Red Hat Enterprise Linux 7 Developer Guide

An introduction to application development tools in Red Hat Enterprise Linux 7

Jacquelynn East Don Domingo Robert Krátký

Red Hat Enterprise Linux 7 Developer Guide

An introduction to application development tools in Red Hat Enterprise Linux 7

Jacquelynn East
Red Hat Customer Content Services
jeast@redhat.com

Don Domingo
Red Hat Customer Content Services
ddomingo@redhat.com

Robert Krátký
Red Hat Customer Content Services
rkratky@redhat.com

Legal Notice

Abstract

This document describes the different features and utilities that make Red Hat Enterprise Linux 7 an ideal enterprise platform for application development.

Table of Contents

Chapter 1. Collaborating

Effective revision control is essential to all multi-developer projects. It allows all developers in a team to create, review, revise, and document code in a systematic and orderly manner.
Red Hat_Enterprise Linux 7 is distributed with three of the most popular open source revision control systems: **CVS**, **SVN**, and **Git**.

This chapter provides information on how to install and use these tools, as well as links to relevant external documentation.

1.1. Concurrent Versions System (CVS)

Concurrent Versions System, commonly abbreviated as **CVS**, is a *centralized version control system* with a client-server architecture. It is a successor to the older **Revision Control System** (**RCS**). **CVS** allows multiple developers to cooperate on the same project while keeping track of every change made to the files that are under revision control.

1.1.1. Installing and Configuring CVS

Installing the cvs Package

In Red Hat_Enterprise Linux 7, **CVS** is provided by the *cvs* package. To install the *cvs* package and all its dependencies on your system, type the following at a shell prompt as **root**:

```
yum install cvs
```

This installs a command line **CVS** client, a **CVS** server, and other related tools to the system.

Setting Up the Default Editor

When using **CVS** on the command line, certain commands, such as **cvs import** or **cvs commit**, require the user to write a short log message. To determine which text editor to start, the **cvs** client application first reads the contents of the environment variable **$CVSEDITOR**, then reads the more general environment variable **$EDITOR**, and if none of these is set, it starts **vi**.

To persistently change the value of the **$CVSEDITOR** environment variable, run the following command:

```
echo "export CVSEDITOR=command" >> ~/.bashrc
```

This adds the **export CVSEDITOR=*command*** line to your **~/.bashrc** file. Replace *command* with a command that runs the editor of your choice (for example, **emacs**). Note that for this change to take effect in the current shell session, you must execute the commands in **~/.bashrc** by typing the following at a shell prompt:

```
source ~/.bashrc
```

> **Example 1.1. Setting up the default text editor**
>
> To configure the **CVS** client to use **Emacs** as a text editor, type:

```
~]$ echo "export CVSEDITOR=emacs" >> ~/.bashrc
~]$ source ~/.bashrc
```

1.1.2. Creating a New Repository

A CVS *repository* is a central place to store files and directories that are under revision control, as well as additional data, such as a complete history of changes or information about who made those changes and when. A typical CVS repository stores multiple projects in separate subdirectories called *modules*. When publicly accessible, it allows several developers to create a *working copy* of any of the modules, make changes, and share these changes with others by *committing* them back to the repository.

Initializing an Empty Repository

To create a new, empty CVS repository in a directory of your choice, run the following command:

```
cvs -d path init
```

Note that *path* must be an absolute path to the directory in which you want to store the repository (for example, **/var/cvs/**). Alternatively, you can specify this path by changing the value of the **$CVSROOT** environment variable:

```
export CVSROOT=path
```

This allows you to omit the path from **cvs init** and other CVS-related commands:

```
cvs init
```

Example 1.2. Initializing a new CVS repository

To create an empty CVS repository in the **~/cvs/** directory, type:

```
~]$ export CVSROOT=~/cvs
~]$ cvs init
```

Importing Data to a Repository

To put an existing project under revision control, change to the directory in which the project is stored and run the following command:

```
cvs [-d cvs_repository] import [-m "commit message"] module vendor_tag
release_tag
```

Note that *cvs_repository* is a path to the CVS repository, *module* is the subdirectory into which you want to import the project (such as **project**), and *vendor_tag* and *release_tag* are vendor and release tags.

Example 1.3. Importing a project to a CVS repository

Let us assume that the directory with your project has the following contents:

```
~]$ ls myproject
AUTHORS  doc  INSTALL  LICENSE  Makefile  README  src  TODO
```

Let us further assume that you have an empty CVS repository in the **~/cvs/** directory. To import the project under the **project** directory in this repository with the **mycompany** vendor tag and the **init** release tag, type:

```
myproject]$ export CVSROOT=~/cvs
myproject]$ cvs import -m "Initial import." project mycompany init
N project/Makefile
N project/AUTHORS
N project/LICENSE
N project/TODO
N project/INSTALL
...
```

1.1.3. Checking Out a Working Copy

To check out a working copy of a project in a CVS repository, run the following command:

```
cvs -d cvs_repository checkout module
```

This creates a new directory called *module* with a working copy of a project in it. Note that *cvs_repository* is the URL of the CVS repository and *module* is the subdirectory in which the project is stored (such as **project**). Alternatively, you can set the **$CVSROOT** environment variable as follows:

```
export CVSROOT=cvs_repository
```

Then you can use the **cvs checkout** command without the **-d** option:

```
cvs checkout module
```

Example 1.4. Checking out a working copy

Let us assume that you have a CVS repository in the **~/cvs/** directory and that this repository contains a module named **project**. To check out a working copy of this module, type:

```
~]$ export CVSROOT=~/cvs
~]$ cvs checkout project
cvs checkout: Updating project
U project/AUTHORS
U project/INSTALL
U project/LICENSE
U project/Makefile
U project/TODO
```

1.1.4. Adding and Deleting Files

Adding a File

To add an existing file to a CVS repository and put it under revision control, change to the directory with the working copy of the file and run the following command:

```
cvs add file
```

This schedules the file for addition to the CVS repository. To proceed and actually add the file to the repository, run the **cvs commit** command as described in Section 1.1.6, “Committing Changes”.

Example 1.5. Adding a file to a CVS repository

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  CVS  doc  INSTALL  LICENSE  Makefile  README  src
TODO
```

With the exception of **ChangeLog**, all files and directories within this directory are already under revision control. To schedule this file for addition to the CVS repository, type:

```
project]$ cvs add ChangeLog
cvs add: scheduling file `ChangeLog' for addition
cvs add: use 'cvs commit' to add this file permanently
```

Deleting a File

To remove a file from a CVS repository, change to the directory with the working copy of the file and delete it locally:

```
rm file
```

Then schedule this file for removal by using the following command:

```
cvs remove file
```

To proceed and actually remove the file from the repository, run the **cvs commit** command as described in Section 1.1.6, “Committing Changes”.

Example 1.6. Removing a file from a CVS repository

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  CVS  doc  INSTALL  LICENSE  Makefile  README  src
TODO
```

All files in this directory are under revision control. To schedule the **TODO** file for removal from the CVS repository, type:

```
project]$ rm TODO
project]$ cvs remove TODO
cvs remove: scheduling `TODO' for removal
cvs remove: use 'cvs commit' to remove this file permanently
```

1.1.5. Viewing Changes

Viewing the Status

To determine the current status of a working copy, change to the directory with the working copy and run the following command:

```
cvs status
```

This displays detailed information about each file that is under revision control, including its current status (such as **Up-to-date**, **Locally Added**, **Locally Removed**, or **Locally Modified**) and revision. To display only changes in your working copy, simplify the output by typing the following at a shell prompt:

```
cvs status 2>/dev/null | grep Status: | grep -v Up-to-date
```

Example 1.7. Viewing the status of a working copy

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  CVS  doc  INSTALL  LICENSE  Makefile  README  src
```

With the exception of **ChangeLog**, which is scheduled for addition to the CVS repository, all files and directories within this directory are already under revision control. The **TODO** file, which is also under revision control, has been scheduled for removal and is no longer present in the working copy. Finally, **Makefile** contains local changes. To display the status of such a working copy, type:

```
project]$ cvs status 2>/dev/null | grep Status: | grep -v Up-to-date
File: ChangeLog         Status: Locally Added
File: Makefile          Status: Locally Modified
File: no file TODO              Status: Locally Removed
```

Viewing Differences

To view differences between a working copy and the checked-out content, change to the directory with the working copy and run the following command:

```
cvs diff [file]
```

This displays changes to all files in the working copy. To display only changes to a particular file, supply the file name on the command line.

Example 1.8. Viewing changes to a working copy

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  CVS  doc  INSTALL  LICENSE  Makefile  README  src
```

All files in this directory are under revision control, and **Makefile** contains local changes. To view these changes, type:

```
project]$ cvs diff
cvs diff: Diffing .
cvs diff: ChangeLog is a new entry, no comparison available
Index: Makefile
===================================================================
RCS file: /home/john/cvs/project/Makefile,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 Makefile
156c156
<       -rm -f $(MAN1)
---
>       -rm -f $(MAN1) $(MAN7)
cvs diff: TODO was removed, no comparison available
cvs diff: Diffing doc
...
```

1.1.6. Committing Changes

To share your changes with others and commit them to a CVS repository, change to the directory with the working copy of your repository and run the following command:

```
cvs commit [-m "commit message"]
```

Note that unless you specify the commit message on the command line, CVS opens an external text editor (**vi** by default) for you to write it. For information on how to configure which editor to start, see Section 1.1.1, “Installing and Configuring CVS”.

Example 1.9. Committing changes to a CVS repository

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  CVS  doc  INSTALL  LICENSE  Makefile  README  src
```

In this working copy, **ChangeLog** is scheduled for addition to the CVS repository, **Makefile** is already under revision control and contains local changes, and the **TODO** file, which is also under revision control, has been scheduled for removal and is no longer present in the working copy. To commit these changes to the CVS repository, type:

```
project]$ cvs commit -m "Updated the makefile."
```

```
cvs commit: Examining .
cvs commit: Examining doc
...
RCS file: /home/john/cvsroot/project/ChangeLog,v
done
Checking in ChangeLog;
/home/john/cvsroot/project/ChangeLog,v  <--  ChangeLog
initial revision: 1.1
done
Checking in Makefile;
/home/john/cvsroot/project/Makefile,v  <--  Makefile
new revision: 1.2; previous revision: 1.1
done
Removing TODO;
/home/john/cvsroot/project/TODO,v  <--  TODO
new revision: delete; previous revision: 1.1.1.1
done
```

1.1.7. Updating a Working Copy

To update a working copy and get the latest changes from a CVS repository, change to the directory with the working copy of your repository and run the following command:

```
cvs update
```

Example 1.10. Updating a working copy

Let us assume that the directory with your working copy of a CVS repository has the following contents:

```
project]$ ls
AUTHORS  CVS  doc  INSTALL  LICENSE  Makefile  README  src TODO
```

Let us further assume that another user has recently added **`ChangeLog`** to the repository, removed **`TODO`**, and made some changes to **`Makefile`**. To update this working copy, type:

```
myproject]$ cvs update
cvs update: Updating .
U ChangeLog
U Makefile
cvs update: TODO is no longer in the repository
cvs update: Updating doc
cvs update: Updating src
```

1.1.8. Additional Resources

A detailed description of all supported features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- cvs(1) — The manual page for the **cvs** client program provides detailed information on its usage.

1.2. Apache Subversion (SVN)

Apache Subversion, commonly abbreviated as **SVN**, is a *centralized version control system* with a client-server architecture. It is a successor to the older **Concurrent Versions System** (**CVS**), preserves the same development model, and addresses problems often encountered with **CVS**.

1.2.1. Installing and Configuring Subversion

Installing the subversion Package

In Red Hat_Enterprise Linux 7, **Subversion** is provided by the *subversion* package. To install the *subversion* package and all its dependencies on your system, type the following at a shell prompt as **root**:

```
yum install subversion
```

This installs a command line **Subversion** client, a **Subversion** server, and other related tools to the system.

Setting Up the Default Editor

When using **Subversion** on the command line, certain commands, such as **svn import** or **svn ommit**, require the user to write a short log message. To determine which text editor to start, the **svn** client application first reads the contents of the environment variable **$SVN_EDITOR**, then reads more general environment variables **$VISUAL** and **$EDITOR**, and if none of these is set, it reports an error.

To persistently change the value of the **$SVN_EDITOR** environment variable, run the following command:

```
echo "export SVN_EDITOR=command" >> ~/.bashrc
```

This adds the **export SVN_EDITOR=*command*** line to your **~/.bashrc** file. Replace *command* with a command that runs the editor of your choice (for example, **emacs**). Note that for this change to take effect in the current shell session, you must execute the commands in **~/.bashrc** by typing the following at a shell prompt:

```
source ~/.bashrc
```

Example 1.11. Setting up the default text editor

To configure the **Subversion** client to use **Emacs** as a text editor, type:

```
~]$ echo "export SVN_EDITOR=emacs" >> ~/.bashrc
~]$ . ~/.bashrc
```

1.2.2. Creating a New Repository

A Subversion *repository* is a central place to store files and directories that are under revision control, as well as additional data, such as a complete history of changes or information about who made those changes and when. A typical Subversion repository stores multiple projects in separate subdirectories. When publicly accessible, it allows several developers to create a *working copy* of any of the subdirectories, make changes, and share these changes with others by *committing* them back to the repository.

Initializing an Empty Repository

To create a new, empty Subversion repository in a directory of your choice, run the following command:

```
svnadmin create path
```

Note that *path* is an absolute or relative path to the directory in which you want to store the repository (for example, **/var/svn/**). If the directory does not exist, **svnadmin create** creates it for you.

Example 1.12. Initializing a new Subversion repository

To create an empty Subversion repository in the **~/svn/** directory, type:

```
~]$ svnadmin create svn
```

Importing Data to a Repository

To put an existing project under revision control, run the following command:

```
svn import local_path svn_repository/remote_path [-m "commit message"]
```

Note that *local_path* is an absolute or relative path to the directory in which you keep the project (use **.** for the current working directory), *svn_repository* is the URL of the Subversion repository, and *remote_path* is the target directory in the Subversion repository (for example, **project/trunk**).

Example 1.13. Importing a project to a Subversion repository

Let us assume that the directory with your project has the following contents:

```
~]$ ls myproject
AUTHORS  doc  INSTALL  LICENSE  Makefile  README  src  TODO
```

Let us further assume that you have an empty Subversion repository in the **~/svn/** directory (in this example, **/home/john/svn/**). To import the project under **project/trunk** into this repository, type:

```
~]$ svn import myproject file:///home/john/svn/project/trunk -m
"Initial import."
Adding         project/AUTHORS
Adding         project/doc
```

```
Adding         project/doc/index.html
Adding         project/INSTALL
Adding         project/src
...
```

1.2.3. Checking Out a Working Copy

To check out a working copy of a project in a Subversion repository, run the following command:

```
svn checkout svn_repository/remote_path [directory]
```

This creates a new directory called *directory* with a working copy of the project in it. Note that *svn_repository* is the URL of the Subversion repository, and *remote_path* is the subdirectory in which the project is stored.

Example 1.14. Checking out a working copy

Let us assume that you have a Subversion repository in the **~/svn/** directory (in this case, **/home/john/svn/**) and that this repository contains the latest version of the project in the **project/trunk** subdirectory. To check out a working copy of this project, type:

```
~]$ svn checkout svn:///home/john/svn/project/trunk project
A    project/AUTHORS
A    project/doc
A    project/doc/index.html
A    project/INSTALL
A    project/src
...
```

1.2.4. Adding, Renaming, and Deleting Files

Adding a File or Directory

To add an existing file to a Subversion repository and put it under revision control, change to the directory with a working copy of the file and run the following command:

```
svn add file
```

Similarly, to add a directory and all files that are in it, type:

```
svn add directory
```

This schedules the files and directories for addition to the Subversion repository. To proceed and actually add this content to the repository, run the **svn commit** command as described in Section 1.2.6, “Committing Changes”.

Example 1.15. Adding a file to a Subversion repository

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  doc  INSTALL  LICENSE  Makefile  README  src  TODO
```

With the exception of **ChangeLog**, all files and directories within this directory are already under revision control. To schedule this file for addition to the Subversion repository, type:

```
project]$ svn add ChangeLog
A         ChangeLog
```

Renaming a File or Directory

To rename an existing file or directory in a Subversion repository, change to the directory with a working copy of the file or the directory and run the following command:

```
svn move old_name new_name
```

This creates a duplicate of the original file or directory, schedules it for addition, and automatically deletes the original. To proceed and actually rename the content in the Subversion repository, run the **svn commit** command as described in Section 1.2.6, “Committing Changes”.

Example 1.16. Renaming a file in a Subversion repository

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  doc  INSTALL  LICENSE  Makefile  README  src  TODO
```

All files in this directory are under revision control. To schedule the **LICENSE** file for renaming to **COPYING**, type:

```
project]$ svn move LICENSE COPYING
A         COPYING
D         LICENSE
```

Note that **svn move** automatically renames the file in your working copy:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  doc  INSTALL  Makefile  README  src TODO
```

Deleting a File or Directory

To remove a file from a Subversion repository, change to the directory with a working copy of the file and run the following command:

```
svn delete file…
```

Similarly, to remove a directory and all files that are in it, type:

```
svn delete directory…
```

This schedules the files and directories for removal from the Subversion repository. To proceed and actually remove this content from the repository, run the **svn commit** command as described in Section 1.2.6, “Committing Changes”.

