the build—anything in the file system tracked by eMake for versioning is an FSChain, for example, file names, file contents. Because both names and contents are tracked, this should be at least twice the number of files accessed in the build.
totalNameVersions	Total number of Name versions. Available with `--emake-debug=g`.
totalVersions	Total number of FSChain versions. Available with `--emake-debug=g`.
usageBytes	The number of bytes received for usage data.
writeThrottleWaitTime	Amount of time spent waiting for the write throttle (which is in place to avoid slowing the build by seeking the disk head back and forth too often).

Timers

Note that most timers will not be available unless you run with `--emake-debug=g` (for profiling). These timers correspond to the amount of time eMake spent in certain areas of the code or in a certain state.

Metric	Description
timer:agentManager.startup	The time spent for the Agent Manager to start up.

Metric	Description
timer:agentManager.stop	Time spent for the Agent Manager to shut down.
timer:agentManager.work	Time spent with the Agent Manager actively doing work outside shouldRequestAgents.
timer:annoUpload.startup	Time spent starting up the thread to upload annotation.
timer:annoUpload.work	Time spent uploading annotation.
timer:bench	Benchmark showing the cost of 100 invocations of the timer code (start/stop).
timer:directory.populate	Time spent making sure that eMake's model of the directory contents is fully populated.
timer:history.parsePrune	Time spent in parse jobs signaling stale history entries to prune stale events.
timer:history.pruneFollowers	Time spent signaling stale submakes (for jobs that have followers) to be pruned.
timer:history.pruneNo Follower	Time spent signaling stale submakes (for jobs that have no followers) to be pruned in jobs.
timer:idle.agentManager	Time spent in the Agent Manager sleeping between shouldRequestAgents checks.
timer:idle.agentRun	Time spent by worker threads waiting for requests from the agent.
timer:idle.annoUpload	Time spent by the annotation upload thread waiting for data.
timer:idle.noJobs	Time spent by worker threads waiting for a new runnable job to enter the job queue.
timer:idle.untilCompleted	Time spent by the Terminator thread waiting for jobs to be completed.
timer:idle.waitForAgent	Time spent by worker threads waiting for an agent to become available.
timer:jobqueue	Time spent within the lock guarding the job queue.
timer:Ledger.close	Time spent closing the Ledger and flushing data to disk.
timer:Ledger.commit	Time spent committing Ledger data to the database.
timer:Ledger.isUpToDate	Time spent querying the Ledger to find if a file is up to date.

Metric	Description
timer:Ledger.staleAttributes	Time spent by the Ledger code statting files to ensure recorded attributes match the actual attributes on disk.
timer:Ledger.update	Time spent updating the Ledger.
timer:main.commit	Time spent by the Terminator thread committing jobs, less time spent flushing deferred writes to disk.
timer:main.history	Time spent reading and writing history files.
timer:main. lockedWriteToDisk	Time spent by the Terminator thread flushing deferred writes to disk.
timer:main.terminate	Time spent by the Terminator thread terminating jobs.
timer:main.writeToDisk	Time spent by the Terminator thread writing operations to disk, less the time covered by any of the other main.writeToDisk timers.
timer:main.writeToDisk. append	Time spent appending to existing files on disk.
timer:main.writeToDisk. createdir	Time spent creating directories on disk.
timer:main.writeToDisk. createdir.attrs	Time spent changing directory attributes on disk for newly created directories.
timer:main.writeToDisk. createdir.chown	Time spent changing file ownership on disk for newly created files.
timer:main.writeToDisk. createdir.chown	Time spent changing directory ownership on disk for newly created directories.
timer:main.writeToDisk. createdir.times	Time spent changing directory times on disk for newly created directories.
timer:main.writeToDisk. createfile	Time spent writing file data to disk for newly created files.
timer:main.writeToDisk. createfile.attrs	Time spent changing file attributes on disk for newly created files.
timer:main.writeToDisk. createfile.times	Time spent changing file times on disk for newly created files.
timer:main.writeToDisk.link	Time spent creating links on disk.

Metric	Description
timer:main.writeToDisk. modify	Time spent modifying existing files on disk.
timer:main.writeToDisk. modifyAttrs	Other time spent in writing attribute changes to disk (mostly notifying the file system that attributes have gone “stale”).
timer:main.writeToDisk. modifyAttrs.attrs	Time spent modifying attributes of existing files on disk.
timer:main.writeToDisk. modifyAttrs.chown	Time spent modifying ownership of existing files on disk.
timer:main.writeToDisk. modifyAttrs.times	Time spent modifying times of existing files on disk.
timer:main.writeToDisk. unlink	Time spent unlinking existing files on disk.
timer:main.writeToDisk. unlink.data	Time spent recording the fact that a file was removed.
timer:main.writeToDisk. unlink.tree	Time spent removing entire trees on disk.
timer:mergeArchiveRefs	Time spent modifying word lists for multi-word archive references like `lib(member1 member2 ...)`.
timer:mutex.DirCache	Time spent waiting for a directory cache.
timer:mutex.filedata.nodelist	Time spent within the lock guarding the list of agents allocated to the build.
timer:mutex.jobcreate	Time spent within the lock used to synchronize the terminator and worker threads when creating new jobs.
timer:mutex.joblist	Time spent within the lock used to protect the job list.
timer:mutex.jobrunstate	Time spent within the lock used to coordinate starting and canceling jobs.
timer:mutex.nodeinit	Time spent within the lock used to protect the list of hosts while initializing agents.
timer:mutex.target	Time spent within the lock protecting failure tracking on a target.
timer:node.putAllVersions. getShortName	Time spent getting file short names on Windows when doing an E2A_ PUT_ALL_VERSIONS.