Example 1.17. Deleting a file from a Subversion repository

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  doc  INSTALL  Makefile  README  src TODO
```

All files in this directory are under revision control. To schedule the **TODO** file for removal from the Subversion repository, type:

```
project]$ svn delete TODO
D         TODO
```

Note that **svn delete** automatically deletes the file from your working copy:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  doc  INSTALL  Makefile  README  src
```

1.2.5. Viewing Changes

Viewing the Status

To determine the current status of a working copy, change to the directory with the working copy and run the following command:

```
svn status
```

This displays information about all changes to the working copy. See Table 1.1, “Subversion Status Symbols” for an explanation of the symbols used in the output of the **svn status** command.

Table 1.1. Subversion Status Symbols

Symbol	Meaning
A	File is scheduled for addition.
D	File is scheduled for removal.
M	File contains local changes.
C	File with contains unresolved conflicts.
?	File is not under revision control.

Example 1.18. Viewing the status of a working copy

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  doc  INSTALL  Makefile  README  src
```

With the exception of **ChangeLog**, which is scheduled for addition to the Subversion repository, all files and directories within this directory are already under revision control. The **TODO** file, which is also under revision control, has been scheduled for removal and is no longer present in the working copy. The **LICENSE** file has been renamed to **COPYING**, and **Makefile** contains local changes. To display the status of such a working copy, type:

```
project]$ svn status
D       LICENSE
D       TODO
A       ChangeLog
A  +    COPYING
M       Makefile
```

Viewing Differences

To view differences between a working copy and the checked-out content, change to the directory with the working copy and run the following command:

```
svn diff [file]
```

This displays changes to all files in the working copy. To display only changes to a particular file, supply the file name on the command line.

Example 1.19. Viewing changes to a working copy

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  CVS  doc  INSTALL  Makefile  README  src
```

All files in this directory are under revision control and **Makefile** contains local changes. To view these changes, type:

```
project]$ svn diff Makefile
Index: Makefile
===================================================================
--- Makefile    (revision 1)
+++ Makefile    (working copy)
@@ -153,7 +153,7 @@
      -rmdir $(man1dir)

clean:
-       -rm -f $(MAN1)
+       -rm -f $(MAN1) $(MAN7)

%.1: %.pl
      $(POD2MAN) --section=1 --release="Version $(VERSION)" \
```

1.2.6. Committing Changes

To share your changes with others and commit them to a Subversion repository, change to the directory with a working copy of the changes and run the following command:

```
svn commit [-m "commit message"]
```

Note that unless you specify the commit message on the command line, Subversion opens an external text editor for you to write it. For information on how to configure which editor to start, see Section 1.2.1, “Installing and Configuring Subversion”.

Example 1.20. Committing changes to a Subversion repository

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  ChangeLog  COPYING  doc  INSTALL  Makefile  README  src
```

In this working copy, **ChangeLog** is scheduled for addition to the Subversion repository, **Makefile** already is under revision control and contains local changes, and **TODO**, which is also under revision control, has been scheduled for removal and is no longer present in the working copy. Additionally, the **LICENSE** file has been renamed to **COPYING**. To commit these changes to the Subversion repository, type:

```
project]$ svn commit -m "Updated the makefile."
Adding         COPYING
Adding         ChangeLog
Deleting       LICENSE
Sending        Makefile
Deleting       TODO
Transmitting file data ..
Committed revision 2.
```

1.2.7. Updating a Working Copy

To update a working copy and get the latest changes from a Subversion repository, change to the directory with the working copy and run the following command:

```
svn update
```

Example 1.21. Updating a working copy

Let us assume that the directory with your working copy of a Subversion repository has the following contents:

```
project]$ ls
AUTHORS  doc  INSTALL  LICENSE  Makefile  README  src TODO
```

Let us further assume that somebody recently added **ChangeLog** to the repository, removed the **TODO** file from it, changed the name of **LICENSE** to **COPYING**, and made some changes to **Makefile**. To update this working copy, type:

```
myproject]$ svn update
D    LICENSE
D    TODO
A    COPYING
A    Changelog
M    Makefile
Updated to revision 2.
```

1.2.8. Additional Resources

A detailed description of all supported features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- **svn help** — The output of the **svn help** command provides detailed information about the use of **svn**.
- **svnadmin help** — The output of the **svnadmin help** command provides detailed information about the use of **svnadmin**.

Online Documentation

- Version Control with Subversion — The official Subversion website refers to the *Version Control with Subversion* manual, which provides an in-depth description of Subversion, its administration, and its usage.

1.3. Git

Git is a *distributed revision control system* with a peer-to-peer architecture. As opposed to centralized version control systems with a client-server model, **Git** ensures that each working copy of a **Git** repository is its exact copy with complete revision history. This not only allows you to work on and contribute to projects without the need to have permission to push your changes to their official repositories, but also makes it possible for you to work with no network connection.

1.3.1. Installing and Configuring Git

Installing the git Package

In Red Hat_Enterprise Linux 7, **Git** is provided by the *git* package. To install the *git* package and all its dependencies on your system, type the following at a shell prompt as **root**:

```
~]# yum install git
```

Configuring the Default Text Editor

Certain **Git** commands, such as **git commit**, require the user to write a short message or make some changes in an external text editor. To determine which text editor to start, **Git** attempts to read

the value of the **GIT_EDITOR** environment variable, the **core.editor** configuration option, the **VISUAL** environment variable, and finally the **EDITOR** environment variable in this particular order. If none of these options and variables are specified, the **git** command starts **vi** as a reasonable default option.

To change the value of the **core.editor** configuration option in order to specify a different text editor, type the following at a shell prompt:

```
git config --global core.editor command
```

Replace *command* with the command to be used to start the selected text editor.

Example 1.22. Configuring the Default Text Editor

To configure **Git** to use **vim** as the default text editor, type the following at a shell prompt:

```
~]$ git config --global core.editor vim
```

Setting Up User Information

In **Git**, each commit (or revision) is associated with the full name and email of the person responsible for it. By default, **Git** uses an identity based on the user name and the host name.

To change the full name associated with your **Git** commits, type the following at a shell prompt:

```
git config --global user.name "full name"
```

To change the email address associated with your **Git** commits, type:

```
git config --global user.email "email_address"
```

Example 1.23. Setting Up User Information

To configure **Git** to use **John Doe** as your full name and **john@example.com** as your email address, type the following at a shell prompt:

```
~]$ git config --global user.name "John Doe"
~]$ git config --global user.email "john@example.com"
```

1.3.2. Creating a New Repository

A *repository* is a place where **Git** stores all files that are under revision control, as well as additional data related to these files, such as the complete history of changes or information about who made those changes and when. Unlike in centralized revision control systems like Subversion or CVS, a **Git** repository and a *working directory* are usually the same. A typical **Git** repository also only stores a single project and when publicly accessible, it allows anyone to create its clone with a complete revision history.

Initializing an Empty Repository

To create a new, empty **Git** repository, change to the directory in which you want to keep the repository and type the following at a shell prompt:

```
git init
```

This creates a hidden directory named `.git` in which all repository information is stored.

Importing Data to a Repository

To put an existing project under revision control, create a **Git** repository in the directory with the project and run the following command:

```
git add .
```

This marks all files and directories in the current working directory as ready to be added to the **Git** repository. To proceed and actually add this content to the repository, commit the changes by typing the following at a shell prompt:

```
git commit [-m "commit message"]
```

Replace *commit message* with a short description of your revision. Omit the `-m` option to write the commit message in an external text editor. For information on how to configure the default text editor, see Section 1.3.1, “Configuring the Default Text Editor”.

1.3.3. Cloning an Existing Repository

To clone an existing **Git** repository, type the following at a shell prompt:

```
git clone git_repository [directory]
```

Replace *git_repository* with a URL or a path to the **Git** repository you want to clone, and *directory* with a path to the directory in which you want to store the clone.

1.3.4. Adding, Renaming, and Deleting Files

Adding Files and Directories

To add an existing file to a **Git** repository and put it under revision control, change to the directory with your local **Git** repository and type the following at a shell prompt:

```
git add file
```

Replace *file* with the file or files you want to add. This command marks the selected file or files as ready to be added to the **Git** repository. Similarly, to add all files that are stored in a certain directory to a **Git** repository, type:

```
git add directory
```

Replace *directory* with the directory or directories you want to add. This command marks all files in the selected directory or directories as ready to be added to the **Git** repository.

To proceed and actually add this content to the repository, commit the changes as described in Section 1.3.6, “Committing Changes”.

Renaming Files and Directories

To rename an existing file or directory in a **Git** repository, change to the directory with your local **Git** repository and type the following at a shell prompt:

```
git mv old_name new_name
```

Replace *old_name* with the current name of the file or directory and *new_name* with the new name. This command renames the selected file or directory and marks it as ready to be renamed in the **Git** repository.

To proceed and actually rename the content in the repository, commit the changes as described in Section 1.3.6, “Committing Changes”.

Deleting Files and Directories

To delete an existing file from a **Git** repository, change to the directory with your local **Git** repository and type the following at a shell prompt:

```
git rm file
```

Replace *file* with the file or files you want to delete. This command deletes all selected files and marks them as ready to be deleted form the **Git** repository. Similarly, to delete all files that are stored in a certain directory from a **Git** repository, type:

```
git rm -r directory
```

Replace *directory* with the directory or directories you want to delete. This command deletes all selected directories and marks them as ready to be deleted from the **Git** repository.

To proceed and actually delete this content from the repository, commit the changes as described in Section 1.3.6, “Committing Changes”.

1.3.5. Viewing Changes

Viewing the Current Status

To determine the current status of your local **Git** repository, change to the directory with the repository and type the following command at a shell prompt:

```
git status
```

This command displays information about all uncommitted changes in the repository (**new file**, **renamed**, **deleted**, or **modified**) and tells you which changes will be applied the next time you commit them. For information on how to commit your changes, see Section 1.3.6, “Committing Changes”.

Viewing Differences

To view all changes in a **Git** repository, change to the directory with the repository and type the following at a shell prompt:

```
git diff
```

This command displays changes between the files in the repository and their latest revision. If you are only interested in changes in a particular file, supply its name on the command line as follows:

```
git diff file...
```

Replace *file* with the file or files you want to view.

1.3.6. Committing Changes

To apply your changes to a **Git** repository and create a new revision, change to the directory with the repository and type the following command at a shell prompt:

```
git commit [-m "commit message"]
```

Replace *commit message* with a short description of your revision. This command commits all changes in files that are explicitly marked as ready to be committed. To commit changes in all files that are under revision control, add the **-a** command line option as follows:

```
git commit -a [-m "commit message"]
```

Note that if you omit the **-m** option, the command allows you to write the commit message in an external text editor. For information on how to configure the default text editor, see Section 1.3.1, “Configuring the Default Text Editor”.

1.3.7. Sharing Changes

Unlike in centralized version control systems such as **CVS** or **Subversion**, when working with **Git**, project contributors usually do not make their changes in a single, central repository. Instead, they either create a publicly accessible clone of their local repository, or submit their changes to others over email as patches.

Pushing Changes to a Public Repository

To push your changes to a publicly accessible **Git** repository, change to the directory with your local repository and type the following at a shell prompt:

```
git push remote_repository
```

Replace *remote_repository* with the name of the remote repository you want to push your changes to. Note that the repository from which you originally cloned your local copy is automatically named **origin**.

Creating Patches from Individual Commits

To create patches from your commits, change to the directory with your local **Git** repository and type the following at a shell prompt:

```
git format-patch remote_repository
```

Replace *remote_repository* with the name of the remote repository from which you made your local copy. This creates a patch for each commit that is not present in this remote repository.

1.3.8. Updating a Repository

To update your local copy of a **Git** repository and get the latest changes from a remote repository, change to the directory with your local **Git** repository and type the following at a shell prompt:

```
git fetch remote_repository
```

Replace *remote_repository* with the name of the remote repository. This command fetches information about the current status of the remote repository, allowing you to review these changes before applying them to your local copy. To proceed and merge these changes with what you have in your local **Git** repository, type:

```
git merge remote_repository
```

Alternatively, you can perform both these steps at the same time by using the following command instead:

```
git pull remote_repository
```

1.3.9. Additional Resources

A detailed description of **Git** and its features is beyond the scope of this book. For more information about this revision control system, see the resources listed below.

Installed Documentation

- **gittutorial**(7) — The manual page named **gittutorial** provides a brief introduction to **Git** and its usage.
- **gittutorial-2**(7) — The manual page named **gittutorial-2** provides the second part of a brief introduction to **Git** and its usage.
- *Git User's Manual* — HTML documentation for **Git** is located at **/usr/share/doc/git-1.8.3/user-manual.html**.

Online Documentation

- Pro Git — The online version of the *Pro Git* book provides a detailed description of **Git**, its concepts and its usage.

Chapter 2. Libraries and Runtime Support

Red Hat Enterprise Linux supports the development of custom applications in a wide variety of programming languages using proven, industrial-strength tools. This chapter describes the runtime support libraries provided in Red Hat Enterprise Linux 7.

2.1. Version Information

The following table compares the version information for runtime support packages in supported programming languages between Red Hat Enterprise Linux 7, Red Hat Enterprise Linux 6, Red Hat Enterprise Linux 5, and Red Hat Enterprise Linux 4.

This is not an exhaustive list. Instead, this is a survey of standard language runtimes, and key dependencies for software developed on Red Hat Enterprise Linux 7.

Table 2.1. Language and Runtime Library Versions

Package Name	Red Hat Enterprise 7	Red Hat Enterprise 6	Red Hat Enterprise 5	Red Hat Enterprise 4
glibc	2.12	2.12	2.5	2.3
libstdc++	4.8	4.4	4.1	3.4
boost	1.53	1.41	1.33	1.32
java	1.7	1.5 (IBM), 1.6 (IBM, OpenJDK, Oracle Java)	1.4, 1.5, and 1.6	1.4
python	2.7	2.6	2.4	2.3
php	5.4	5.3	5.1	4.3
ruby	2.0	1.8	1.8	1.8
httpd	2.4	2.2	2.2	2.0
postgresql	9.2	8.4	8.1	7.4
mysql	5.4	5.1	5.0	4.1
nss	3.15	3.12	3.12	3.12
openssl	1.0.1e	1.0.0	0.9.8e	0.9.7a
libX11	1.6	1.3	1.0	
firefox	24.4	3.6	3.6	3.6
kdebase	4.10	4.3	3.5	3.3
gtk2	2.24	2.18	2.10	2.04

Note

The **compat-glibc** RPM is included with Red Hat Enterprise Linux 7, but it is not a runtime package and therefore not required for running anything. It is solely a development package, containing header files and dummy libraries for linking. This allows compiling and linking packages to run in older Red Hat Enterprise Linux versions (using **compat-gcc-*** against those headers and libraries). Running **rpm -qpi compat-glibc-*** will provide some information on how to use this package.

For more information on **compat-glib**, see Section 2.3.1, "**compat-glibc**"

2.2. Compatibility

Compatibility specifies the portability of binary objects and source code across different instances of a computer operating environment. Officially, Red Hat supports current release and two consecutive prior versions. This means that applications built on Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5 will continue to run on Red Hat Enterprise Linux 6 as long as they comply with Red Hat guidelines (using the symbols that have been white-listed, for example).

Red Hat understands that as an enterprise platform, customers rely on long-term deployment of their applications. For this reason, applications built against C/C++ libraries with the help of compatibility libraries continue to be supported for ten years.

There are two types of compatibility:

Source Compatibility

Source compatibility specifies that code will compile and execute in a consistent and predictable way across different instances of the operating environment. This type of compatibility is defined by conformance with specified Application Programming Interfaces (APIs).

Binary Compatibility

Binary Compatibility specifies that compiled binaries in the form of executables and *Dynamic Shared Objects* (DSOs) will run correctly across different instances of the operating environment. This type of compatibility is defined by conformance with specified Application Binary Interfaces (ABIs).

For further information regarding this and all levels of compatibility between core and non-core libraries, see Red Hat Enterprise Linux supported releases accessed at https://access.redhat.com/support/policy/updates/errata/ and the general Red Hat Enterprise Linux compatibility policy, accessed at https://access.redhat.com/site/articles/119073.

2.2.1. Static Linking

Static linking is emphatically discouraged for all Red Hat Enterprise Linux releases. Static linking causes far more problems than it solves, and should be avoided at all costs.

The main drawback of static linking is that it is only guaranteed to work on the system on which it was built, and even then only until the next release of glibc or libstdc++ (in the case of C++). There is no forward or backward compatibility with a static build. Furthermore, any security fixes (or general-purpose fixes) in subsequent updates to the libraries will not be available unless the affected statically linked executables are re-linked.

A few more reasons why static linking should be avoided are:

- Larger memory footprint.
- Slower application startup time.
- Reduced glibc features with static linking.
- Security measures like load address randomization cannot be used.
- Dynamic loading of shared objects outside of glibc is not supported.

For additional reasons to avoid static linking, see: Static Linking Considered Harmful.

2.3. Library and Runtime Details

2.3.1. compat-glibc

compat-glibc provides a subset of the shared static libraries from previous versions of Red Hat Enterprise Linux. For Red Hat Enterprise Linux 7, the following libraries are provided:

- **libanl**
- **libcidn**
- **libcrypt**
- **libc**
- **libdl**
- **libm**
- **libnsl**
- **libpthread**
- **libresolv**
- **librt**
- **libthread_db**
- **libutil**

This set of libraries allows developers to create a Red Hat Enterprise Linux 6 application with Red Hat Enterprise Linux 7, provided the application uses only the above libraries. Use the following command to do so:

```
# gcc -fgnu89-inline -I /usr/lib/x86_64-redhat-linux6E/include -B
/usr/lib/x86_64-redhat-linux6E/lib64/
```

Important

Applications that violate the ISO with regards to overlapping source or destination memory locations for **memcpy** and other functions will likely fail.

2.3.2. The GNU C++ Standard Library

The **libstdc++** package contains the GNU C++ Standard Library, which is an ongoing project to implement the ISO 14882 Standard C++ library.

Installing the **libstdc++** package will provide just enough to satisfy link dependencies (that is, only shared library files). To make full use of all available libraries and header files for C++ development, you must install **libstdc++-devel** as well. The **libstdc++-devel** package also contains a GNU-specific implementation of the Standard Template Library (STL).

For Red Hat Enterprise Linux 4, 5, and 6, the C++ language and runtime implementation has remained stable and as such no compatibility libraries are required for **libstdc++**. However, this is

not the case for Red Hat Enterprise Linux 2 and 3. For Red Hat Enterprise Linux 2 **compat-libstdc++-296** is required to be installed. For Red Hat Enterprise Linux 3 **compat-libstdc++-33** is required to be installed. Neither of these are installed by default so have to be added separately.

2.3.2.1. GNU C++ Standard Library Updates

The Red Hat Enterprise Linux 6 version of the GNU C++ Standard Library features the following improvements over its Red Hat Enterprise Linux 5 version:

- Added support for elements of ISO C++ TR1, namely:
  - **<tr1/array>**
  - **<tr1/complex>**
  - **<tr1/memory>**
  - **<tr1/functional>**
  - **<tr1/random>**
  - **<tr1/regex>**
  - **<tr1/tuple>**
  - **<tr1/type_traits>**
  - **<tr1/unordered_map>**
  - **<tr1/unordered_set>**
  - **<tr1/utility>**
  - **<tr1/cmath>**
- Added support for elements of the upcoming ISO C++ standard, C++0x. These elements include:
  - **<array>**
  - **<chrono>**
  - **<condition_variable>**
  - **<forward_list>**
  - **<functional>**
  - **<initalizer_list>**
  - **<mutex>**
  - **<random>**
  - **<ratio>**
  - **<regex>**
  - **<system_error>**
  - **<thread>**
  - **<tuple>**

- **<type_traits>**
- **<unordered_map>**
- **<unordered_set>**

- Added support for the **-fvisibility** command.
- Added the following extensions:
  - **__gnu_cxx::typelist**
  - **__gnu_cxx::throw_allocator**

For more information about updates to **libstdc++** in Red Hat Enterprise Linux, see the *C++ Runtime Library* section of the following documents:

- *GCC 4.2 Release Series Changes, New Features, and Fixes*: http://gcc.gnu.org/gcc-4.2/changes.html
- *GCC 4.3 Release Series Changes, New Features, and Fixes*: http://gcc.gnu.org/gcc-4.3/changes.html
- *GCC 4.4 Release Series Changes, New Features, and Fixes*: http://gcc.gnu.org/gcc-4.4/changes.html

2.3.2.2. GNU C++ Standard Library Documentation

To use the **man** pages for library components, install the **libstdc++-docs** package. This will provide **man** page information for nearly all resources provided by the library; for example, to view information about the **vector** container, use its fully-qualified component name:

```
man std::vector
```

This will display the following information (abbreviated):

```
std::vector(3)
std::vector(3)

NAME
       std::vector -

       A standard container which offers fixed time access to individual
       elements in any order.

SYNOPSIS
       Inherits std::_Vector_base< _Tp, _Alloc >.

   Public Types
       typedef _Alloc allocator_type
       typedef __gnu_cxx::__normal_iterator< const_pointer, vector >
           const_iterator
       typedef _Tp_alloc_type::const_pointer const_pointer
       typedef _Tp_alloc_type::const_reference const_reference
       typedef std::reverse_iterator< const_iterator >
```

The **libstdc++-docs** package also provides manuals and reference information in HTML form at the following directory:

file:///usr/share/doc/libstdc++-docs-*version*/html/spine.html

The main site for the development of libstdc++ is hosted on gcc.gnu.org.

2.3.3. Boost

The **boost** package contains a large number of free peer-reviewed portable C++ source libraries. These libraries are suitable for tasks such as portable file-systems and time/date abstraction, serialization, unit testing, thread creation and multi-process synchronization, parsing, graphing, regular expression manipulation, and many others.

Installing the **boost** package will provide just enough libraries to satisfy link dependencies (that is, only shared library files). To make full use of all available libraries and header files for C++ development, you must install **boost-devel** as well.

The **boost** package is actually a meta-package, containing many library sub-packages. These sub-packages can also be installed individually to provide finer inter-package dependency tracking. The meta-package includes all of the following sub-packages:

- **boost-date-time**
- **boost-filesystem**
- **boost-graph**
- **boost-iostreams**
- **boost-math**
- **boost-program-options**
- **boost-python**
- **boost-regex**
- **boost-serialization**
- **boost-signals**
- **boost-system**
- **boost-test**
- **boost-thread**
- **boost-wave**

Not included in the meta-package are packages for static linking or packages that depend on the underlying Message Passing Interface (MPI) support.

MPI support is provided in two forms: one for the default Open MPI implementation [1] , and another for the alternate MPICH2 implementation. The selection of the underlying MPI library in use is up to the user and depends on specific hardware details and user preferences. Please note that these packages can be installed in parallel, as installed files have unique directory locations.

For Open MPI:

- **boost-openmpi**
- **boost-openmpi-devel**
- **boost-graph-openmpi**
- **boost-openmpi-python**

For MPICH2:

- **boost-mpich2**
- **boost-mpich2-devel**
- **boost-graph-mpich2**
- **boost-mpich2-python**

If static linkage cannot be avoided, the **boost-static** package will install the necessary static libraries. Both thread-enabled and single-threaded libraries are provided.

2.3.3.1. Boost Updates

The Red Hat Enterprise Linux 6 version of Boost features many packaging improvements and new features.

Several aspects of the **boost** package have changed. As noted above, the monolithic **boost** package has been augmented by smaller, more discrete sub-packages. This allows for more control of dependencies by users, and for smaller binary packages when packaging a custom application that uses Boost.

In addition, both single-threaded and multi-threaded versions of all libraries are packaged. The multi-threaded versions include the **mt** suffix, as per the usual Boost convention.

Boost also features the following new libraries:

- Foreach
- Statechart
- TR1
- Typeof
- Xpressive
- Asio
- Bitmap
- Circular Buffer
- Function Types
- Fusion
- GIL
- Interprocess
- Intrusive
- Math/Special Functions
- Math/Statistical Distributions
- MPI
- System

- Accumulators
- Exception
- Units
- Unordered
- Proto
- Flyweight
- Scope Exit
- Swap
- Signals2
- Property Tree

Many of the existing libraries have been improved, bug-fixed, and otherwise enhanced.

2.3.3.2. Boost Documentation

The **boost-doc** package provides manuals and reference information in HTML form located in the following directory:

file:///usr/share/doc/boost-doc-*version*/index.html

The main site for the development of Boost is hosted on boost.org.

2.3.4. Qt

The **qt** package provides the Qt (pronounced "cute") cross-platform application development framework used in the development of GUI programs. Aside from being a popular "widget toolkit", Qt is also used for developing non-GUI programs such as console tools and servers. Qt was used in the development of notable projects such as Google Earth, KDE, Opera, OPIE, VoxOx, Skype, VLC media player and VirtualBox. It is produced by Nokia's Qt Development Frameworks division, which came into being after Nokia's acquisition of the Norwegian company Trolltech, the original producer of Qt, on June 17, 2008.

Qt uses standard C++ but makes extensive use of a special pre-processor called the *Meta Object Compiler* (MOC) to enrich the language. Qt can also be used in other programming languages via language bindings. It runs on all major platforms and has extensive internationalization support. Non-GUI Qt features include SQL database access, XML parsing, thread management, network support, and a unified cross-platform API for file handling.

Distributed under the terms of the GNU Lesser General Public License (among others), Qt is free and open source software. The Red Hat Enterprise Linux 6 version of Qt supports a wide range of compilers, including the GCC C++ compiler and the Visual Studio suite.

2.3.4.1. Qt Updates

Some of the improvements the Red Hat Enterprise Linux 6 version of Qt include:

- Advanced user experience
  - **Advanced Graphics Effects:** options for opacity, drop-shadows, blur, colorization, and other similar effects

- **Animation and State Machine:** create simple or complex animations without the hassle of managing complex code
- Gesture and multi-touch support

- Support for new platforms
  - Windows 7, Mac OSX 10.6, and other desktop platforms are now supported
  - Added support for mobile development; Qt is optimized for the upcoming Maemo 6 platform, and will soon be ported to Maemo 5. In addition, Qt now supports the Symbian platform, with integration for the S60 framework.
  - Added support for Real-Time Operating Systems such as QNX and VxWorks
- Improved performance, featuring added support for hardware-accelerated rendering (along with other rendering updates)
- Updated cross-platform IDE

For more details on updates to Qt included in Red Hat Enterprise Linux 6, see the following links:

- http://doc.qt.nokia.com/4.6/qt4-6-intro.html
- http://doc.qt.nokia.com/4.6/qt4-intro.html

2.3.4.2. Qt Creator

Qt Creator is a cross-platform IDE tailored to the requirements of Qt developers. It includes the following graphical tools:

- An advanced C++ code editor
- Integrated GUI layout and forms designer
- Project and build management tools
- Integrated, context-sensitive help system
- Visual debugger
- Rapid code navigation tools

2.3.4.3. Qt Library Documentation

The **qt-doc** package provides HTML manuals and references located in **/usr/share/doc/qt4/html/**. This package also provides the *Qt Reference Documentation*, which is an excellent starting point for development within the Qt framework.

You can also install further demos and examples from **qt-demos** and **qt-examples**. To get an overview of the capabilities of the Qt framework, see **/usr/bin/qtdemo-qt4** (provided by **qt-demos**).

2.3.5. KDE Development Framework

The **kdelibs-devel** package provides the KDE libraries, which build on Qt to provide a framework for making application development easier. The KDE development framework also helps provide consistency across the KDE desktop environment.

2.3.5.1. KDE4 Architecture

The KDE development framework's architecture in Red Hat Enterprise Linux uses KDE4, which is built on the following technologies:

Plasma

Plasma replaces KDesktop in KDE4. Its implementation is based on the **Qt Graphics View Framework**, which was introduced in Qt 4.2. For more information about **Plasma**, see http://techbase.kde.org/Development/Architecture/KDE4/Plasma.

Sonnet

Sonnet is a multilingual spell-checking application that supports automatic language detection, primary/backup dictionaries, and other useful features. It replaces **kspell2** in KDE4.

KIO

The KIO library provides a framework for network-transparent file handling, allowing users to easily access files through network-transparent protocols. It also helps provides standard file dialogs.

KJS/KHTML

KJS and KHTML are fully-fledged JavaScript and HTML engines used by different applications native to KDE4 (such as **konqueror**).

Solid

Solid is a hardware and network awareness framework that allows you to develop applications with hardware interaction features. Its comprehensive API provides the necessary abstraction to support cross-platform application development. For more information, see http://techbase.kde.org/Development/Architecture/KDE4/Solid.

Phonon

Phonon is a multimedia framework that helps you develop applications with multimedia functionalities. It facilitates the usage of media capabilities within KDE. For more information, see http://techbase.kde.org/Development/Architecture/KDE4/Phonon.

Telepathy

Telepathy provides a real-time communication and collaboration framework within KDE4. Its primary function is to tighten integration between different components within KDE. For a brief overview on the project, see http://community.kde.org/Real-Time_Communication_and_Collaboration.

Akonadi

Akonadi provides a framework for centralizing storage of *Parallel Infrastructure Management* (PIM) components. For more information, see http://techbase.kde.org/Development/Architecture/KDE4/Akonadi.

Online Help within KDE4

KDE4 also features an easy-to-use Qt-based framework for adding online help capabilities to applications. Such capabilities include tooltips, hover-help information, and **khelpcenter** manuals. For a brief overview on online help within KDE4, see http://techbase.kde.org/Development/Architecture/KDE4/Providing_Online_Help.

KXMLGUI

KXMLGUI is a framework for designing user interfaces using XML. This framework allows you to design UI elements based on "actions" (defined by the developer) without having to revise source code. For more information, see http://techbase.kde.org/Development/Architecture/KDE4/XMLGUI_Technology.

Strigi

Strigi is a desktop search daemon compatible with many desktop environments and operating systems. It uses its own **jstream** system which allows for deep indexing of files. For more information on the development of **Strigi**, see http://www.vandenoever.info/software/strigi/.

KNewStuff2

KNewStuff2 is a collaborative data sharing library used by many KDE4 applications. For more information, see http://techbase.kde.org/Projects/KNS2.

2.3.5.2. kdelibs Documentation

The **kdelibs-apidocs** package provides HTML documentation for the KDE development framework in **/usr/share/doc/HTML/en/kdelibs4-apidocs/**. The following links also provide details on KDE-related programming tasks:

- http://techbase.kde.org/
- http://techbase.kde.org/Development/Tutorials
- http://techbase.kde.org/Development/FAQs
- http://api.kde.org

2.3.6. NSS Shared Databases

The NSS shared database format, introduced on NSS 3.12, is now available in Red Hat Enterprise 6. This encompasses a number of new features and components to improve access and usability.

Included, is the NSS certificate and key database which are now sqlite-based and allow for concurrent access. The legacy **key3.db** and **cert8.db** are also replaced with new SQL databases called **key4.db** and **cert9.db**. These new databases will store PKCS #11 token objects, which are the same as what is currently stored in **cert8.db** and **key3.db**.

Having support for shared databases enables a system-wide NSS database. It resides in **/etc/pki/nssdb** where globally trusted CA certificates become accessible to all applications. The command **rv = NSS_InitReadWrite("sql:/etc/pki/nssdb");** initializes NSS for applications. If the application is run with root privileges, then the system-wide database is available on a read and write basis. However, if it is run with normal user privileges it becomes read only.

Additionally, a PEM PKCS #11 module for NSS allows applications to load into memory certificates and keys stored in PEM-formatted files (for example, those produced by openssl).

2.3.6.1. Backwards Compatibility

The binary compatibility guarantees made by NSS upstream are preserved in NSS for Red Hat Enterprise Linux 6. This guarantee states that the NSS 3.12 is backwards compatible with all older NSS 3.x shared libraries. Therefore, a program linked with an older NSS 3.x shared library will work without recompiling or relinking, and any applications that restrict the use of NSS APIs to the NSS

Public Functions remain compatible with future versions of the NSS shared libraries.

Red Hat Enterprise Linux 5 and 4 run on the same version of NSS as Red Hat Enterprise Linux 6 so there are no ABI or API changes. However, there are still differences as NSS's internal cryptographic module in Red Hat Enterprise Linux 6 is the one from NSS 3.12, whereas Red Hat Enterprise Linux 5 and 4 still use the older one from NSS 3.15. This means that new functionality that had been introduced with NSS 3.12, such as the shared database, is now available with Red Hat Enterprise Linux 6's version of NSS.

2.3.6.2. NSS Shared Databases Documentation

Mozilla's wiki page explains the system-wide database rationale in great detail and can be accessed *here*.

2.3.7. Python

The **python** package adds support for the Python programming language. This package provides the object and cached bytecode files required to enable runtime support for basic Python programs. It also contains the **python** interpreter and the **pydoc** documentation tool. The **python-devel** package contains the libraries and header files required for developing Python extensions.

Red Hat Enterprise Linux also ships with numerous **python**-related packages. By convention, the names of these packages have a **python** prefix or suffix. Such packages are either library extensions or python bindings to an existing library. For instance, **dbus-python** is a Python language binding for D-Bus.

Note that both cached bytecode (***.pyc**/***.pyo** files) and compiled extension modules (***.so** files) are incompatible between Python 2.4 (used in Red Hat Enterprise Linux 5), Python 2.6 (used in Red Hat Enterprise Linux 6), and Python 2.7 (used in Red Hat Enterprise Linux 7). As such, you will be required to rebuild any extension modules you use that are not part of Red Hat Enterprise Linux.

2.3.7.1. Python Updates

Red Hat Enterprise Linux 7 ships with Python 2.7. For information about these changes, see the following project resource:

- What's New in Python 2.7: http://docs.python.org/dev/whatsnew/2.7.html

Both resources also contain advice on porting code developed using previous Python versions.

Important

Python provides various APIs for use with C extension modules. One of these APIs, PyCObject, was deprecated in Python 2.7. By default, deprecation warnings are ignored so this will not normally cause any problems.

However, if the standard warning settings are overridden, there may be problems with modules that use PyCObject and assume that the import succeeds. In particular, if warnings have been set to "error", it is possible to make the Python interpreter abort or even segfault when importing such modules due to reading through the NULL pointer triggered by the deprecation error.

To enable errors-for-warnings and use such a module, add an override so that a **`PendiingDeprecationWarning`** is logged instead of raising an exception.

```
>>> import warnings
>>> warnings.simplefilter('error')
>>> warnings.simplefilter('default', PendingDeprecationWarning)
```

2.3.7.2. Python Debug Build

Red Hat Enterprise Linux 7 ships with a debug build of the python interpreter in the *python-debug* package.

The debug interpreter (found in **`/usr/bin/python-debug`**) runs at about half the speed as the optimized interpreter (found in **`/usr/bin/python`**) and requires extentions models to be rebuilt for it but is still of use when writing and debugging Python C extension modules. Within the debug build, optimization levels are turned down, making it easier to step through code within the debugger.

The debug build is configured with additional debug settings:

`--with-pydebug`

Adds various useful methods to sys, such as **`sys.gettotalrefcount()`** and **`sys.getobjects()`**.

`--with-count-allocs`

Enables the **`COUNT_ALLOCS`** setting, which adds a **`sys.getcounts()`** method, providing information on all types.The default upstream behavior is to always dump this information on **`stdout`** when the process exits. This is patched downstream so that the information is only dumped on exit if **`PYTHONDUMPCOUNTS`** is set in the environment.

`--with-call-profile`

Enables the **`CALL_PROFILE`** setting. This counts the number of function calls executed, and on how the interpreter handled those calls.

`--with-tsc (only on x86_64 and ppc64)`

Adds a **`sys.settscdump()`** method, adding very low-level profiling of the interpreter.

The debug build uses the same bytecode files as the regular optimized build, but extension modules (**.so** files) are not compatible. This is because the in-memory layout of Python objects differs due to the extra instrumentation. Given an optimized extension model ***foo*.so**, the debug build is patched to look for ***foo*_d.so**.

For more information on the debug build and its settings, see the notes upstream at http://svn.python.org/projects/python/trunk/Misc/SpecialBuilds.txt.

2.3.7.3. Python Documentation

For more information about Python, see **man python**. You can also install **python-docs**, which provides HTML manuals and references in the following location:

file:///usr/share/doc/python-docs-*version*/html/index.html

For details on library and language components, use **pydoc *component_name***. For example, **pydoc math** will display the following information about the **math** Python module:

```
Help on module math:

NAME
 math

FILE
 /usr/lib64/python2.6/lib-dynload/mathmodule.so

DESCRIPTION
 This module is always available.  It provides access to the
 mathematical functions defined by the C standard.

FUNCTIONS
 acos[...]
  acos(x)

  Return the arc cosine (measured in radians) of x.

 acosh[...]
  acosh(x)

  Return the hyperbolic arc cosine (measured in radians) of x.

 asin(...)
  asin(x)

  Return the arc sine (measured in radians) of x.

 asinh[...]
  asinh(x)

  Return the hyperbolic arc sine (measured in radians) of x.
```

The main site for the Python development project is hosted on python.org.

2.3.8. Java

Red Hat Enterprise Linux 7 is constantly updated to ship the latest version of JDK. The

java-*version_number*-openjdk package adds support for the Java programming language. This package provides the **java** interpreter. The **java-*version_number*-openjdk-devel** package contains the **javac** compiler, as well as the libraries and header files required for developing Java extensions.

Red Hat Enterprise Linux also ships with numerous **java**-related packages. By convention, the names of these packages have a **java** prefix or suffix.

2.3.8.1. Java Features

Java has a number of new features with Red Hat Enterprise Linux 7. These include the following:

Support for dynamically-typed languages (InvokeDynamic)

Enhancements are made to Hotspot, Open JDK's Java Virtual Machine (JVM). These are designed to support dynamically typed languages with minimal performance cost as compared to statically typed languages and Java itself. Specifically, the invokedynamic instruction was added to the Java bytecode specification and implemented in the JVM.

Small language enhancements (Project Coin)

A number of Java language-level improvements that provide programmer conveniences, more elegant code, and reduces some common programming errors.

Strings in switch

In prior Java versions, switch statements allowed the use of byte, short, char, and int primitives and their corresponding object types, as well as enums. As of Java 7, string values may also be used in switch statements.

Binary integral literals and underscores in numeric literals

Programmers may now express integral literals in binary form, or separate groups of digits in numerical literal values by underscores, in order to improve code readability.

Multi-catch

Java's catch syntax has been improved so that more than one exception type can be caught in a single catch clause, reducing redundant code.

More precise rethrow

The Java 7 compiler has been improved so that a method that catches and then rethrows an exception can be more precise in the throws clause of the method declaration in some circumstances.

Improved type inference for generic instance creation (diamond)

This syntactical improvement allows programmers to use the diamond operator (that is, <>) instead of the full generic type (for example, <ClassName>) when instantiating variables of generic types. The type is inferred from that variable's declaration instead.

Try-with-resources statement

This is a new form of try statement for use with closeable resources, such as streams and files. Using this feature, programmers no longer need to explicitly close these resources.

Simplified varags method invocation

Previously, a compiler warning would be issued when calling vararg methods with non-

reifiable arguments. This has been removed and replaced with a warning at the declaration of a vararg method that can accept non-reifiable arguments. An annotation can be used to suppress this warning, in which case the developer takes responsibility that the arguments are correct. This primarily applies to vararg methods that accept generic arguments.

Concurrency and collections updates

A number of new classes and interfaces, including a fork/join framework for divide and conquer type algorithms, has been added to the **java.util.concurrency** package. These can be useful for improving performance and correctness of multi-threaded programs.

New I/O AIPs for the Java platform

This includes a new file system API to improve cross-platform compatibility while making graceful failure handling easier for developers. It provides improved socket/channel API in the **java.nio.channels** package to remove unintuitive dependences on the **java.net** package. It also provides a new asynchronous I/O API.

Nimbus look and feel for swing

Informally introduced in Java 6 under the **com.sun.java.swing** package namespace, Nimbus has a vector-graphics based look and feel for swing. With Java 7, it has become an official API and moved to the **javax.swing** package.

2.3.8.2. Java Documentation

For more information about Java, see **man java**. Some associated utilities also have their own respective **man** pages.

You can also install other Java documentation packages for more details about specific Java utilities. By convention, such documentation packages have the **javadoc** suffix (for example, **dbus-java-javadoc**).

The main site for the development of Java is hosted on http://openjdk.java.net/. The main site for the library runtime of Java is hosted on http://icedtea.classpath.org.

2.3.9. Ruby

The **ruby** package provides the Ruby interpreter and adds support for the Ruby programming language. The **ruby-devel** package contains the libraries and header files required for developing Ruby extensions.

Red Hat Enterprise Linux also ships with numerous **ruby**-related packages. By convention, the names of these packages have a **ruby** or **rubygem** prefix or suffix. Such packages are either library extensions or Ruby bindings to an existing library.

Examples of **ruby**-related packages include:

- ruby-irb
- ruby-libguestfs
- ruby-libs
- ruby-qpid
- ruby-rdoc

- ruby-ri
- ruby-tcltk
- rubygems
- rubygem-bigdecimal
- rubygem-devel
- rubygem-io-console
- rubygem-json
- rubygem-minitest
- rubygem-rake

Note

If the Bundler is used for managing application dependencies, please always use the Bundler provided by the **rubygem-bundler** package. The upstream package is not compatible with the RubyGems layout used by Red Hat Enterprise Linux 7.

2.3.9.1. Ruby Updates

For information about updates to the Ruby language in Red Hat Enterprise Linux 7, see the following resources:

- **file:///usr/share/doc/ruby-*version*/NEWS**
- **file:///usr/share/doc/ruby-*version*/NEWS-*version***

Ruby has undergone significant changes in its filesystem layout, which now better conforms with FHS. Binary libraries and extensions of Gems are placed under **/usr/lib** (or **/usr/lib64** for 64-bit systems) and pure Ruby libraries and Gems are placed under **/usr/share**. Gems are located in three places according to the selected method of their installation:

- **/usr**
- **/usr/local**
- **~/.gem**

2.3.9.2. Ruby Documentation

For more information about Ruby, see **man ruby**. You can also use the **ri** command, which is the Ruby API reference front end. For gem documentation, use the **gem server** command that makes HTML manuals and references about gems installed on your system available in a browser.

Note

It may be necessary to install the *-doc* sub-package to make the documentation available using the **ri** and **gem server** commands.

The main site for the development of Ruby is hotsed on http://www.ruby-lang.org. The http://www.ruby-doc.org site also contains Ruby documentation. Online documentation for gems can be found at http://rdoc.info.

Documentation for the **ri** command can be found in **/usr/share/ri/system**.

2.3.10. Perl

The **perl** package adds support for the Perl programming language. This package provides some of the Perl core modules, the Perl Language Interpreter, and the **perldoc** tool. Red Hat Enterprise Linux 7 ships with *perl-5.16*. To install all of the core modules, use the **yum install perl-core** command.

Red Hat also provides various perl modules in package form; these packages are named with the **perl-*** prefix. These modules provide stand-alone applications, language extensions, Perl libraries, and external library bindings.

An RPM package can contain more Perl modules. Each module intended for public use is provided by the package in the form *perl(The::Module)*. This expression can be passed to **yum** to install the appropriate packages.

> **Example 2.1. Install perl module**
>
> ```
> # yum install 'perl(LWP::UserAgent)'
> ```
>
> This will install the RPM package *perl-libwww-perl*, which contains the **LWP::UserAgent** module, allowing a programer to use the command **use LWP::UserAgent;**.

2.3.10.1. Perl Updates

Red Hat Enterprise Linux 7 ships with perl 5.16 which has a number of changes since the 5.10 version shipped in Red Hat Enterprise Linux 6. These include:

Perl 5.12 Updates

Perl 5.12 has the following updates:

- Perl conforms closer to the Unicode standard.
- Experimental APIs allow Perl to be extended with "pluggable" keywords and syntax.
- Perl will be able to keep accurate time well past the "Y2038" barrier.
- Package version numbers can be directly specified in "package" statements.
- Perl warns the user about the use of depreciated features by default.

The Perl 5.12 delta can be accessed at http://perldoc.perl.org/perl5120delta.html.

Perl 5.14 Updates

Perl 5.14 has the following updates:

- Unicode 6.0 support.
- Improved support for IPv6.

- Easier auto-configuration of the CPAN client.
- A new /r flag that makes s/// substitutions non-destructive.
- New regular expression flags to control whether matched strings should be treated as ASCII or Unicode.
- New **`package Foo { }`** syntax.
- Less memory and CPU usage than previous releases.
- A number of bug fixes.

The Perl 5.14 delta can be accessed at http://perldoc.perl.org/perl5140delta.html.

Perl 5.16 Updates

Perl 5.16 has the following updates:

- Support for Unicode 6.1.
- **$$** variable is writable.
- Improved debugger.
- Accessing Unicode database files directly is now depreciated; use *Unicode::UCD* instead.
- *Version::Requirements* is depreciated in favor of *CPAN::Meta::Requirements*.
- A number of perl4 libraries are removed:
  - *abbrev.pl*
  - *assert.pl*
  - *bigfloat.pl*
  - *bigint.pl*
  - *bigrat.pl*
  - *cacheout.pl*
  - *complete.pl*
  - *ctime.pl*
  - *dotsh.pl*
  - *exceptions.pl*
  - *fastcwd.pl*
  - *flush.pl*
  - *getcwd.pl*
  - *getopt.pl*
  - *getopts.pl*
  - *hostname.pl*

- *importenv.pl*
- *lib/find.pl*
- *lib/finddepth.pl*
- *look.pl*
- *newgetopt.pl*
- *open2.pl*
- *open3.pl*
- *pwd.pl*
- *hellwords.pl*
- *stat.pl*
- *tainted.pl*
- *termcap.pl*
- *timelocal.pl*

The Perl 5.16 delta can be accessed at http://perldoc.perl.org/perl5160delta.html.

2.3.10.2. Installation

Perl's capabilities can be extended by installing additional modules. These modules come in the following forms:

Official Red Hat RPM

The official module packages can be installed with **yum** or **rpm** from the Red Hat Enterprise Linux repositories. They are installed to **/usr/share/perl5** and either **/usr/lib/perl5** for 32bit architectures or **/usr/lib64/perl5** for 64bit architectures, as well as **vendor_perl** subdirectories.

Modules from CPAN

Use the **cpan** tool provided by the *perl-CPAN* package to install modules directly from the CPAN website. They are installed to **/usr/local/share/perl5** and either **/usr/local/lib/perl5** for 32bit architectures or **/usr/local/lib64/perl5** for 64bit architectures if these directories exist and are writable by the current user.

If the directories do not exist the **cpan** tool will offer different solutions.

```
Warning: You do not have write permission for Perl library
directories.

To install modules, you need to configure a local Perl library
directory or escalate your privileges. CPAN can help you by
bootstrapping the local::lib module or by configuring itself to
use 'sudo' (if available). You may also resolve this problem
manually if you need to customize your setup.
```

```
What approach do you want? (Choose 'local::lib', 'sudo' or
'manual')
[local::lib]
```

For example, if 'manual' is selected, it will assme the user will ensure the directories exist and are writable before installing modules from CPAN.

Third party and custom module packages

These packaged modules are installed to **/usr/share/perl5/vendor_perl** and either **/usr/lib/perl5/vendor_perl** for 32bit architectures or **/usr/lib64/perl5/vendor_perl** for 64bit architectures. If their file names conflict with Red Hat Enterprise Linux packages, either change the file names or properly replace the Red Hat Enterprise Linux packages with the delivering packages.

Warning

If an official version of a module is already installed, installing its non-official version can create conflicts in the **/usr/share/man** directory.

If an additional Perl module search path is necessary, the **/usr/local/share/perl5/sitecustomize.pl** script can be used for system-wide modification (see the **perlrun(1)** man page), or *perl-homedir* package for user specific modifications (see the *perl-homedir* package description).

2.3.10.3. Perl Documentation

The **perldoc** tool provides documentation on language and core modules. To learn more about a module, use **perldoc module_name**. For example, **perldoc CGI** will display the following information about the CGI core module:

```
NAME
CGI - Handle Common Gateway Interface requests and responses

SYNOPSIS
use CGI;

my $q = CGI->new;

[...]

DESCRIPTION
CGI.pm is a stable, complete and mature solution for processing and
preparing HTTP requests and responses.  Major features including
processing form submissions, file uploads, reading and writing cookies,
query string generation and manipulation, and processing and preparing
HTTP headers. Some HTML generation utilities are included as well.

[...]

PROGRAMMING STYLE
There are two styles of programming with CGI.pm, an object-oriented style
and a function-oriented style.  In the object-oriented style you create
```

```
one or more CGI objects and then use object methods to create the various
elements of the page.  Each CGI object starts out with the list of named
parameters that were passed to your CGI script by the server.

[...]
```

For details on Perl functions, use **perldoc -f *function_name***. For example, perldoc -f split wil display the following information about the split function:

```
split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split   Splits the string EXPR into a list of strings and returns that
list.  By default, empty leading fields are preserved, and empty trailing
ones are deleted.  (If all fields are empty, they are considered to be
trailing.)

In scalar context, returns the number of fields found. In scalar and void
context it splits into the @_ array.  Use of split in scalar and void
context is deprecated, however, because it clobbers your subroutine
arguments.

If EXPR is omitted, splits the $_ string.  If PATTERN is also omitted,
splits on whitespace (after skipping any leading whitespace).  Anything
matching PATTERN is taken to be a delimiter separating the fields.  (Note
that the delimiter may be longer than one character.)

[...]
```

Current perldoc documentation can be found on perldoc.perl.org.

Core and external modules are documented on the Comprehensive Perl Archive Network.

2.3.11. `libStorageMgmt` Plug-ins

Red Hat Enterprise Linux 7 ships with a new library called **libStorageMgmt**. It is a storage array independent Application Programming Interface (API) that provides a stable and consistent API allowing developers to programmatically manage different storage arrays and leverage the hardware accelerated features provided.

For more information on the **libStorageMgmt** library see Red Hat's *Storage Administration Guide*. This section details how to write plug-ins for the library.

Plug-ins work somewhat differently with the **libStorageMgmt** library. The plug-ins execute in their own address space as stand-alone executables with inter-process communication (IPC) between the client and plug-in. When a client application or the **libStorageMgmt** command line (**lsmcli**) utilizes the library, the following occurs:

1. The library uses the uniform resource identifier (URI) and parses out which plug-in was specified. For example, **LSMCLI_URI=sim://** refers to the simulator plug-in.

2. The library uses the plut-in name **sim** and looks for the unix domain socket in the socket directory. The default directory is **/var/run/lsm/ipc** but this can be changed at run-time by specifying the **LSM_UDS_PATH** environment variable.

3. The client library opens the unix domain socket, causing the **lsmd** daemon to accept the connection from the client. The daemon then forks and executes the plut-in, passing the socket descriptor on the command line to the plug-in. The client process now has a direct connection to the plug-in.

4. The **lsmd** is no longer in the path and goes back to sleep waiting for another process to open a socket.

There are a number of benifits to this different design. These include:

- If a daemon dies or is killed, any existing client plug-in sessions remain active.
- If a plug-in crashes, the client process will remain operational.
- The daemon needs to know nothing of the IPC protocol, keeping it simple.
- The plug-in can be closed source if required by the vendor.

2.3.11.1. Writing a plug in for `libStorageMgmt` library

The **libStorageMgmt** library has a plug-in API for both C and Python. Any language that supports suckets and text can also be used to write a plug-in, but the library provides the abstraction that hides this complexity.

The following are some general guidelines for plug-in design regardless of the programming language used:

Threading or multi-process

The library does not provide locking, nor does it keep any global state. As such, it is valid for a client to have a spearate plug-in instance in use for each thread or process. Plug-ins can anticipate that multiple instances of themselves can and possibley will be running at concurrently to different arrays. As the library provides a mechanism for long-running operations, multiple plug-in instances for the same array are not needed.

Plug-ins execute with non-root privilages

To reduce the potential for local exploits, plug-ins have reduced privilages. This needs to be taken into account when writing and designing plug-ins.

Plug-in lifetime

The client API provides for a handle that is opened and closed for each plug-in instance. During this time the plug-in is free to cache whatever data is necessary to provide correct opperation. When using the **lsmcli** tool, the lifetime is only for one command.

Logging

Plug-ins log errors to **syslog**. Helper functions exist to facilitate this in the library.

Errors

The library uses well defined error codes in order to remain language agnostic. Additional error data can be retrieved when they occur to provide textual error messages and optionally debug data from the plug-in or the array itslef. It is the library callers' responsibility to retrieve this additional information after an error occurs and before issuing another command. If additional error data exists

and other functions are called, then the aditional error information will be lost. C does not support exceptions. For languages that do support exceptions, a custom exception class containing the error code and additional information is provided.

Location and naming

Plug-ins are located in the **/usr/bin** directory. The name format must be **_lsmplugin**. This is because when the daemon startsit iterates in the directory enumerating them.

Job Control

The methods to get and set the time-out are used to specify how long the plug-in waits for a response from the array. If an operation can not safely complete within the time-out, the call returns a job id so that the client can check on the status of the operation. Job IDs are free form strings and are plug-in defined. The plug-in implementation needs to determine everything about the asynchronous operation from this string between invocations of the plug-in.

To write a plug-in, the following base functions or methods are required for all plug-ins, regardless of the language used:

- get/set timeout
- startup/shutdown
- job status
- job free
- capabilities
- plug-in information
- pools
- systems

A unique name must also be chosen so that the main execuatble has the form ***name*_lsmplugin**.

The following sections detail how to write a plug-in for python and for C.

2.3.11.1.1. Writing a plug-in with Python

First, implement the interface that supports the level of functionality to be provided (see **iplugin.py**). Most plug-ins will either inherit from IStorageAreaNetwork or INfs, or both if the plug-in supports block and network file systems.

Next, call the plug-in runner, passing the name of the class and the command line arguments to it for processing and executing the run method.