Metric	Description
timer:node.setup	Time spent connecting to hosts and initializing them.
timer:node.svc.getData	Time spent handling A2E_GET_FILE_DATA and A2E_RESEND_ FILE_DATA, not including the time spent sending data in response.
timer:node.svc.getData. acquireLock	Time spent waiting for the ChainLock when handling A2E_GET_ FILE_DATA and A2E_RESEND_FILE_DATA.
timer:node.svc.getData.copy	Time spent copying file data for E2A_LOAD_LOCAL_FILE. This is the local agent version of timer:node.svc.getData.send.
timer:node.svc.getData. insideLock	Time spent holding the ChainLock when handling A2E_GET_FILE_ DATA and A2E_RESEND_FILE_DATA.
timer:node.svc.getData.send	Time spent sending file data during E2A_PUT_FILE_DATA and E2A_ PUT_BIG_FILE_DATA.
timer:node.svc.getDir	Time spent sending directory entry data to the agent.
timer:node.svc.getVersions	Time spent handling A2E_GET_ALL_VERSIONS requests.
timer:node.svc.getVersions. acquireLock	Time spent waiting for the ChainLock when handling A2E_GET_ALL_ VERSIONS.
timer:node.svc.getVersions. insideLock	Time spent holding the ChainLock when handling A2E_GET_ALL_ VERSIONS.
timer:node.svc.runCommand	Time spent handling A2E_RUN_COMMAND requests.
timer:usage.io	Time spent reading and responding to usage data.
timer:usage.io.makedata	Time spent creating file data from usage.
timer:usage.io. makedata.local	Time spent copying file data from usage reported by local agents; a subset of timer:usage.io.makedata.
timer:usage.latency	Time spent dispatching incoming usage data and saving output files.
timer:usage.record	Time spent recording usage data, including resolving new name IDs, removing duplicate lookup records, and so on.
timer:worker.continuationjob	Time spent by worker threads running continuationJobs.
timer:worker.endjob	Time spent by worker threads running endJobs.
timer:worker.existencejob	Time spent by worker threads running existenceJobs.
timer:worker.followjob	Time spent by worker threads running followJobs.

Metric	Description
timer:worker.invoke	Time spent invoking remote jobs.
timer:worker.invokelocal	Time spent invoking local jobs.
timer:worker.other	Otherwise unaccounted time spent by worker threads.
timer:worker.parsejob	Time spent by worker threads running parseJobs.
timer:worker.remakejob	Time spent by worker threads running remakeJobs.
timer:worker.rulejob	Time spent by worker threads running ruleJobs, less the time spent actually running commands and figuring out if it needs to run.
timer:worker.rulejob. needtorun	Time spent by ruleJobs figuring out if they need to run.
timer:worker. runcommands	Time spent by worker threads running commands in commandJobs.
timer:worker. shouldRequestAgents	Time spent by worker threads checking to see if they should request agents.
timer:worker.startup	Time spent by worker threads initializing.
timer:worker.stop	Time spent shutting down worker threads.
timerThreadCount	Number of threads in eMake.

Chapter 15: Third-party Integrations

The following topics provide information about how ElectricAccelerator Developer Edition integrates with the following environments:

- ClearCase
- Cygwin
- Eclipse

Using ClearCase with ElectricAccelerator Developer Edition

If your build environment relies on the ClearCase source control system, there are some special considerations for running eMake. ClearCase views can be either “snapshot” or “dynamic.”

- ClearCase *snapshot views* behave like a normal file system, so no special support is required.
- ClearCase *dynamic views* have non-standard file system behavior that requires explicit handling by eMake.

Configuring ElectricAccelerator Developer Edition for ClearCase

LD_LIBRARY_PATH

Using ElectricAccelerator Developer Edition in a ClearCase environment requires your LD_LIBRARY_PATH (on UNIX) or PATH (on Windows) to contain a directory that includes libraries required to run “cleartool.” Library filenames for Windows or UNIX begin with “libatria” (Windows - *libatria.dll*, UNIX - *libatria.so)*.

If you plan to use ClearCase with eMake, you must add the ClearCase shared libraries to the LD_LIBRARY_PATH on your system.

For sh:

```
LD_LIBRARY_PATH=/usr/atria/linux_x86/shlib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

For csh:

```
setenv LD_LIBRARY_PATH /usr/atria/linux_x86/shlib:${LD_LIBRARY_PATH}
```

(`/usr/atria/linux_x86/shlib` is an example and may differ on your system depending on what OS you use and where ClearCase is installed.)

To ensure ElectricAccelerator Developer Edition knows where ClearCase is installed, edit `/etc/ld.so.conf` to include the ClearCase installation location. As a second option, you can include the ClearCase installation location in LD_LIBRARY_PATH.

ClearCase Views

When ElectricAccelerator Developer Edition replicates a ClearCase view on an agent, it appears as a generic file system—ClearCase commands that run as part of a build will not work on the host. The Electric File System masks ClearCase’s VOB mounts. If your build runs ClearCase commands, these commands must be *runlocal* steps. For additional information, see Managing Temporary Files.

Note: Because of the potential adverse interaction between two different filesystems (ClearCase and ElectricAccelerator Developer Edition), Electric Cloud recommends that you do ***not*** install ClearCase on the ElectricAccelerator Developer Edition machine.If you must run ClearCase on the ElectricAccelerator Developer Edition machine, ensure that whichever one you need to start and stop frequently is configured to start “second” at system startup time.

--emake-clearcase

The Electric Make command-line option `--emake-clearcase` controls which ClearCase features are supported for a build. By default, ClearCase integration for `rofs`, `symlink`, and `vobs` is disabled. To turn on support for these specific ClearCase features, if your build relies on these options, use `--emake-clearcase=LIST`, where `LIST` is a comma-separated list of one or more of the following values:

- `rofs` : detect read-only file systems

  eMake queries ClearCase for each file it accesses to determine whether the file should be considered 'read-only'.

- `symlink` : detect symbolic links (Windows only)

  eMake queries ClearCase for each file it accesses to determine whether the file is a ClearCase symbolic link.

- `vobs` : configure separate temporary directories for each vob

  Normally, eMake uses the `--emake-tmpdir` setting to determine where to place temporary directories for each device. With the 'vobs' option enabled, eMake automatically configures one directory per VOB. On Windows, eMake also communicates with ClearCase to determine which VOB a file belongs to so it can select the correct temporary directory.

Note: If `--emake-clearcase` is not specified on the command line and the environment variable `EMAKE_CLEARCASE` is present, Electric Make takes the options from the environment.

eMake's "fake" Interface for ClearCase

In addition to a direct interface to ClearCase, eMake also provides a "fake" interface that allows the end user to pass information manually to eMake about the ClearCase environment. Normally, you invoke ClearCase functionality by specifying `--emake-clearcase=LIST` to eMake, at which point eMake attempts to load ecclearcase6.so and ecclearcase7.so (.dll on Windows). Whichever library successfully initializes in the ClearCase environment is used to talk to ClearCase through a provided API that is no longer maintained or supported. You can specify the precise library to load by setting the environment variable `EMAKE_CLEARCASE_LIBRARY` to the path to the desired library.

Under some conditions, the ClearCase API does not function properly. For this circumstance, eMake provides ecclearcase_fake.so (.dll on Windows). If you point EMAKE_CLEARCASE_LIBRARY to the fake interface, eMake loads that instead. The fake interface then loads the file specified in the environment by ECCLEARCASE_FAKE_INI, defaulting to ecclearcase_fake.ini. The `ini` file has two sections: `[vobs]` and `[attrs]`.

The `[vobs]` section maps a VOB path to a comma-separated set of attributes. Currently, `public` should be present for a public VOB and `ro` for read-only.

The `[attrs]` section maps a filename to *symlink***type*, where *symlink* may be empty if the file is not a symbolic link and *type* can be `null, version, directory_version, symbolic_link, view_private, view_derived, derived_object, checked_out_file, checked_out_dir`. If *symlink* is not empty, `symbolic_link` is assumed. If *type* is `version` or `directory_version` and `--emake-clearcase=rofs` is active, the EFS returns EROFS (or STATUS_ACCESS_DENIED on Windows) when an attempt is made to write the file.

If CLEARCASE_ROOT is set in the environment (as by `cleartool setview`), all `[attrs]` entries are tracked under their exact path as well as one with the CLEARCASE_ROOT prepended. If CLEARCASE_ROOT is set to `/view/testview`, setting `/vobs/test/symlink2` in `[attrs]` is the same as setting both `/vobs/test/symlink2` and `/view/testview/vobs/test/symlink2`.

Sample `ini` files for UNIX:

```
[vobs]
/vobs/test=public
/vobs/readonly=public,ro

[attrs]
/vobs/test/symlink2=symlink
/vobs/test/symlink/alpha=*directory_version
/vobs/test/symlink/beta=alpha
```

and Windows:

```
[vobs]
\test=public
```

```
\readonly=public,ro

[attrs]
S:/test/symlink2=symlink
S:/test/symlink/alpha=*directory_version
S:/test/symlink/beta=alpha
```

Where ClearCase Dynamic Views Affect eMake Behavior

Read-only mounts

ClearCase can mount files in a read-only mode, which means they appear to be writable, but any attempts to modify these files fail with a “read-only file system” (UNIX) or “access denied” (Windows) error message. Because eMake cannot tell whether a file is modifiable using normal file system interfaces, it does not know to disallow modifications performed by commands running on the agents. This activity leads to failures when eMake attempts to commit changes (incorrectly) allowed on the agent.
A simple test case:

```
unix% cleartool ls
Makefile
clock@@/main/2  Rule: /main/LATEST

unix% cat Makefile
all:
mv clock clock.old
```

The file “clock” is checked in to ClearCase. Makefile attempts to rename it. If you just run “make”, it fails immediately, but can be instructed to ignore the error:

```
unix% make -i
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
```

Note that this filesystem is ***not*** mounted *read-only,* so a Makefile can be created. Because “clock” is checked-in, it cannot be renamed without checking it out first, and ROFS is the error ClearCase gives.

Now try this with eMake:

```
unix% emake --emake-root=/vobs -i
Starting build: 114626
mv clock clock.old
ERROR EC1124: Unable to rename file
/vobs/test/drivel/clock to /vobs/test/drivel/clock.old: Read-only
file system (error code 0x1e): Read-only file system
Interrupted build: 114626   Duration: 0:00 (m:s)
```

Without activating ClearCase support, eMake does not know “clock” cannot be moved, so the operation succeeds on the agent, then fails when eMake attempts to commit it to disk. Specifying the “`-i`” flag to ignore errors will not work here.

```
unix% /home/user/Projects/4.2/i686_Linux/ecloud/emake
/emake --emake-clearcase=rofs --emake-root=/vobs -i
Starting build: 114630
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
Finished build: 114630   Duration: 0:00 (m:s)
```

When eMake knows to replicate ClearCase’s behavior, the error occurs on the host and can be handled normally.

Multiple VOBs

eMake writes uncommitted files into temporary directories, and moves them into their correct location after resolving any conflicts. eMake automatically places a temporary directory in the current working directory where it is invoked, and also creates a temporary directory in each location specified by the `--emake-tmpdir` option or the `EMAKE_TMPDIR` environment variable. When possible, eMake writes uncommitted files to the same physical device where the file will be saved when it is committed, which makes the commit operation a lightweight "rename" instead of a heavyweight "copy" operation.

Under ClearCase, each VOB functions as a separate physical disk, so to achieve optimal performance, a temporary directory must be specified for each VOB where the build writes files. `--emake-clearcase=vobs` sets up this directory for you automatically.

- On UNIX, each VOB has a distinct physical device ID, and this option is nothing more than a "shorthand" for specifying `EMAKE_TMPDIR=/vobs/foo:/vobs/bar:...` in the environment.
- On Windows, you must interface with ClearCase directly to make this distinction, so using `--emake-clearcase=vobs` is important to get the most speed for a build that writes to multiple VOBs.

Symbolic links

On Windows, ClearCase conceals the nature of its symbolic links from other programs, so what is actually a single file appears to be two different files to other programs. This situation creates an issue for eMake's versioning mechanism as it tracks two separate chains of revisions for one underlying entity; a job's view of the file can get out of sync and lead to build failures. `--emake-clearcase=symlinks` interfaces directly with ClearCase to determine whether a particular ClearCase file is a symbolic link and represents it on the agent as a reparse point, which is the native Windows equivalent of a symbolic link. All file operations are redirected to the target of the symbolic link, avoiding synchronization problems.

This issue does not occur on UNIX platforms because ClearCase uses native file system support for symbolic links.

A simple test case:

Beginning with a directory, "alpha", and a symlink to that directory, "beta":

```
windows% cleartool ls -d alpha beta
alpha@@/main/1   Rule: \main\LATEST
beta --> alpha
```

And a makefile:

```
all:
        @echo "Furshlugginer" > alpha/foo
        @echo "Potrzebie" > beta/foo
        @cat alpha/foo

windows% emake --emake-root=. -f symlink.mk
Starting build: 50070 Furshlugginer
Finished build: 50070   Duration: 0:02 (m:s)

windows% emake --emake-root=. -f symlink.mk --emake-clearcase=symlink
Starting build: 50071 Potrzebie
Finished build: 50071   Duration: 0:01 (m:s)
```

Explanation: ClearCase has no way to tell the Windows file system that the symlink is a symlink, so `alpha/foo` and `beta/foo` appear to be distinct files. (On UNIX, this is not an issue because symlinks are a standard operating system feature, which means ClearCase can show them as such.) If a build does not contain any ClearCase symbolic links, there is no reason to turn on the integration; if it does, though, you can run into the problem where eMake assumes there are two different files when there is actually just one underlying file, in which case you need to turn on the "symlink" part of our ClearCase integration.

Performance Considerations

Running builds from ClearCase dynamic views can impose a considerable performance cost depending on the ClearCase configuration and your build. The best performance is achieved by using ClearCase snapshot views. If using snapshots is not possible, there are a few things to consider when setting up an eMake build.

Enabling the “symlink” or “rofs” options incurs a performance cost because of the need to communicate with the ClearCase server when accessing a file for the first time. Many builds do not need these features, even if they are running inside a ClearCase dynamic view, so consider leaving them turned off unless you encounter unexpected build failures.

Enabling the “vobs” option should have minimal performance cost, and may significantly speed up your build if build output is written back to your dynamic view.

Because of improved caching, eMake may provide a significant performance boost beyond that provided by running build steps in parallel. eMake caches much of the file system state, reducing the total number of requests to the ClearCase server during the build. Depending on how heavily loaded your ClearCase server is, this can significantly improve build performance. If you notice build speedups higher than you would expect given the number of agents, improved caching may be the reason.

Using the “fake” interface for ClearCase (see eMake’s “fake” Interface for ClearCase), which lets you specify the details of VOBs and files in a static file, is much faster than communicating with ClearCase. This may suffice for many users.

Using Cygwin with ElectricAccelerator Developer Edition (Windows Only)

Cygwin is a Linux-like environment for Windows that consists of two parts:

- A DLL (`cygwin1.dll`) that acts as a Linux API emulation layer, providing substantial Linux API functionality.
- A collection of tools that provide a Linux look and feel.

If your builds used gmake in a Cygwin environment, you might need to use Electric Make’s `--emake-emulation=cygwin` option.

For more information about other Cygwin-specific Electric Make command-line options and corresponding environment variables, see Windows-specific Commands. Specifically, the following command-line options:

`--emake-cygwin=<`*`Y|N|A`*`>`

and

`--emake-ignore-cygwin-mounts=<`*`mounts`*`>`

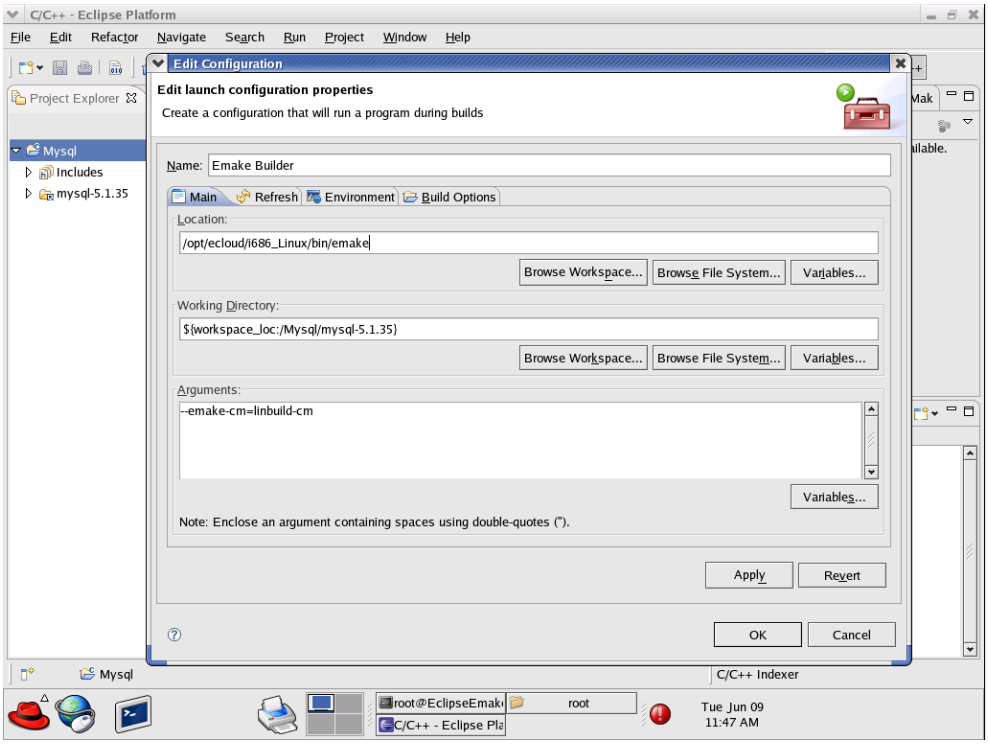
Using Eclipse with ElectricAccelerator Developer Edition

To configure Eclipse to run eMake, follow this procedure:

1. Open your C++ project.
2. Go to the project’s Properties > Builders and click **New**.
3. Select **Program** and click **OK**.
4. Fill-in the following information for the new builder under the Main tab:
   - Name
   - Location (the full path to `emake`, which is OS dependent)

- Working Directory
- Arguments (arguments are specific to your configuration)

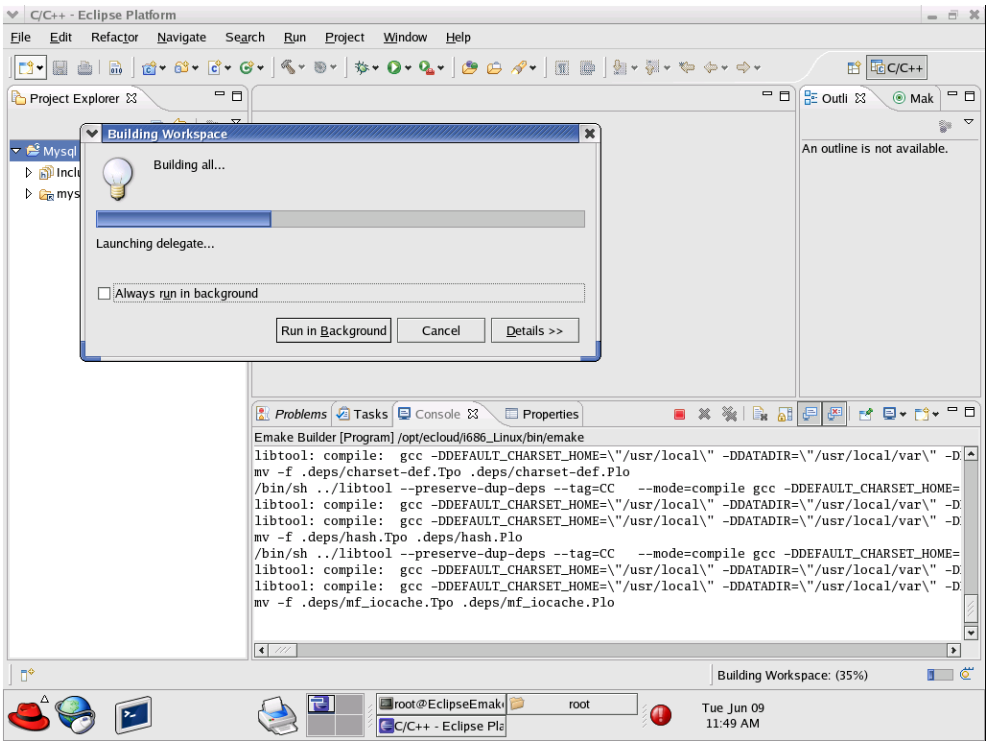
The following screenshot illustrates the Edit Configuration dialog.



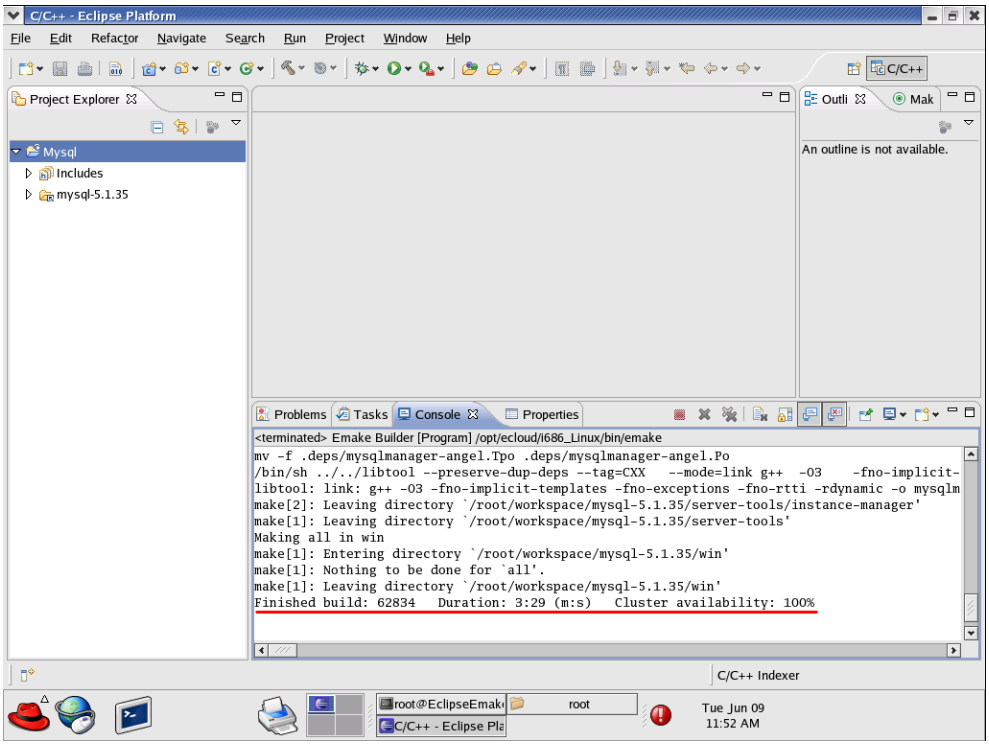
5. Click the Build Options tab. Enable Run the builder for the following *only*:
   - After a “Clean”
   - During manual builds
   - During auto builds
6. Click **OK**. Your new builder is displayed in the Builders pane.
7. Create another builder for “cleans” only. On its Main tab, ensure `clean` is included for Arguments. On its Build Options tab, enable Run the builder for the following *only*:
   - During a “Clean”
8. Click **OK**. Your second builder is displayed in the Builders pane.
9. Deselect CDT Builder in the Builders pane and then click **OK**.

Now you can build your project. Click **Project > Build all**.

The following screenshot illustrates a build in progress.



The following screenshot illustrates a successfully completed build.



Chapter 16: Electrify

Electrify accelerates builds by parallelizing the build process and distributing build steps across clustered resources.

Known Limitations

- The tool you want to monitor must provide parallel support, such as SCons `-j`, or MSBuild `/m` switch, and so on.
- Electrify does not provide any of Electric Make's dependency detection or correction features. The build tools you use with Electrify must be capable of accurate parallel execution on their own.
- The information written into annotation is more limited with Electrify than what is provided by Electric Make. Electrify annotation provides information only on the commands executed on the cluster, including command lines, file usage, and raw command output. Electrify does not provide information about dependencies, job relationships, targets, or other logical build structure data.

Recommendations

Electric Cloud has evaluated the following build tools for use with Electrify.

- SCons - Electric Cloud recommends using SCons with Electrify. Using SCons does not have any known limitations.
- MSBuild - Using MSbuild with Electrify is supported for Visual Basic and C# projects.
- Devenv - Using Devenv with Electrify is supported for C++ projects.
- Ant - Electric Cloud does not currently recommend using Ant with Electrify.

Electrify as Part of the Build Process

The following sections describe how to run builds using Electrify.

Important Reminders About MSBuild

- For MSBuild 3.5 (included in Visual Studio 2008) and MSBuild 4.0 (included in Visual Studio 2010):

  MSBuild 3.5 requires special configuration when distributing calls to the `vbc.exe` and `csc.exe` compilers. Under MSBuild 3.5, these tools use files written to the TMP directory. Ensure that the directory specified by the TMP environment variable is in the eMake root, or these tools will not operate correctly. If you wish to change the value of the TMP environment variable in order to locate it within the eMake root, you can do so by using the following command: `set "TMP=your path"`

Make sure that the entire 'TMP=' expression is enclosed in double-quotes if your TMP directory path contains any space characters.

- By default, MSBuild does not terminate its worker processes, but keeps them resident for the next build. This behavior can interfere with the correct operation of ElectricAccelerator. To avoid this, you must specify `/nr:false` so that msbuild.exe child processes exit when the build finishes.
- Information regarding use of the `/maxcpucount:[number]` switch (where *number* is the count of worker threads):
  - You can use `/m:[number]` as the short form.
  - Set `/m:[number]` to the number of agents that you want to use.
  - If you do not set `/m:[number]`, it is automatically set to the number of CPUs detected locally.

Important Reminders About Devenv

Devenv does not use the `/m` switch. To set the number of agents used in the build, do the following:

1. From Devenv, select Tools > Options > Project and Solutions > Build and Run
2. This displays a dialog box with the "maximum number of parallel project builds" option.

Note: Because the ElectricAccelerator Visual Studio Converter Add-in requires the "maximum number of parallel project builds" to be 1, you must use either Electrify ***or*** the add-in.

Running Electrify on Windows

Example

```
electrify [args] other tools' command line
```

Visual Studio C++ Projects Using MSBuild Example

```
electrify --emake-cm=<CM name> --electrify-remote=cl.exe;lib.exe;link.exe --
emake-root=<project root directory>;%temp% msbuild /m:4 /nr:false /t:build
<solution name>.sln
```

Visual Studio C++ Projects Using Devenv Example

```
electrify --emake-cm=<CM name> --electrify-remote=cl.exe;lib.exe;link.exe devenv
<solution name>.sln /build
```

Running Electrify on Linux

Example

```
electrify [args] other tools' command line
```

SCons Example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:ranlib:ar scons -j 4
```

GNU Make C++ Example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:<any other tools used
in the build> make -j 4 -f makefile
```

Important Reminders About Electrify

- Ensure `cl.exe`, `link.exe`, and so on, are those of Microsoft Visual Studio. The wrapper application may have changed them to its version.
- Though all of the usual eMake arguments are available, Electrify uses only a subset of them.
- When using Visual Studio 2010, building with `devenv` or `msbuild` with `electrifymon` errors out with a file/application not found error. To complete the build, use the appropriate workaround:
  - For `msbuild`, add `tracker.exe (--electrify-not-remote=msbuild.exe;tracker.exe)`
  - For `devenv`, add `msbuild.exe` and `tracker.exe (--electrify-not-remote=devenv.exe;devenv.com;msbuild.exe;tracker.exe)`
- On 64-bit Windows platforms, if you did not install ElectricAccelerator in its default install location, you must specify the complete location of `electrifymon.exe` (including the executable name) for the `EMAKE_ELECTRIFYMON` environment variable.
  For example, if your custom install location is `C:\programs\ECloud`, then set the `EMAKE_ELECTRIFYMON` environment variable using:
  `set EMAKE_ELECTRIFYMON = C:\programs\ECloud \i686_win32\64\bin\electrifymon.exe`.
- On UNIX platforms, electrifymon must locate `electrifymon.so` so it can tell monitored programs to load the monitoring library that reports back to electrifymon. By default, electrifymon looks in the following locations:

Platform	32-bit	64-bit
Linux	/opt/ecloud/i686_Linux/lib	/opt/ecloud/i686_Linux/64/lib
Solaris (SPARC)	/opt/ecloud/sun4u_SunOS/lib	/opt/ecloud/sun4u_SunOS/64/lib
Solaris (x86)	/opt/ecloud/i686_SunOS.5.10/lib	/opt/ecloud/i686_SunOS.5.10/64/lib

  You can override these locations using the following environment variables: `ELECTRIFYMON32DIR` and `ELECTRIFYMON64DIR`.
- On Linux, you can change the mode that Electrify uses for monitoring. Use `--emake-electrify=<mode>`, where mode can be: `preload` for LD_PRELOAD intercept, or `trace` for ptrace.

Electrify Arguments

All arguments are optional.

`--electrify-remote=<x;y>`	x and y are commands that are distributed to the cluster. Use the command's full name, such as `cl.exe`, `link.exe`, `gcc.exe`, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). Limited to 2048 characters Environment variable: `ELECTRIFY_REMOTE` `devenv` for C# projects is not supported.
`--electrify-not-remote=<x;y;>`	x and y are commands that ***are not*** distributed to the cluster. Use the command's full name, such as `cl.exe`, `link.exe`, `gcc.exe`, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). `--electrify-not-remote` and `--electrify-remote` are mutually exclusive. If you use `--electrify-not-remote`, all other tools' command lines are executed remotely, by default. Generally, this is not what you want, so to do this, you must add a command to this list. `devenv` for C# projects is not supported.
`--electrify-not-intercept=<x;y;>`	xxx and yyy are commands that you do ***not*** want to be monitored, meaning the monitor process does not inject a dll to them and their child processes, so they will ***not*** be distributed. Environment variable: `ELECTRIFY_NOT_INTERCEPT`
`--electrify-log=<fullpath>`	fullpath is the path of the file you want to log. This logs all process creation and interception information. Environment variable: `ELECTRIFY_LOG`

`--electrify-localfile=<x>`	(Windows only) Integrates local file access (create, rename, and so on) by locally running tools with the remote file system. You can set x to two different flags: `NT` or `y`. Set `nt` if you want to monitor undocumented low-level file access Nt functions. This monitors the following functions: NtCreateFile, NtDeleteFile, NtClose, NtWriteFile, and NtSetInformationFile. Though this includes only five functions, their functionality is rich, so this selection includes nearly all scenarios where the local file system changes. Set `y` to monitor documented Win32 APIs for file access. This monitors the following Win32 APIs: CreateFileW, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryW, DeleteFileA, DeleteFileW, MoveFileA, MoveFileExA, MoveFileExW, MoveFileW, RemoveDirectoryA, RemoveDirectoryW, SetFileAttributesA, and SetFileAttributesW. Though there are many functions, their functionality is less than Nt functions, particularly because some tools such as Cygwin cp.exe use NtCreateFile and so on. In general, use the `y` flag for testing purposes only.
`--electrify-allow-regexp=` `<perl-regular-expression>`	(Linux only) Sends all processes to the cluster whose full command-line matches the regular expression.
`--electrify-deny-regexp=` `<perl-regular-expression>`	(Linux only) The processes whose command-line match the expression will be executed locally. You can use this after the `--electrify-allow-regexp` option to tune the selection of processes that are sent to the cluster more precisely.

Using Whole Command-Line Matching and efpredict

Whole Command-Line Matching

On Linux, you can use a process's entire command-line to determine if it should be sent to the cluster for execution. Prior to Accelerator v7.0, the only way to discriminate was by the process name. Unfortunately, this did not work well for scripting languages or for languages that run in a VM, such as Java, because the process name is always 'java' no matter which particular program is being executed. This means that in previous versions you could send all Java programs to the cluster or none, and that was the limit of your discretion.

Additional Electrify command-line options:

`--electrify-allow-regexp=<perl-regular-expression>`

`--electrify-deny-regexp=<perl-regular-expression>`

These options allow you to specify which sub-processes should/should not be executed in the cluster. You use a perl-style regular expression that is matched against both the process name and all of its arguments, such as the name of the script or JAR file that is being executed. When Electrify detects that a process is started, it constructs the command-line for that process by joining all of the components of its "argv" array together with

spaces and then applying the list of "allow" and "deny" regular expressions in the sequence that they were supplied on the command-line.

Whole Command-Line Matching Example:

You have three processes:

```
java -jar runstep.jar -x86

java -jar otherjar.jar

java -jar runstep.jar -armv7
```

The following options send the first process to the cluster but not the second or third:

```
--electrify-regexp-allow="[^ ]+java\s.*runstep.jar.*"  --electrify-regexp-
deny=".*\-armv7.*"
```

Prior to Accelerator v7.0, you were only able to send all or none.

Initially, a process is considered to be for local execution only, but successive 'allow-regexp' options can change this state if any of them match. Any "deny-regexp" in the sequence will, if it matches, short-circuit the decision immediately and cause that process to be executed locally.

efpredict

efpredict helps you verify that the expressions you entered actually select the correct processes. Without efpredict, you would need to perform a full build and then examine the annotation file to see if the correct decisions were made. You would have to repeat this process each time there were any mistakes, and it could result in a long process. Instead, you can test settings with efpredict. Provide the same options as you would for Electrify and then enter command-lines into efpredict's standard input to see if it selects them for local or cluster execution. One easy way to do this is to pipe an old build log into efpredict. Check the resulting output visually to see if the desired processes were executed remotely.

efpredict Example:

```
cat oldlog | efpredict --electrify-regexp-allow="[^ ]+java\s.*runstep.jar.*"  --
electrify-regexp-deny=".*\-armv7.*"
```

gives this output:

```
remote_allow: java -jar runstep.jar -x86

remote_deny: java -jar otherjar.jar

remote_deny: java -jar runstep.jar -armv7
```

Important Notes

- Whole command-line matching is Linux only
- efpredict is Linux only
- If a process was executed by a shell, variables will be expanded, quotes will be removed, and white-space between tokens will be replaced with single spaces before Electrify matches the process. This means that if you look at a process invocation in a shell script or makefile, that may not be the exact text that Electrify sees when it attempts to intercept the invocation of that process.

  For example, in a script you might see:

  ```
  'gcc        "$SOURCE/myfile.c"       -o "$OUTPUT/myfile.o" -c    '
  ```

  but when Electrify intercepts this and tries to reconstruct the command-line, it will see:

  ```
  "gcc src/myfile.c -o out/myfile.o -c".
  ```

Regular expressions must be written to match what Electrify will be able to see.

- One regexp can match many commands. For example, to send both the gcc and ld commands to the cluster you can use:

```
--electrify-regexp-allow='[^ ]*((gcc)|(ld))(\s.*)?'
```

Additional Electrify Information

Selecting Commands to Parallelize

A large portion of the build process acceleration will be achieved through parallelizing a small number of specific commands, such as compiling and linking. Expending additional effort to select and parallelize many different additional commands may not result in a significant amount of further acceleration.

Note: If a file is created or modified by one or more parallelized commands, then you should parallelize all commands that use that file.

Using Electrify with GNU Make

If you intend to use Electrify with GNU Make, Electric Cloud recommends using eMake instead. eMake provides superior performance and correctness and full annotation information.

How an Electrify Build Differs from an eMake Build

An important difference between an eMake build and an Electrify build is what portion of the build activity occurs remotely versus locally. In an eMake build, effectively all build activity (except “#pragma runlocal” jobs) takes place on the cluster, where the EFS is used to monitor filesystem accesses and propagate changes made by one job to other jobs in the build.

In an Electrify build, a greater portion of the build activity takes place on the local system—at the very least, the build process itself (such as SCons) runs locally. Typically, filesystem modifications made by processes running locally are “invisible” to Electrify, and therefore to processes running on the cluster—just as “#pragma runlocal” jobs may make changes that are invisible to eMake. Electrifymon provides a means to update the virtual filesystem state in Electrify in response to filesystem modifications made by local processes.

If you know that a build will not run any processes locally that modify filesystem, you need not use electrifymon when invoking Electrify. However, some build tools will themselves make changes to the filesystem (for example, when SCons employs its build-avoidance mechanism by copying a previously built object in lieu of invoking the compiler), so the safest choice is to use electrifymon to start.

In addition to filesystem monitoring, electrifymon on Windows provides a sophisticated mechanism for intercepting processes invocations and determining which processes to distribute to the cluster. On Linux, process interception is handled by the explicit use of proxy commands.

Chapter 17: Troubleshooting

The following topics discuss information to assist you with troubleshooting.

Topics:

Agent Errors Establishing the Virtual Filesystem

Agent errors regarding the establishment of the virtual filesystem for a particular build will be displayed if there are at least three errors. These errors would occur during the initial setup of the agent's build-specific environment but before any particular build step is run on that agent. The most common type of error involves eMake roots or Cygwin mounts, where virtual filesystem setup is specific to the build but not to any particular build step.

Agents Do Not Recognize Changes on Agent Machines

If you manually mount a filesystem or change automounter settings on agent machines *after* they are started, you must restart the Agents for them to recognize your changes.

Electric Make Debug Log Levels

This section discusses Electric Make debug log levels. Content was adapted from the “Electric Make debug log levels” blog post on http://blog.melski.net/ and was the most recent information available at the time of the article's posting.

Disclaimer: eMake debug logs are intended for use by Electric Cloud engineering and support staff. Debug logging contents and availability are subject to change in any release, for any or no reason.

Often when analyzing builds executed with Electric Make, all of the information you need is in the annotation file—an easily digested XML file containing data such as the relationships between the jobs, the commands run, and the timing of each job. But sometimes you need more detail, and that is where the eMake debug log is useful.

Enabling eMake Debug Logging

To enable eMake debug logging, specify this pair of command-line arguments:

`--emake-debug=<value>` specifies the types of debug logging to enable. Provide a set of single-letter values, such as “`jng`”.

`--emake-logfile=<path>` specifies the location of the debug log.

eMake Debug Log Level Descriptions

Available log levels:

a: agent allocation	l: ledger
c: cache	m: memory
e: environment	n: node
f: filesystem	o: parse output
g: profiling	p: parse
h: history	P: parse avoidance
j: job	r: parse relocation
L: nmake lexer	s: subbuild
	Y: security

a: agent allocation

Agent allocation logging provides detailed information about eMake's attempts to procure agents from the Cluster Manager during the build. If you think eMake may be stalled trying to acquire agents, allocation logging will help to understand what is happening.

c: cache

Cache logging records details about the filesystem cache used by eMake to accelerate parse jobs in cluster builds. For example, it logs when a directory's contents are added to the cache, and the result of lookups in the cache. Because it is only used during remote parse jobs, you must use it with the `--emake-rdebug=value` option. Use cache logging if you suspect a problem with the *cached local filesystem*.

e: environment

Environment logging augments node logging with a dump of the entire environment block for every job as it is sent to an agent. Normally this is omitted because it is quite verbose (could be as much as 32 KB per job). Generally, it is better to use env-level annotation, which is more compact and easier to parse.

f: filesystem

Filesystem logging records numerous details about eMake's interaction with its versioned filesystem data structure. In particular, it logs every time that eMake looks up a file (when doing up-to-date checks, for example), and it logs every update to the versioned file system caused by file usage during the build's execution. This level of logging is very verbose, so it is not usually enabled. It is most often used when diagnosing issues related to the versioned filesystem and conflicts.

g: profiling

Profiling logging is one of the easiest to interpret and most useful types of debug logging. When enabled, eMake will emit hundreds of performance metrics at the end of the build. This is a very lightweight logging level, and is safe (even advisable) to enable for all builds.

h: history

History logging prints messages related to the data tracked in the eMake history file—both filesystem dependencies and autodep information. When history logging is enabled, eMake will print a message every time a dependency is added to the history file, and it will print information about the files checked during up-to-date checks based on autodep data. Enable history logging if you suspect a problem with autodep behavior.

j: job

Job logging prints minimal messages related to the creation and execution of jobs. For each job you will see a message when it starts running, when it finishes running, and when eMake checks the job for conflicts. If there is a conflict in the job, you will see a message about that, too. If you just want a general overview of how the build is progressing, j-level logging is a good choice.

L: nmake lexer

eMake uses a generated parser to process portions of NMAKE makefiles. Lexer debug logging enables the debug logging in that generated code. This is generally not useful to end-users because it is too low-level.

l: ledger

Ledger debug logging prints information about build decisions based on data in the ledger file, as well as updates made to the ledger file. Enable it if you believe the ledger is not functioning correctly.

m: memory

When memory logging is enabled, eMake prints memory usage metrics to the debug log once per second. This includes the total process memory usage as well as current and peak memory usage grouped into several “buckets” that correspond to various types of data in eMake. For example, the “Operation” bucket indicates the amount of memory used to store file operations; the “Variable” bucket is the amount of memory used for makefile variables. This is most useful when you are experiencing an out-of-memory failure in eMake because it can provide guidance about how memory is being utilized during the build, and how quickly it is growing.

n:node

Node logging prints detailed information about all messages between eMake and the agents, including filesystem data and commands executed. Together with job logging, this can give a very comprehensive picture of the behavior of a build. However, node logging is extremely verbose, so enable it only when you are chasing a specific problem.

o: parse output

Parse output logging instructs eMake to preserve the raw result of parsing a makefile. The result is a binary file containing information about all targets, rules, dependencies, and variables extracted from makefiles read during a parse job. This can be useful when investigating parser incompatibility issues and scheduling issues (for example, if a rule is not being scheduled for execution when you expect). Note that this debug level only makes sense when parsing, which means you must specify it in the `--emake-rdebug` option. The parse results will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.out`. Note that the directory may be on the local disk of the remote nodes, depending on the value you specify.

p: parse

Parse debug logging prints extremely detailed information about the reading and interpretation of makefiles during a parse job. This is most useful when investigating parser compatibility issues. This output is very verbose, so enable it only when you are pursuing a specific problem. Like parse output logging, this debug level only makes sense during parsing, which means you must specify it in the `--emake-rdebug` option. The parse log files will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.dlog`. Note that the directory may be on the local disk of the remote nodes, depending on the value you specify.

P: parse avoidance

Parse avoidance logging indicates when a new parse is required, and if so, why it was required.

r: parse relocation

Parse relocation logging prints low-level information about the process of transmitting parse result data to eMake at the end of a parse job. It is used only internally when we the parse result format is being extended, so is unlikely to be of interest to end-users.

s: subbuild

Subbuild logging prints details about decisions made while using the eMake subbuild feature. Enable it if you believe that the subbuild feature is not working correctly.

Y: authentication

Authentication logging is a subset of node logging that prints only those messages related to authenticating eMake to agents and vice-versa. Enable this debug level, if you are having problems using the authentication feature.

Index