```
#!/usr/bin/env python
import sys

from lsm.pluginrunner import PluginRunner
from lsm.simulator import StorageSimulator

if __name__ == '__main__':
    PluginRunner(StorageSimulator, sys.argv).run()
```

Note

During development it is possible to call the plug-in directly on the command line for easier debugging.

2.3.11.1.2. Writing a plug-in with C

First, include the required header file **`#include <libstoragemgmt/libstoragemgmt_plug_interface.h>`**.

Then, implement the callback functions that will be supported, along with the required ones.

Finally, pass the command line count and arguments to the library with load and unload functions.

```
#include <libstoragemgmt/libstoragemgmt_plug_interface.h>
#include <stdlib.h>
#include <stdint.h>

static char name[] = "Simple limited plug-in example";
static char version [] = "0.01";

struct plugin_data {
    uint32_t tmo;
    /* All your other variables as needed */
};

/* Create the functions you plan on implementing that
    match the callback signatures */
static int tmoSet(lsm_plugin_ptr c, uint32_t timeout, lsm_flag flags )
{
    int rc = LSM_ERR_OK;
    struct plugin_data *pd = (struct
plugin_data*)lsm_private_data_get(c);
    /* Do something with state to set timeout */
    pd->tmo = timeout;
    return rc;
}

static int tmoGet(lsm_plugin_ptr c, uint32_t *timeout, lsm_flag flags )
{
    int rc = LSM_ERR_OK;
    struct plugin_data *pd = (struct
plugin_data*)lsm_private_data_get(c);
    /* Do something with state to get timeout */
    *timeout = pd->tmo;
    return rc;
}

/* Setup the function addresses in the appropriate
    required callback structure */
static struct lsm_mgmt_ops_v1 mgmOps = {
    tmoSet,
    tmoGet,
    NULL,
```

```
    NULL,
    NULL,
    NULL,
    NULL
};

int load( lsm_plugin_ptr c, const char *uri, const char *password,
                        uint32_t timeout, lsm_flag flags )
{
    /* Do plug-in specific init. and setup callback structures */
    struct plugin_data *data = (struct plugin_data *)
                                malloc(sizeof(struct plugin_data));

    if (!data) {
        return LSM_ERR_NO_MEMORY;
    }

    /* Call back into the framework */
    int rc = lsm_register_plugin_v1( c, data, &mgmOps, NULL, NULL, NULL);
    return rc;
}

int unload( lsm_plugin_ptr c, lsm_flag flags)
{
    /* Get a handle to your private data and do clean-up */
    struct plugin_data *pd = (struct
plugin_data*)lsm_private_data_get(c);
    free(pd);
    return LSM_ERR_OK;
}

int main(int argc, char *argv[] )
{
    return lsm_plugin_init_v1(argc, argv, load, unload, name, version);
}
```

2.3.11.2. Writing Plug-in References

- The libStorageMgmt Writing Plug-ins wiki
  https://sourceforge.net/p/libstoragemgmt/wiki/WritingPlugins/

[1] MPI support is not available on IBM System Z machines (where Open MPI is not available).

Chapter 3. Compiling and Building

Red Hat Enterprise Linux includes many packages used for software development, including tools for compiling and building source code. This chapter discusses several of these packages and tools used to compile source code.

3.1. GNU Compiler Collection (GCC)

The GNU Compiler Collection (GCC) is a set of tools for compiling a variety of programming languages (including C, C++, ObjectiveC, ObjectiveC++, Fortran, and Ada) into highly optimized machine code. These tools include various compilers (like **gcc** and **g++**), run-time libraries (like **libgcc**, **libstdc++**, **libgfortran**, and **libgomp**), and miscellaneous other utilities.

Red Hat Enterprise Linux 7 is on PPC64 architecture. This means that GCC generates code for POWER7 platforms with tuning to POWER7 by default. The following tables detail the default codes for various platforms:

Table 3.1. Default Codes for i?86

Option	Default Code
mtune	generic
march	x86-64

redhat-rpm-config, used for building packages, also contains the **-mfpmath=sse** option.

Table 3.2. Default Codes for x86_64

Option	Default Code
mtune	generic
march	x86-64

Table 3.3. Default Codes for s390 and s390x

Option	Default Code
mtune	zEC12
march	z196

Table 3.4. Default Codes for ppc, ppc64, and ppc64p7

Option	Default Code
mtune	power7
mcpu	power7

The above table is for both 32-bit and 64-bit compilations.

It is possible to generate code for other targets using the **-march=CPU** option. Tuning the code for a specific chip is performed by using the **-mtune=CPU** option. The **-march** option implies the **-mtune** option. The **-march=native** option instructs the compiler to generate code for the processor type of the compiling machine.

Note

The **-march** option does not exist on PPC64 architecture. Use the **-mcpu** option instead.

3.1.1. Changes in GCC

Red Hat Developer Toolset 2.0 is distributed with **GCC 4.8**, which provides a number of bug fixes and feature enhancements over the Red Hat Enterprise Linux system version and the version included in Red Hat Developer Toolset 1.1. Below is a comprehensive list of new features and compatibility changes in this release.

3.1.1.1. Changes Since Red Hat Developer Toolset 1.1

The following features have been added since the release of GCC included in Red Hat Developer Toolset 1.1.

3.1.1.1.1. Caveats

Aggressive Loop Optimizations

The loop optimizer of GCC has been improved to use language constraints in order to derive bounds for the number of iterations of a loop. The bounds are then used as a guide to loop unrolling, peeling, and loop exit test optimizations.

The optimizations assume that the loop code does not invoke undefined behavior by, for example, causing signed integer overflows or making out-of-bound array accesses. For example, consider the following code fragment:

```
unsigned int foo()
{
    unsigned int data_data[128];

    for (int fd = 0; fd < 128; ++fd)
        data_data[fd] = fd * (0x02000001); // error

    return data_data[0];
}
```

When the value of the **fd** variable is 64 or above, the **fd * 0x02000001** operation overflows, which is invalid in both C and C++ for signed integers. In the example above, GCC may generate incorrect code or enter an infinite loop.

To fix this error, use the appropriate casts when converting between signed and unsigned types to avoid overflows, for instance:

```
data_data[fd] = (uint32_t) fd * (0x02000001U); // ok
```

If necessary, this optimization can be turned off by using the new command line option **-fno-aggressive-loop-optimizations**.

3.1.1.1.2. General Improvements and Changes

New Local Register Allocator

GCC 4.8 features a new *Local Register Allocator* (LRA), which replaces the 26-year old reload pass and improves the quality of generated code. The new local register allocator is meant to be simpler, easier to debug, and does a better job of register allocation.

AddressSanitizer

A fast memory error detector called *AddressSanitizer* has been added and can be enabled by using the **-fsanitize=address** command line option. It augments memory access instructions in order to detect use-after-free and out-of-bound accesses to objects on the heap.

ThreadSanitizer

A fast data race detector called *ThreadSanitizer* has been added in GCC 4.8. The option to enable this feature is **-fsanitize=thread**.

Compiling Extremely Large Functions

Many scalability bottlenecks have been removed from GCC optimization passes. As a consequence, it is now possible to compile extremely large functions with smaller memory consumption in less time.

New -Og Optimization Level

A new general optimization level, **-Og**, has been introduced. This optimization level addresses the need for fast compilation and a superior debugging experience while providing a reasonable level of runtime performance. Overall, the development experience should be better than the default optimization level **-O0**.

Caret Diagnostic Messages

The diagnostic messages of GCC, which display a line of source code, now also show a caret that indicates the column where the problem was detected. For example:

```
fred.cc:4:15: fatal error: foo: No such file or directory
 #include <foo>
               ^
compilation terminated.
```

New -fira-hoist-pressure Option

A new command line option, **-fira-hoist-pressure**, has been added. This option uses the register allocator to help decide when it is worthwhile to move expressions out of loops. It can reduce the size of the compiler code, but it slows down the compiler. This option is enabled by default at **-Os**.

New -fopt-info Option

A new command line option, **-fopt-info**, has been added. This option controls printing information about the effects of particular optimization passes, and takes the following form:

```
-fopt-info[-info][=file_name]
```

The *info* part of the option controls what is printed. Replace it with **optimized** to print information when optimization takes place, **missed** to print information when optimization does not take place, **note** to print more verbose information, or **optall** to print everything.

Replace *file_name* with the name of the file in which you want the information to be written. If you omit this part of the option, GCC writes the information to the standard error output stream.

For example, to display a list of optimizations that were enabled by the **-O2** option but had no effect when compiling a file named **foo.c**, type:

```
gcc -O2 -fopt-info-missed foo.c
```

New -floop-nest-optimize Option

A new command line option, **-floop-nest-optimize**, has been added. This option enables an experimental ISL-based loop nest optimizer, a generic loop nest optimizer that is based on the Pluto optimization algorithms and that calculates a loop structure optimized for data-locality and paralelism. For more information about this optimizer, see http://pluto-compiler.sourceforge.net.

Hot and Cold Attributes on Labels

The hot and cold function attributes can now also be applied to labels. Hot labels tell the compiler that the execution path following the label is more likely than any other execution path, and cold labels convey the opposite meaning. These attributes can be used in cases where **__builtin_expect** cannot be used, for instance with a computed **goto** or **asm goto**.

3.1.1.1.3. Debugging Enhancements

DWARF4

DWARF4 is now used as the default debugging data format when generating debugging information. To get the maximum benefit from this new debugging representation, use the latest version of **Valgrind**, **elfutils**, and **GDB** included in this release.

New -gsplit-dwarf Option

A new command line option, **-gsplit-dwarf**, has been added. This option tells the compiler driver to separate as much DWARF debugging information as possible into a separate output file with the **.dwo** file extension, and allows the build system to avoid linking files with debugging information.

In order to be useful, this option requires a debugger capable of reading **.dwo** files, such as the version of **GDB** included in Red Hat Developer Toolset 2.0.

Note

elfutils, **SystemTap**, and **Valgrind** do *not* support the **.dwo** files.

3.1.1.1.4. C++ Changes

Experimental C++ Features from an Upcoming Standard

g++ now supports a new command line option, **-std=c++1y**. This option can be used for experimentation with features proposed for the next revision of the standard that is expected around 2014. Currently, the only difference from **-std=c++11** is support for return type deduction in normal functions as proposed in N3386.

New thread_local Keyword

g++ now implements the C++11 **thread_local** keyword. In comparison with the GNU **__thread** keyword, **thread_local** allows dynamic initialization and destruction semantics.

The use of the **thread_local** keyword has currently one important limitation: when the **dlclose()** function is used to unload a dynamically loaded DSO that contains the definition of a **thread_local** object, the **thread_local** object is destroyed, its destructor is called and the DSO is unmapped from the address space of the process. If a thread in the process tries to access the **thread_local** object after this, the program may terminate unexpectedly. As a result, the programmer may have to take extra care to ensure that **thread_local** objects in a DSO are not referred after it has been unloaded.

See also the next item for dynamic initialization issues.

Dynamic Initialization of Thread-local Variables

The C++11 and OpenMP standards allow thread-local and thread-private variables to have dynamic (that is, runtime) initialization. To support this, any use of such a variable goes through a wrapper function that performs necessary initialization.

When the use and definition of the variable are in the same translation unit, this overhead can be optimized away, but when the use is in a different translation unit, there is significant overhead even if the variable does not actually need dynamic initialization. If the programmer can be sure that no use of the variable in a non-defining translation unit needs to trigger dynamic initialization (either because the variable is statically initialized, or a use of the variable in the defining translation unit will be executed before any uses in another translation unit), they can avoid this overhead by using the new **-fno-extern-tls-init** option.

By default, **g++** uses the **-fextern-tls-init** option.

C++11 Attribute Syntax

g++ now implements the C++11 attribute syntax, for example:

```
[[noreturn]] void f();
```

C++11 Alignment Specifier

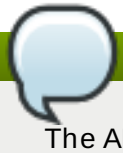
g++ now implements the C++11 alignment specifier, for example:

```
alignas(double) int i;
```

3.1.1.1.5. Fortran Changes

3.1.1.1.5.1. Caveats

The version of module files (the **.mod** files) has been incremented. Fortran modules compiled by earlier GCC versions have to be recompiled when they are used by files compiled with GCC 4.8, as this version of GCC is not able to read **.mod** files created by earlier versions; attempting to do so fails with an error message.



Note

The ABI of the produced assembler data itself has not changed; object files and libraries are fully compatible with older versions except as noted in Section 3.1.1.1.5.2, “ABI Compatibility”.

3.1.1.1.5.2. ABI Compatibility

Some internal names used in the assembler or object file have changed for symbols declared in the specification part of a module. If an affected module — or a file using it via use association — is recompiled, the module and all files which directly use such symbols have to be recompiled as well. This change only affects the following kind of module symbols:

- *Procedure pointers.* Note that C-interoperable function pointers (**type(c_funptr)**) are not affected, nor are procedure-pointer components.
- *Deferred-length character strings.*

3.1.1.1.5.3. Other Changes

BACKTRACE Intrinsic

A new intrinsic subroutine, **BACKTRACE**, has been added. This subroutine shows a backtrace at an arbitrary place in user code, program execution continues normally afterwards.

Floating Point Numbers with “q” as Exponential

Reading floating point numbers that use **q** for the exponential (such as **4.0q0**) is now supported as a vendor extension for better compatibility with old data files. It is strongly recommended to use the equivalent but standard conforming **e** (such as **4.0e0**) for I/O.

For Fortran source code, consider replacing the **q** in floating-point literals by a kind parameter (such as **4.0e0_qp** with a suitable **qp**). Note that — in Fortran source code — replacing **q** with a simple **e** is not equivalent.

GFORTRAN_TMPDIR Environment Variable

The **GFORTRAN_TMPDIR** environment variable for specifying a non-default directory for files opened with **STATUS="SCRATCH"**, is not used anymore. Instead, **gfortran** checks the POSIX/GNU standard **TMPDIR** environment variable and if **TMPDIR** is not defined, **gfortran** falls back to other methods to determine the directory for temporary files as documented in the user manual.

Fortran 2003

Support for unlimited polymorphic variables (**CLASS(*)**) has been added. Non-constant character lengths are not yet supported.

TS 29113

Assumed types (**TYPE(*)**) are now supported.

Experimental support for assumed-rank arrays (**dimension(..)**) has been added. Note that at the moment, the **gfortran** array descriptor is used, which is different from the array descriptor defined in *TS 29113*. For more information, see the header file of **gfortran** or use the **Chasm** language interoperability tools.

3.1.1.1.6. x86-specific Improvements

New Instructions

GCC 4.8 has added support for the Intel **FXSR**, **XSAVE**, and **XSAVEOPT** instructions. Corresponding intrinsics and built-in functions can now be enabled by using the **-mfxsr**, **-mxsave**, and **-mxsaveopt** command line options respectively.

In addition, support for the **RDSEED**, **ADCX**, **ADOX**, and **PREFETCHW** instructions has been added and can be enabled by using the **-mrdseed**, **-madx**, and **-mprfchw** command line options.

New Built-in Functions to Detect Run-time CPU Type and ISA

A new built-in function, **__builtin_cpu_is()**, has been added to detect if the run-time CPU is of a particular type. This function accepts one string literal argument with the CPU name, and returns a positive integer on a match and zero otherwise. For example, **__builtin_cpu_is("westmere")** returns a positive integer if the run-time CPU is an Intel Core i7 Westmere processor. For a complete list of valid CPU names, see the user manual.

A new built-in function, **__builtin_cpu_supports()**, has been added to detect if the run-time CPU supports a particular ISA feature. This function accepts one string literal argument with the ISA feature, and returns a positive integer on a match and zero otherwise. For example, **__builtin_cpu_supports("ssse3")** returns a positive integer if the run-time CPU supports **SSSE3** instructions. For a complete list of valid ISA names, see the user manual.

Important

If these built-in functions are called before any static constructors are invoked, such as **IFUNC** initialization, then the CPU detection initialization must be explicitly run using this newly provided built-in function, **__builtin_cpu_init()**. The initialization needs to be done only once. For example, the following is sample invocation inside an **IFUNC** initializer:

```
static void (*some_ifunc_resolver(void))(void)
{
     __builtin_cpu_init();
     if (__builtin_cpu_is("amdfam10h") ...
     if (__builtin_cpu_supports("popcnt") ...
}
```

Function Multiversioning

Function multiversioning allows the programmer to specify multiple versions of the same function, each of which is specialized for a particular variant of a given target. At runtime, the appropriate version is automatically executed depending upon the target where the execution takes place. For example, consider the following code fragment:

```
__attribute__ ((target ("default"))) int foo () { return 0; }
__attribute__ ((target ("sse4.2"))) int foo () { return 1; }
__attribute__ ((target ("arch=atom"))) int foo () { return 2; }
```

When the function **foo()** is executed, the result returned depends upon the architecture where the program runs, not the architecture where the program was compiled. See the GCC Wiki for more details.

New RTM and HLE Intrinsics

Support for the Intel RTM and HLE intrinsics, built-in functions, and code generation has been added and can be enabled by using the **-mrtm** and **-mhle** command line options. This is done via intrinsics for *Restricted Transactional Memory* (RTM) and extensions to the memory model for *Hardware Lock Elision* (HLE).

For HLE, two new flags can be used to mark a lock as using hardware elision:

__ATOMIC_HLE_ACQUIRE

Starts lock elision on a lock variable. The memory model in use must be **__ATOMIC_ACQUIRE** or stronger.

__ATOMIC_HLE_RELEASE

Ends lock elision on a lock variable. The memory model must be **__ATOMIC_RELEASE** or stronger.

For example, consider the following code fragment:

```
while (__atomic_exchange_n (& lockvar, 1, __ATOMIC_ACQUIRE
                                        | __ATOMIC_HLE_ACQUIRE))
    _mm_pause ();

// work with the acquired lock

__atomic_clear (& lockvar, __ATOMIC_RELEASE | __ATOMIC_HLE_RELEASE);
```

The new intrinsics that support Restricted Transactional Memory are:

unsigned _xbegin (void)

Attempts to start a transaction. If it succeeds, this function returns **_XBEGIN_STARTED**, otherwise it returns a status value indicating why the transaction could not be started.

void _xend (void)

Commits the current transaction. When no transaction is active, this function causes a fault. All memory side effects of the transactions become visible to other threads in an atomic manner.

int _xtest (void)

Returns a non-zero value if a transaction is currently active, or zero if it is not.

void _xabort (unsigned char status)

Aborts the current transaction. When no transaction is active, this is a no-op. The parameter **status** is included in the return value of any **_xbegin()** call that is aborted by this function.

The following example illustrates the use of these intrinsics:

```
if ((status = _xbegin ()) == _XBEGIN_STARTED)
{
    // some code
    _xend ();
}
else
{
    // examine the status to see why the transaction failed and possibly
retry
}
```

Transactions Using Transactional Synchronization Extensions

Transactions in the transactional memory feature (the **-fgnu-tm** option) of GCC can now be run using *Transactional Synchronization Extensions* (TSX) if available on x86 hardware.

Support for AMD Family 15h Processors

The x86 backend of GCC now supports CPUs based on AMD Family 15h cores with the 64-bit x86 instruction set support. This can be enabled by using the **-march=bdver3** option.

Support for AMD Family 16h Processors

The x86 backend of GCC now supports CPUs based on AMD Family 16h cores with the 64-bit x86 instruction set support. This can be enabled by using the **-march=btver2** option.

3.1.1.2. Changes Since Red Hat Enterprise Linux 6.4 and 5.9

The following features have been added since the release of GCC included in Red Hat Enterprise Linux 6.4 and 5.9:

3.1.1.2.1. Status and Features

3.1.1.2.1.1. C++11

GCC 4.7 and later provides experimental support for building applications compliant with C++11 using the **-std=c++11** or **-std=gnu++11** command line options. However, there is no guarantee for compatibility between C++11 code compiled by different versions of the compiler. See Section 3.1.1.2.3.1, “C++ ABI” for details.

The C++ runtime library, **libstdc++**, supports a majority of the C++11 features. However, there is no or only partial support for some features such as certain properties on type traits or regular expressions. For details, see the **libstdc++** documentation, which also lists implementation-defined behavior.

Support for C++11 **exception_ptr** and **future** requires changes to the exception handling runtime in the system *libstdc++* package. These changes will be distributed through the normal Z-stream channel. Application of all Red Hat Enterprise Linux errata may be required to see correct runtime functionality when using these features.

3.1.1.2.1.2. C11

GCC 4.7 and later provides experimental support for some of the features from the C11 revision of the ISO C standard, and in addition to the previous (now deprecated) **-std=c1x** and **-std=gnu1x** command line options, gcc now accepts **-std=c11** and **-std=gnu11**. Note that since this support is experimental, it may change incompatibly in future releases.

Examples for features that are supported are Unicode strings (including the predefined macros **__STDC_UTF_16__** and **__STDC_UTF_32__**), nonreturning functions (**_Noreturn** and <stdnoreturn.h>), and alignment support (**_Alignas**, **_Alignof**, **max_align_t**, and <stdalign.h>).

3.1.1.2.1.3. Parallelism and Concurrency

GCC 4.7 and later provides improved support for programming parallel applications:

1. The GCC compilers support the OpenMP API specification for parallel programming, version 3.1. See the OpenMP website for more information about this specification.

2. The C++11 and C11 standards provide programming abstractions for multi-threaded programs. The respective standard libraries include programming abstractions for threads and thread-related features such as locks, condition variables, or futures. These new versions of the standard also define a memory model that precisely specifies the runtime behavior of a multi-threaded program, such as the guarantees provided by compilers and the constraints programmers have to pay attention to when writing multi-threaded programs.

   Note that support for the memory model is still experimental (see below for details). For more information about the status of support for C++11 and C11, see Section 3.1.1.2.1.1, “C++11” and Section 3.1.1.2.1.2, “C11” respectively.

The rest of this section describes two new GCC features in more detail. Both these features make it easier for programmers to handle concurrency (such as when multiple threads do not run truly in parallel but instead have to synchronize concurrent access to shared state), and both provide atomicity for access to memory but differ in their scope, applicability, and complexity of runtime support.

C++11 Types and GCC Built-ins for Atomic Memory Access

C++11 has support for *atomic types*. Access to memory locations of this type is atomic, and appears as one indivisible access even when other threads access the same memory location concurrently. The atomicity is limited to a single read or write access or one of the other atomic operations supported by such types (for example, two subsequent operations executed on a variable of atomic type are each atomic separately, but do not form one joint atomic operation).

An atomic type is declared as **`atomic<T>`**, where *T* is the non-atomic base type and must be trivially copyable (for example, **`atomic<int>`** is an atomic integer). GCC does not yet support any base type *T*, but only those that can be accessed atomically with the atomic instructions offered by the target architecture. This is not a significant limitation in practice, given that atomics are primarily designed to expose hardware primitives in an architecture-independent fashion; pointers and integrals that are not larger than a machine word on the target are supported as base types. Using base types that are not yet supported results in link-time errors.

The code generated for operations on atomic types, including the memory orders, implements the semantics specified in the C++11 standard. However, support for the C++11 memory model is still experimental, and for example GCC might not always preserve data-race freedom when optimizing code.

GCC also supports new built-ins for atomic memory accesses, which follow the design of the memory model and new atomic operations. The former set of synchronization built-ins (that is, those prefixed with **`__sync`**) are still supported.

Transactional Memory

Transactional Memory (TM) allows programs to declare that a piece of code is supposed to execute as a transaction, that is, virtually atomically and in isolation from other transactions. GCC's transactional memory runtime library, **`libitm`**, then ensures this atomicity guarantee when executing the compiled program. Compared to atomic memory accesses, it is a higher-level programming abstraction, because it is not limited to single memory locations, does not require special data types for the data it modifies, and because transactions can contain arbitrary code and be nested within other transactions (with some restrictions explained subsequently).

GCC implements transactions as specified in the Draft Specification for Transactional Language Constructs for C++, version 1.1. This draft does not yet specify the language constructs for C, but GCC already supports a C-compatible subset of the constructs when compiling C source code.

The main language constructs are transaction statements and expressions, and are declared by the **__transaction_atomic** or **__transaction_relaxed** keywords followed by a compound statement or expression, respectively. The following example illustrates how to increment a global variable **y** if another variable **x** has a value less than 10:

```
__transaction_atomic { if (x < 10) y++; }
```

This happens atomically even in a multi-threaded execution of the program. In particular, even though the transaction can load **x** and **y** and store to **y**, all these memory accesses are virtually executed as one indivisible step.

Note that in line with the C++11 memory model, programs that use transactions must be free of data races. Transactions are guaranteed to be virtually executed serially in a global total order that is determined by the transactional memory implementation and that is consistent with and contributes to the happens-before order enforced by the rest of the program (that is, transaction semantics are specified based on the C++11 memory model, see the draft specification linked above). Nonetheless, if a program is not data-race-free, then it has undefined behavior. For example, a thread can first initialize some data and then make it publicly accessible by code like this:

```
init(data);
__transaction_atomic { data_public = true;}  // data_public is initially
false
```

Another thread can then safely use the data, for instance:

```
__transaction_atomic { if (data_public) use(data); }
```

However, the following code has a data race and thus results in undefined behavior:

```
__transaction_atomic { temp = copy(data); if (data_public) use(temp); }
```

Here, **copy(data)** races with **init(data)** in the initializing thread, because this can be executed even if **data_public** is not true. Another example for data races is one thread accessing a variable **x** transactionally and another thread accessing it nontransactionally at potentially the same time. Note that the data can be safely reclaimed using code like this (assuming only one thread ever does this):

```
__transaction_atomic { data_public = false; }
destruct(data);
```

Here, **destruct()** does not race with potential concurrent uses of the data because after the transaction finishes, it is guaranteed that **data_public** is false and thus data is private. See the specification and the C++11 memory model for more background information about this.

Note that even if transactions are required to virtually execute in a total order, this does not mean that they execute mutually exclusive in time. Transactional memory implementations attempt to run transactions as much in parallel as possible to provide scalable performance.

There are two variants of transactions: *atomic transactions* (**__transaction_atomic**) and *relaxed transactions* (**__transaction_relaxed**). The former guarantee atomicity with regard to all other code, but allow only code that is known to not include nontransactional kinds of synchronization, such as atomic or volatile memory access. In contrast, relaxed transactions allow all code (for example calls to I/O functions), but only provide atomicity with regard to other transactions.

Therefore, atomic transactions can be nested within other atomic and relaxed transactions, but relaxed transactions can only be nested within other relaxed transactions. Furthermore, relaxed transactions are likely to be executed with less performance, but this depends on the implementation and available hardware.

GCC verifies these restrictions statically at compile time (for example, the requirements on code allowed to be called from within atomic transactions). This has implications for when transactions call functions that are defined within other compilation unit (source file) or within libraries. To enable such cross-compilation-unit calls for transactional code, the respective functions must be marked to contain code that is safe to use from within atomic transactions. Programmers can do so by adding the **`transaction_safe`** function attribute to the declarations of these functions and by including this declaration when defining the function. In turn, GCC then verifies that the code in these functions is safe for atomic transactions and generates code accordingly. If the programmer does not follow these constraints and/or steps, compile-time or link-time errors occur. Note that within a compilation unit, GCC detects automatically whether a function is safe for use within transactions, and the attributes therefore typically do not need to be added. See the draft specification linked above for further details.

GCC's transactional memory support is designed in such a way that it does not decrease the performance of programs that do not use transactions, nor the performance of nontransactional code, except due to the normal kinds of interference by concurrent threads that use the same resources such as the CPU.

Transactional memory support in GCC and **`libitm`** is still experimental, and both the ABI and API could change in the future if this is required due to the evolution of the specification of the language constructs, or due to implementation requirements. Note that when executing applications built with the **`-fgnu-tm`** command line option, it is currently a prerequisite to also have the appropriate version of the **`libitm.so.1`** shared library installed.

3.1.1.2.1.4. Architecture-specific Options

Red Hat Developer Toolset 2.0 is only available for Red Hat Enterprise Linux 5 and 6, both for the 32-bit and 64-bit Intel and AMD architectures. Consequently, the options described below are only relevant to these architectures.

Optimization for several processors is now available through the command line options described in Table 3.5, “Processor Optimization Options”.

Table 3.5. Processor Optimization Options

Option	Description
`-march=core2` and **`-mtune=core2`**	Optimization for Intel Core 2 processors.
`-march=corei7` and **`-mtune=corei7`**	Optimization for Intel Core i3, i5, and i7 processors.
`-march=corei7-avx` and **`-mtune=corei7-avx`**	Optimization for Intel Core i3, i5, and i7 processors with AVX.
`-march=core-avx-i`	Optimization for the Intel processor code-named Ivy Bridge with RDRND, FSGSBASE, and F16C.
`-march=core-avx2`	Optimization for a next-generation processor from Intel with AVX2, FMA, BMI, BMI2, and LZCNT.
`-march=bdver2` and **`-mtune=bdver2`**	Optimization for AMD Opteron processors code-named Piledriver.
`-march=btver1` and **`-mtune=btver1`**	Optimization for AMD family 14 processors code-named Bobcat.

Option	Description
-march=bdver1 and **-mtune=bdver1**	Optimization for AMD family 15h processors code-named Bulldozer.

Support for various processor-specific intrinsics and instructions is now available through the command line options described in Table 3.6, "Support for Processor-specific Intrinsics and Instructions".

Table 3.6. Support for Processor-specific Intrinsics and Instructions

Option	Description
-mavx2	Support for Intel AVX2 intrinsics, built-in functions, and code generation.
-mbmi2	Support for Intel BMI2 intrinsics, built-in functions, and code generation.
-mlzcnt	Implementation and automatic generation of **__builtin_clz*** using the **lzcnt** instruction.
-mfma	Support for Intel FMA3 intrinsics and code generation.
-mfsgsbase	Enables the generation of new segment register read/write instructions through dedicated built-ins.
-mrdrnd	Support for the Intel **rdrnd** instruction.
-mf16c	Support for two additional AVX vector conversion instructions.
-mtbm	Support for TBM (Trailing Bit Manipulation) built-in functions and code generation.
-mbmi	Support for AMD's BMI (Bit Manipulation) built-in functions and code generation.
-mcrc32	Support for **crc32** intrinsics.
-mmovbe	Enables the use of the **movbe** instruction to implement **__builtin_bswap32** and **__builtin_bswap64**.
-mxop, **-mfma4**, and **-mlwp**	Support for the XOP, FMA4, and LWP instruction sets for the AMD Orochi processors.
-mabm	Enables the use of the **popcnt** and **lzcnt** instructions on AMD processors.
-mpopcnt	Enables the use of the **popcnt** instruction on both AMD and Intel processors.

When using the x87 floating-point unit, GCC now generates code that conforms to ISO C99 in terms of handling of floating-point excess precision. This can be enabled by **-fexcess-precision=standard** and disabled by **-fexcess-precision=fast**. This feature is enabled by default when using standards conformance options such as **-std=c99**.

Vectors of type **vector long long** or **vector long** are passed and returned using the same method as other vectors with the **VSX** instruction set. Previously GCC did not adhere to the ABI for 128-bit vectors with 64-bit integer base types (see GCC PR 48857).

The **-mrecip** command line option has been added, which indicates whether the reciprocal and reciprocal square root instructions should be used.

The **-mveclibabi=mass** command line option has been added. This can be used to enable the compiler to auto-vectorize mathematical functions using the Mathematical Acceleration Subsystem library.

The **-msingle-pic-base** command line option has been added, which instructs the compiler to avoid loading the **PIC** base register in function prologues. The PIC base register must be initialized by the runtime system.

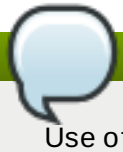
The **-mblock-move-inline-limit** command line option has been added, which enables the

user to control the maximum size of inlined **memcpy** calls and similar.

3.1.1.2.1.5. Link-time Optimization

Link-time optimization (LTO) is a compilation technique in which GCC generates an internal representation of each compiled input file in addition to the native code, and writes both to the output object file. Subsequently, when several object files are linked together, GCC uses the internal representations of the compiled code to optimize inter-procedurally across all the compilation units. This can potentially improve the performance of the generated code (for example, functions defined in one file can potentially be inlined when called in another file).

To enable LTO, the **-flto** option needs to be specified at both compile time and link time. For further details, including interoperability with linkers and parallel execution of LTO, see the documentation for **-flto** in the GCC 4.7.0 Manual. Also note that the internal representation is not a stable interface, so LTO will only apply to code generated by the same version of GCC.



Note

Use of Link-time Optimization with debug generation is not yet supported in gcc 4.7 and 4.8 and so use of the **-flto** and the **-g** options together is unsupported in Red Hat Developer Toolset.

3.1.1.2.1.6. Miscellaneous

-Ofast is now supported as a general optimization level. It operates similar to **-O3**, adds options that can yield better-optimized code, but in turn might invalidate standards compliance (for example, **-ffast-math** is enabled by **-Ofast**).

GCC can now inform users about cases in which code generation might be improved by adding attributes such as **const**, **pure**, and **noreturn** to functions declared in header files. Use the **-Wsuggest-attribute=[const|pure|noreturn]** command line option to enable this.

Assembler code can now make use of a goto feature that allows for jumps to labels in C code.

3.1.1.2.2. Language Compatibility

In this section, we describe the compatibility between the Red Hat Developer Toolset compilers and the Red Hat Enterprise Linux system compilers at the programming-language level (for example, differences in the implementation of language standards such as C99, or changes to the warnings generated by **-Wall**).

Some of the changes are a result of bug fixing, and some old behaviors have been intentionally changed in order to support new standards, or relaxed in standards-conforming ways to facilitate compilation or runtime performance. Some of these changes are not visible to the naked eye and will not cause problems when updating from older versions. However, some of these changes are visible, and can cause grief to users porting to Red Hat Developer Toolset's version of GCC. The following text attempts to identify major issues and suggests solutions.

3.1.1.2.2.1. C

Constant expressions are now handled by GCC in a way that conforms to C90 and C99. For code expressions that can be transformed into constants by the compiler but are in fact not constant expressions as defined by ISO C, this may cause warnings or errors.

Ill-formed redeclarations of library functions are no longer accepted by the compiler. In particular, a function with a signature similar to the built-in declaration of a library function (for example, **abort()** or **memcpy()**) must be declared with **extern "C"** to be considered as a redeclaration, otherwise it is ill-formed.

Duplicate Member

Consider the following **struct** declaration:

```
struct A { int *a; union { struct { int *a; }; }; };
```

Previously, this declaration used to be diagnosed just by the C++ compiler, now it is also diagnosed by the C compiler. Because of the anonymous unions and structs, there is ambiguity about what **.a** actually refers to and one of the fields therefore needs to be renamed.

3.1.1.2.2.2. C++

Header Dependency Changes

<iostream>, **<string>**, and other STL headers that previously included **<unistd.h>** as an implementation detail (to get some feature macros for **gthr*.h** purposes) no longer do so, because it was a C++ standard violation. This can result in diagnostic output similar to the following:

```
error: ‘truncate’ was not declared in this scope
error: ‘sleep’ was not declared in this scope
error: ‘pipe’ was not declared in this scope
error: there are no arguments to 'offsetof' that depend on a template
parameter, so a declaration of 'offsetof' must be available
```

To fix this, add the following line early in the source or header files that need it:

```
#include <unistd.h>
```

Many of the standard C++ library include files have been edited to no longer include **<cstddef>** to get namespace-**std**-scoped versions of **size_t** and **ptrdiff_t**. As such, C++ programs that used the macros **NULL** or **offsetof** without including **<cstddef>** will no longer compile. The diagnostic produced is similar to the following:

```
error: 'ptrdiff_t' does not name a type
error: 'size_t' has not been declared
error: 'NULL' was not declared in this scope
error: there are no arguments to 'offsetof' that depend on a template
parameter, so a declaration of 'offsetof' must be available
```

To fix this issue, add the following line:

```
#include <cstddef>
```

Name Lookup Changes

G++ no longer performs an extra unqualified lookup that it incorrectly performed in the past. Instead, it implements the two-phase lookup rules correctly, and an unqualified name used in a template must have an appropriate declaration that:

1. is either in scope at the point of the template's definition, or

2. can be found by argument-dependent lookup at the point of instantiation.

Code that incorrectly depends on a second unqualified lookup at the point of instantiation (such as finding functions declared after the template or in dependent bases) will result in compile-time errors.

In some cases, the diagnostics provided by G++ include hints how to fix the bugs. Consider the following code:

```
template<typename T>
int t(T i)
{
    return f(i);
}

int f(int i)
{
    return i;
}

int main()
{
    return t(1);
}
```

The following diagnostics output will be produced:

```
In instantiation of ‘int t(T) [with T = int]’
required from here
error: ‘f’ was not declared in this scope, and no declarations were found
by argument-dependent lookup at the point of instantiation [-fpermissive]
note: ‘int f(int)’ declared here, later in the translation unit
```

To correct the error in this example, move the declaration of function **`f()`** before the definition of template function **`t()`**. The **`-fpermissive`** compiler flag turns compile-time errors into warnings and can be used as a temporary workaround.

Uninitialized const

Consider the following declaration:

```
struct A { int a; A (); };
struct B : public A { };
const B b;
```

An attempt to compile this code now fails with the following error:

```
error: uninitialized const ‘b’ [-fpermissive]
note: ‘const struct B’ has no user-provided default constructor
```

This happens, because **B** does not have a user-provided default constructor. Either an initializer needs to be provided, or the default constructor needs to be added.

Visibility of Template Instantiations

The ELF symbol visibility of a template instantiation is now properly constrained by the visibility of its template arguments. For instance, users that instantiate standard library components like

std::vector with hidden user defined types such as **struct my_hidden_struct** can now expect hidden visibility for **std::vector<my_hidden_struct>** symbols. As a result, users that compile with the **-fvisibility=hidden** command line option should be aware of the visibility of types included from the library headers used. If the header does not explicitly control symbol visibility, types from those headers will be hidden, along with instantiations that use those types. For instance, consider the following code:

```
#include <vector>                // template std::vector has default
visibility
#include <ctime>                 // struct tm has hidden visibility
template class std::vector<tm>; // instantiation has hidden visibility
```

One approach to adjusting the visibility of a library header **<foo.h>** is to create a forwarding header on the **-I** include path consisting of the following:

```
#pragma GCC visibility push(default)
#include_next <foo.h>
#pragma GCC visibility push
```

User-defined Literal Support

When compiling C++ with the **-std={c++11,c++0x,gnu++11,gnu++0x}** command line option, GCC 4.7.0 and later, unlike older versions, supports user-defined literals, which are incompatible with some valid ISO C++03 code. In particular, white space is now needed after a string literal before something that could be a valid user defined literal. Consider the following code:

```
const char *p = "foobar"__TIME__;
```

In C++03, the **__TIME__** macro expands to some string literal and is concatenated with the other one. In C++11, **__TIME__** is not expanded and instead, operator **"" __TIME__** is being looked up, which results in a warning like:

```
error: unable to find string literal operator ‘operator"" __TIME__’
```

This applies to any string literal followed without white space by some macro. To fix this, add some white space between the string literal and the macro name.

Taking the Address of Temporary

Consider the following code:

```
struct S { S (); int i; };
void bar (S *);
void foo () { bar (&S ()); }
```

Previously, an attempt to compile this code produced a warning message, now it fails with an error. This can be fixed by adding a variable and passing the address of this variable instead of the temporary. The **-fpermissive** compiler flag turns compile-time errors into warnings and can be used as a temporary workaround.

Miscellaneous

G++ now sets the predefined macro **__cplusplus** to the correct value: **199711L** for C++98/03, and **201103L** for C++11.

G++ now properly re-uses stack space allocated for temporary objects when their lifetime ends, which can significantly lower stack consumption for some C++ functions. As a result of this, some code with undefined behavior will now break.

When an extern declaration within a function does not match a declaration in the enclosing context, G++ now properly declares the name within the namespace of the function rather than the namespace which was open just before the function definition.

G++ now implements the proposed resolution of the C++ standard's core issue 253. Default initialization is allowed if it initializes all subobjects, and code that fails to compile can be fixed by providing an initializer such as:

```
struct A { A(); };
struct B : A { int i; };
const B b = B();
```

Access control is now applied to **typedef** names used in a template, which may cause G++ to reject some ill-formed code that was accepted by earlier releases. The **-fno-access-control** option can be used as a temporary workaround until the code is corrected.

G++ now implements the C++ standard's core issue 176. Previously, G++ did not support using the injected-class-name of a template base class as a type name, and lookup of the name found the declaration of the template in the enclosing scope. Now lookup of the name finds the injected-class-name, which can be used either as a type or as a template, depending on whether or not the name is followed by a template argument list. As a result of this change, some code that was previously accepted may be ill-formed, because:

1. the injected-class-name is not accessible because it is from a private base, or
2. the injected-class-name cannot be used as an argument for a template parameter.

In either of these cases, the code can be fixed by adding a nested-name-specifier to explicitly name the template. The first can be worked around with **-fno-access-control**, the second is only rejected with **-pedantic**.

3.1.1.2.2.3. C/C++ Warnings

GCC 4.7.0 and later adds a number of new warnings that are either enabled by default, or by using the **-Wall** option. Although these warnings do not result in a compilation failure on their own, often **-Wall** is used in conjunction with **-Werror**, causing these warnings to act like errors. This section provides a list of these new or newly enabled warnings. Unless noted otherwise, these warnings apply to both C and C++.

The behavior of the **-Wall** command line option has changed and now includes the new warning flags **-Wunused-but-set-variable** and, with **-Wall -Wextra**, **-Wunused-but-set-parameter**. This may result in new warnings in code that compiled cleanly with previous versions of GCC. For example, consider the following code:

```
void fn (void)
{
    int foo;
    foo = bar ();  /* foo is never used.  */
}
```

The following diagnostic output will be produced:

```
warning: variable "foo" set but not used [-Wunused-but-set-variable]
```

To fix this issue, first see if the unused variable or parameter can be removed without changing the result or logic of the surrounding code. If not, annotate it with **__attribute__((__unused__))**. As a workaround, you can use the **-Wno-error=unused-but-set-variable** or **-Wno-error=unused-but-set-parameter** command line option.

The **-Wenum-compare** option causes GCC to report a warning when values of different enum types are being compared. Previously, this option only worked for C++ programs, but now it works for C as well. This warning is enabled by **-Wall** and may be avoided by using a type cast.

Casting integers to larger pointer types now causes GCC to display a warning by default. To disable these warnings, use the **-Wno-int-to-pointer-cast** option, which is available for both C and C++.

Conversions between NULL and non-pointer types now cause GCC to report a warning by default. Previously, these warnings were only displayed when explicitly using **-Wconversion**. To disable these warnings, use the new **-Wno-conversion-null** command line option.

GCC can now warn when a class that has virtual functions and a non-virtual destructor is destroyed by using **delete**. This is unsafe to do because the pointer might see a base class that does not have a virtual destructor. The warning is enabled by **-Wall** and by a new command line option, **-Wdelete-non-virtual-dtor**.

New **-Wc++11-compat** and **-Wc++0x-compat** options are now available. These options cause GCC to display a warning about C++ constructs whose meaning differs between ISO C++ 1998 and ISO C++ 2011 (such as identifiers in ISO C++ 1998 that are keywords in ISO C++ 2011). This warning is enabled by **-Wall** and enables the **-Wnarrowing** option.

3.1.1.2.2.4. Fortran

3.1.1.2.2.4.1. New Features

- A new compile flag **-fstack-arrays** has been added. This flag causes all local arrays to be put on stack memory, which can significantly improve the performance of some programs. Note that programs that use very large local arrays may require you to extend your runtime limits for stack memory.
- Compile time has been significantly improved. For example, the improvement may be noticeable when working with programs that use large array constructors.
- To improve code generation and diagnostics, the **-fwhole-file** compile flag is now enabled by default, and can be used with a newly supported **-fwhole-program** flag. To disable it, use the deprecated **-fno-whole-file** flag.
- A new command line option **-M** is now supported. Similarly to **gcc**, this option allows you to generate Makefile dependencies. Note that the **-cpp** option may be required as well.
- The **-finit-real=** command line option now supports **snan** as a valid value. This allows you to initialize REAL and COMPLEX variables with a signaling NaN (*not a number*), and requires you to enable trapping (for example, by using the **-ffpe-trap=** command line option). Note that compile-time optimizations may turn a signaling NaN into a quiet NaN.
- A new command line option **-fcheck=** has been added. This option accepts the following arguments:
  - The **-fcheck=bounds** option is equivalent to the **-fbounds-check** command line option.
  - The **-fcheck=array-temps** option is equivalent to the **-fcheck-array-temporaries** command line option.

- The **-fcheck=do** option checks for invalid modification of loop iteration variables.
- The **-fcheck=recursive** option checks for recursive calls to subroutines or functions that are not marked as recursive.
- The **-fcheck=pointer** option performs pointer association checks in calls, but does not handle undefined pointers nor pointers in expressions.
- The **-fcheck=all** option enables all of the above options.

- A new command line option **-fno-protect-parens** has been added. This option allows the compiler to reorder REAL and COMPLEX expressions with no regard to parentheses.
- When OpenMP's **WORKSHARE** is used, array assignments and **WHERE** will now be run in parallel.
- More Fortran 2003 and Fortran 2008 mathematical functions can now be used as initialization expressions.
- The **GCC$** compiler directive now enables support for some extended attributes such as **STDCALL**.

3.1.1.2.2.4.2. Compatibility Changes

- The **-Ofast** command line option now automatically enables the **-fno-protect-parens** and **-fstack-arrays** flags.
- Front-end optimizations can now be disabled by the **-fno-frontend-optimize** option, and selected by the **-ffrontend-optimize** option. The former is essentially only desirable if invalid Fortran source code needs to be compiled (for example, when functions—as compared to subroutines—have side-effects) or to work around compiler bugs.
- The **GFORTRAN_USE_STDERR** environment variable has been removed, and GNU Fortran now always prints error messages to standard error.
- The **-fdump-core** command line option and the **GFORTRAN_ERROR_DUMPCORE** environment variable have been removed. When encountering a serious error, GNU Fortran now always aborts the execution of the program.
- The **-fbacktrace** command line option is now enabled by default. When a fatal error occurs, GNU Fortran now attempts to print a backtrace to standard error before aborting the execution of the program. To disable this behavior, use the **-fno-backtrace** option.
- GNU Fortran no longer supports the use of the **-M** command line option to generate Makefile dependencies for the module path. To perform this operation, use the **-J** option instead.
- To significantly reduce the number of warnings, the **-Wconversion** command line option now only displays warnings when a conversion leads to information loss, and a new command line option **-Wconversion-extra** has been added to display warnings about other conversions. The **-Wconversion** option is now enabled with **-Wall**.
- A new command line option **-Wunused-dummy-argument** has been added. This option can be used to display warnings about unused dummy arguments, and is now enabled with **-Wall**. Note that the **-Wunused-variable** option previously also warned about unused dummy arguments.
- The **COMMON** default padding has been changed. Previously, the padding was added before a variable. Now it is added after a variable to increase the compatibility with other vendors, as well as to help to obtain the correct output in some cases. Note that this behavior is in contrast with the behavior of the **-falign-commons** option.

- GNU Fortran no longer links against the **libgfortranbegin** library. The **MAIN__** assembler symbol is the actual Fortran main program and is invoked by the **main** function, which is now generated and put in the same object file as **MAIN__**. Note that the **libgfortranbegin** library is still present for backward compatibility.

3.1.1.2.2.4.3. Fortran 2003 Features

- Improved but still experimental support for polymorphism between libraries and programs and for complicated inheritance patterns.
- Generic interface names which have the same name as derived types are now supported, which allows the creation of constructor functions. Note that Fortran does not support static constructor functions; only default initialization or an explicit structure-constructor initialization are available.
- Automatic (re)allocation: In intrinsic assignments to allocatable variables, the left-hand side will be automatically allocated (if unallocated) or reallocated (if the shape or type parameter is different). To avoid the small performance penalty, you can use **a(:) = ...** instead of **a = ...** for arrays and character strings — or disable the feature using **-std=f95** or **-fno-realloc-lhs**.
- Experimental support of the **ASSOCIATE** construct has been added.
- In pointer assignments it is now possible to specify the lower bounds of the pointer and, for a rank-1 or a simply contiguous data-target, to remap the bounds.
- Deferred type parameter: For scalar allocatable and pointer variables the character length can now be deferred.
- Namelist variables with allocatable attribute, pointer attribute, and with a non-constant length type parameter are now supported.
- Support has been added for procedure-pointer function results and procedure-pointer components (including PASS).
- Support has been added for allocatable scalars (experimental), **DEFERRED** type-bound procedures, and the **ERRMSG=** argument of the **ALLOCATE** and **DEALLOCATE** statements.
- The **ALLOCATE** statement now supports type-specs and the **SOURCE=** argument.
- Rounding (**ROUND=**, **RZ**, ...) for output is now supported.
- The **INT_FAST{8,16,32,64,128}_T** format for **ISO_C_BINDING** intrinsic module type parameters is now supported.
- **OPERATOR(*)** and **ASSIGNMENT(=)** are now allowed as **GENERIC** type-bound procedures (i.e. as type-bound operators).

3.1.1.2.2.4.4. Fortran 2003 Compatibility

Extensible derived types with type-bound procedure or procedure pointer with **PASS** attribute now have to use **CLASS** in line with the Fortran 2003 standard; the workaround to use **TYPE** is no longer supported.

3.1.1.2.2.4.5. Fortran 2008 Features

- A new command line option **-std=f2008ts** has been added. This option enables support for programs that conform to the Fortran 2008 standard and the draft Technical Specification (TS) 29113 on Further Interoperability of Fortran with C. For more information, see the Chart of Fortran TS 29113 Features supported by GNU Fortran.

- The **DO CONCURRENT** construct is now supported. This construct can be used to specify that individual loop iterations do not have any interdependencies.
- Full single-image support except for polymorphic coarrays has been added, and can be enabled by using the **-fcoarray=single** command line option. Additionally, GNU Fortran now provides preliminary support for multiple images via an MPI-based coarray communication library. Note that the library version is not yet usable as remote coarray access is not yet possible.
- The **STOP** and **ERROR STOP** statements have been updated to support all constant expressions.
- The **CONTIGUOUS** attribute is now supported.
- Use of **ALLOCATE** with the **MOLD** argument is now supported.
- The **STORAGE_SIZE** intrinsic inquiry function is now supported.
- The **NORM2** and **PARITY** intrinsic functions are now supported.
- The following bit intrinsics have been added:
  - the **POPCNT** and **POPPAR** bit intrinsics for counting the number of 1 bits and returning the parity;
  - the **BGE**, **BGT**, **BLE**, and **BLT** bit intrinsics for bitwise comparisons;
  - the **DSHIFTL** and **DSHIFTR** bit intrinsics for combined left and right shifts;
  - the **MASKL** and **MASKR** bit intrinsics for simple left and right justified masks;
  - the **MERGE_BITS** bit intrinsic for a bitwise merge using a mask;
  - the **SHIFTA**, **SHIFTL**, and **SHIFTR** bit intrinsics for shift operations;
  - the transformational bit intrinsics **IALL**, **IANY**, and **IPARITY**.
- The **EXECUTE_COMMAND_LINE** intrinsic subroutine is now supported.
- The **IMPURE** attribute for procedures is now supported. This allows the use of **ELEMENTAL** procedures without the restrictions of **PURE**.
- Null pointers (including **NULL()**) and unallocated variables can now be used as an actual argument to optional non-pointer, non-allocatable dummy arguments, denoting an absent argument.
- Non-pointer variables with the **TARGET** attribute can now be used as an actual argument to **POINTER** dummies with **INTENT(IN)**.
- Pointers that include procedure pointers and those in a derived type (pointer components) can now also be initialized by a target instead of only by **NULL**.
- The **EXIT** statement (with construct-name) can now be used to leave the **ASSOCIATE**, **BLOCK**, **IF**, **SELECT CASE**, and **SELECT TYPE** constructs in addition to **DO**.
- Internal procedures can now be used as actual arguments.
- The named constants **INTEGER_KINDS**, **LOGICAL_KINDS**, **REAL_KINDS**, and **CHARACTER_KINDS** of the intrinsic module **ISO_FORTRAN_ENV** have been added. These arrays contain the supported 'kind' values for the respective types.

- The **C_SIZEOF** module procedures of the **ISO_C_BINDINGS** intrinsic module and the **COMPILER_VERSION** and **COMPILER_OPTIONS** module procedures of the **ISO_FORTRAN_ENV** intrinsic module have been implemented.
- The **OPEN** statement now supports the **NEWUNIT=** option. This option returns a unique file unit and therefore prevents inadvertent use of the same unit in different parts of the program.
- Unlimited format items are now supported.
- The **INT{8,16,32}** and **REAL{32,64,128}** format for **ISO_FORTRAN_ENV** intrinsic module type parameters are now supported.
- It is now possible to use complex arguments with the **TAN**, **SINH**, **COSH**, **TANH**, **ASIN**, **ACOS**, and **ATAN** functions. Additionally, the new functions **ASINH**, **ACOSH**, and **ATANH** have been added for real and complex arguments, and **ATAN(Y,X)** now serves as an alias for **ATAN2(Y,X)**.
- The **BLOCK** construct has been implemented.

3.1.1.2.2.4.6. Fortran 2008 Compatibility

The implementation of the **ASYNCHRONOUS** attribute in GCC is now compatible with the candidate draft of *TS 29113: Technical Specification on Further Interoperability with C*.

3.1.1.2.2.4.7. Fortran 77 Compatibility

When the GNU Fortran compiler is issued with the **-fno-sign-zero** option, the **SIGN** intrinsic now behaves as if zero were always positive.

3.1.1.2.3. ABI Compatibility

This section describes compatibility between the Red Hat Developer Toolset compilers and the system compilers at the *application binary interface* (ABI) level.

3.1.1.2.3.1. C++ ABI

Because the upstream GCC community development does not guarantee C++11 ABI compatibility across major versions of GCC, the same applies to use of C++11 with Red Hat Developer Toolset. Consequently, using the **-std=c++11** option is supported in Red Hat Developer Toolset 2.0 only when all C++ objects compiled with that flag have been built using the same major version of Red Hat Developer Toolset. The mixing of objects, binaries and libraries, built by the Red Hat Enterprise Linux 5 or 6 system toolchain GCC using the **-std=c++0x** or **-std=gnu++0x** flags, with those built with the **-std=c++11** or **-std=gnu++11** flags using the GCC in Red Hat Developer Toolset is explicitly not supported.

As later major versions of Red Hat Developer Toolset may use a later major release of GCC, forward-compatibility of objects, binaries, and libraries built with the **-std=c++11** or **-std=gnu++11** options cannot be guaranteed, and so is not supported.

The default language standard setting for Red Hat Developer Toolset is C++98. Any C++98-compliant binaries or libraries built in this default mode (or explicitly with **-std=c++98**) can be freely mixed with binaries and shared libraries built by the Red Hat Enterprise Linux 5 or 6 system toolchain GCC. Red Hat recommends use of this default **-std=c++98** mode for production software development.

Important

Use of C++11 features in your application requires careful consideration of the above ABI compatibility information.

Aside from the C++11 ABI, discussed above, the Red Hat Enterprise Linux Application Compatibility Specification is unchanged for Red Hat Developer Toolset. When mixing objects built with Red Hat Developer Toolset with those built with the Red Hat Enterprise Linux v5.x/v6.x toolchain (particularly .o/.a files), the Red Hat Developer Toolset toolchain should be used for any linkage. This ensures any newer library features provided only by Red Hat Developer Toolset are resolved at link-time.

A new standard mangling for SIMD vector types has been added to avoid name clashes on systems with vectors of varying length. By default the compiler still uses the old mangling, but emits aliases with the new mangling on targets that support strong aliases. **-Wabi** will now display a warning about code that uses the old mangling.

3.1.1.2.3.2. Miscellaneous

GCC now optimizes calls to various standard C string functions such as **strlen()**, **strchr()**, **strcpy()**, **strcat()** and **stpcpy()** (as well as their respective **_FORTIFY_SOURCE** variants) by transforming them into custom, faster code. This means that there might be fewer or other calls to those functions than in the original source code. The optimization is enabled by default at **-O2** or higher optimization levels. It is disabled when using **-fno-optimize-strlen** or when optimizing for size.

When compiling for 32-bit GNU/Linux and not optimizing for size, **-fomit-frame-pointer** is now enabled by default. The prior default setting can be chosen by using the **-fno-omit-frame-pointer** command line option.

Floating-point calculations on x86 targets and in strict C99 mode are now compiled by GCC with a stricter standard conformance. This might result in those calculations executing significantly slower. It can be disabled using **-fexcess-precision=fast**.

3.1.1.2.4. Debugging Compatibility

GCC now generates DWARF debugging information that uses more or newer DWARF features than previously. GDB contained in Red Hat Developer Toolset can handle these features, but versions of GDB older than 7.0 cannot. GCC can be restricted to only generate debugging information with older DWARF features by using the **-gdwarf-2 -gstrict-dwarf** or **-gdwarf-3 -gstrict-dwarf** options (the latter are handled partially by versions of GDB older than 7.0).

Many tools such as **Valgrind**, **SystemTap**, or third-party debuggers utilize debugging information. It is suggested to use the **-gdwarf-2 -gstrict-dwarf** options with those tools.

Note

Use of Link-time Optimization with debug generation is not yet supported in gcc 4.7 and 4.8 and so use of the **-flto** and the **-g** options together is unsupported in Red Hat Developer Toolset.

3.1.1.2.5. Other Compatibility

GCC is now more strict when parsing command line options, and both **gcc** and **g++** report an error

when invalid command line options are used. In particular, when only linking and not compiling code, earlier versions of GCC ignored all options starting with **--**. For example, options accepted by the linker such as **--as-needed** and **--export-dynamic** are not accepted by **gcc** and **g++** anymore, and should now be directed to the linker using **-Wl,--as-needed** or **-Wl,--export-dynamic** if that is intended.

Because of the new link-time optimization feature (see Section 3.1.1.2.1.5, “Link-time Optimization”), support for the older intermodule optimization framework has been removed and the **-combine** command line option is not accepted anymore.

3.2. Distributed Compiling

Red Hat Enterprise Linux also supports *distributed compiling*. This involves transforming one compile job into many smaller jobs; these jobs are distributed over a cluster of machines, which speeds up build time (particularly for programs with large codebases). The **distcc** package provides this capability.

To set up distributed compiling, install the following packages:

- **distcc**
- **distcc-server**

For more information about distributed compiling, see the **man** pages for **distcc** and **distccd**. The following link also provides detailed information about the development of **distcc**:

http://code.google.com/p/distcc

3.3. Autotools

GNU Autotools is a suite of command line tools that allow developers to build applications on different systems, regardless of the installed packages or even Linux distribution. These tools aid developers in creating a **configure** script. This script runs prior to builds and creates the top-level **Makefile**s required to build the application. The **configure** script may perform tests on the current system, create additional files, or run other directives as per parameters provided by the builder.

The Autotools suite's most commonly-used tools are:

autoconf

Generates the **configure** script from an input file (**configure.ac**, for example)

automake

Creates the **Makefile** for a project on a specific system

autoscan

Generates a preliminary input file (that is, **configure.scan**), which can be edited to create a final **configure.ac** to be used by **autoconf**

All tools in the Autotools suite are part of the **Development Tools** group package. You can install this package group to install the entire Autotools suite, or use **yum** to install any tools in the suite as you wish.

3.3.1. Configuration Script

The most crucial function of Autotools is the creation of the **configure** script. This script tests systems for tools, input files, and other features it can use in order to build the project [2]. The **configure** script generates a **Makefile** which allows the **make** tool to build the project based on the system configuration.

To create the **configure** script, first create an input file. Then feed it to an Autotools utility in order to create the **configure** script. This input file is typically **configure.ac** or **Makefile.am**; the former is usually processed by **autoconf**, while the later is fed to **automake**.

If a **Makefile.am** input file is available, the **automake** utility creates a **Makefile** template (that is, **Makefile.in**), which may see information collected at configuration time. For example, the **Makefile** may have to link to a particular library *if and only if* that library is already installed. When the **configure** script runs, **automake** will use the **Makefile.in** templates to create a **Makefile**.

If a **configure.ac** file is available instead, then **autoconf** will automatically create the **configure** script based on the macros invoked by **configure.ac**. To create a preliminary **configure.ac**, use the **autoscan** utility and edit the file accordingly.

3.3.2. Autotools Documentation

Red Hat Enterprise Linux includes **man** pages for **autoconf**, **automake**, **autoscan** and most tools included in the Autotools suite. In addition, the Autotools community provides extensive documentation on **autoconf** and **automake** on the following websites:

- http://www.gnu.org/software/autoconf/manual/autoconf.html
- http://www.gnu.org/software/autoconf/manual/automake.html

The following is an online book describing the use of Autotools. Although the above online documentation is the recommended and most up to date information on Autotools, this book is a good alternative and introduction.

- http://sourceware.org/autobook/

For information on how to create Autotools input files, see:

- http://www.gnu.org/software/autoconf/manual/autoconf.html#Making-configure-Scripts
- http://www.gnu.org/software/autoconf/manual/automake.html#Invoking-Automake

The following upstream example also illustrates the use of Autotools in a simple **hello** program:

- http://www.gnu.org/software/hello/manual/hello.html

3.4. build-id Unique Identification of Binaries

Each executable or shared library built with Red Hat Enterprise Linux Server 6 or later is assigned a unique identification 160-bit SHA-1 string, generated as a checksum of selected parts of the binary. This allows two builds of the same program on the same host to always produce consistent build-ids and binary content.

Display the build-id of a binary with the following command:

```
$ eu-readelf -n usr/bin/bash
[...]
Note section [ 3] '.note.gnu.build-id' of 36 bytes at offset 0x274:
```

```
  Owner          Data size    Type
  GNU                  20    GNU_BUILD_ID
    Build ID: efdd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

Unique identificators of binaries are useful in cases such as analysing core files, documented Section 4.2.1, “Installing Debuginfo Packages for Core Files Analysis”.

3.5. Software Collections and scl-utils

With Software Collections, it is possible to build and concurrently install multiple versions of the same RPM packages on a system. Software Collections have no impact on the system versions of the packages installed by the conventional RPM package manager.

To enable support for Software Collections on a system, install the packages *scl-utils* and by typing the following at a shell prompt as **root**:

```
~]# yum install scl-utils
```

The *scl-utils* package provides the **scl** tool, which is used to enable a Software Collection and to run applications in the Software Collection environment.

General usage of the **scl** tool can be described using the following syntax:

```
scl action software_collection_1 software_collection_2 command
```

Example 3.1. Running an Application Directly

To directly run **Perl** with the **--version** option in the Software Collection named **software_collection_1**, execute the following command:

```
scl enable software_collection_1 'perl --version'
```

Example 3.2. Running a Shell with Multiple Software Collections Enabled

To run the **Bash** shell in the environment with multiple Software Collections enabled, execute the following command:

```
scl enable software_collection_1 software_collection_2 bash
```

The command above enables two Software Collections named **software_collection_1** and **software_collection_2**.

Example 3.3. Running Commands Stored in a File

To execute a number of commands, which are stored in a file, in the Software Collections environment, run the following command:

```
cat cmd | scl enable software_collection_1 -
```

> The above command executes commands, which are stored in the **cmd** file, in the environment of the Software Collection named **software_collection_1**.

For more information regarding Software Collections and **scl-utils**, see the Red Hat Software Collections 1.2 Packaging Guide.

---

[2] For information about tests that **configure** can perform, see the following link:

http://www.gnu.org/software/autoconf/manual/autoconf.html#Existing-Tests

Chapter 4. Debugging

Useful, well-written software generally goes through several different phases of application development, allowing ample opportunity for mistakes to be made. Some phases come with their own set of mechanisms to detect errors. For example, during compilation an elementary semantic analysis is often performed to make sure objects, such as variables and functions, are adequately described.

The error-checking mechanisms performed during each application development phase aims to catch simple and obvious mistakes in code. The debugging phase helps to bring more subtle errors to light that fell through the cracks during routine code inspection.

4.1. ELF Executable Binaries

Red Hat Enterprise Linux uses ELF for executable binaries, shared libraries, or debuginfo files. Within these debuginfo ELF files, the DWARF format is used. Version 3 of DWARF is used in ELF files (that is, **`gcc -g`** is equivalent to **`gcc -gdwarf-3`**). DWARF debuginfo includes:

- names of all the compiled functions and variables, including their target addresses in binaries
- source files used for compilation, including their source line numbers
- local variables location

Important

STABS is occasionally used with UNIX. STABS is an older, less capable format. Its use is discouraged by Red Hat. GCC and GDB support STABS production and consumption on a best effort basis only.

Within these ELF files, the GCC debuginfo level is also used. The default is level 2, where macro information is not present; level 3 has C/C++ macro definitions included, but the debuginfo can be very large with this setting. The command for the default **`gcc -g`** is the same as **`gcc -g2`**. To change the macro information to level three, use **`gcc -g3`**.

There are multiple levels of debuginfo available. Use the command **`readelf -WS file`** to see which sections are used in a file.

Table 4.1. debuginfo levels

Binary State	Command	Notes
Stripped	**`strip file`** or **`gcc -s -o file`**	Only the symbols required for runtime linkage with shared libraries are present. ELF section in use: **`.dynsym`**
ELF symbols	**`gcc -o file`**	Only the names of functions and variables are present, no binding to the source files and no types. ELF section in use: **`.symtab`**

Binary State	Command	Notes
DWARF debuginfo with macros	**gcc -g -o *file***	The source file names and line numbers are known, including types. ELF section in use: **.debug_***
DWARF debuginfo with macros	**gcc -g3 -o *file***	Similar to **gcc -g** but the macros are known to GDB. ELF section in use: **.debug_macro**

Note

GDB never interprets the source files, it only displays them as text. Use **gcc -g** and its variants to store the information into DWARF.

Compiling a program or library with **gcc -rdynamic** is discouraged. For specific symbols, use **gcc -Wl, --dynamic-list=...** instead. If **gcc -rdynamic** is used, the **strip** command or **-s** gcc option have no effect. This is because all ELF symbols are kept in the binary for possible runtime linkage with shared libraries.

ELF symbols can be read by the **readelf -s *file*** command.

DWARF symbols are read by the **readelf -w *file*** command.

The command **readelf -wi *file*** is a good verification of debuginfo, compiled within your program. The commands **strip *file*** or **gcc -s** are commonly accidentally executed on the output during various compilation stages of the program.

The **readelf -w *file*** command can also be used to show a special section called **.eh_frame** with a format and purpose is similar to the DWARF section **.debug_frame**. The **.eh_frame** section is used for runtime C++ exception resolution and is present even if **-g** gcc option was not used. It is kept in the primary RPM and is never present in the debuginfo RPMs.

Debuginfo RPMs contain the sections **.symtab** and **.debug_***. Neither **.eh_frame**, **.eh_frame_hdr**, nor **.dynsym** are moved or present in debuginfo RPMs as those sections are needed during program runtime.

4.2. Installing Debuginfo Packages

Red Hat Enterprise Linux also provides **-debuginfo** packages for all architecture-dependent RPMs included in the operating system. A ***packagename*-debuginfo-*version*-*release*.*architecture*.rpm** package contains detailed information about the relationship of the package source files and the final installed binary. The debuginfo packages contain both **.debug** files, which in turn contain DWARF debuginfo and the source files used for compiling the binary packages.

Note

Most of the debugger functionality is missed if attempting to debug a package without having its debuginfo equivalent installed. For example, the names of exported shared library functions will still be available, but the matching source file lines will not be without the debuginfo package installed.

Use **gcc** compilation option **-g** for your own programs. The debugging experience is better if no optimizations (gcc option **-O**, such as **-O2**) is applied with **-g**.

For Red Hat Enterprise Linux 6, the debuginfo packages are now available on a new channel on the Red Hat Network. To install the **-debuginfo** package of a package (that is, typically ***packagename*-debuginfo**), first the machine has to be subscribed to the corresponding Debuginfo channel. For example, for Red Hat Enterprise Server 6, the corresponding channel would be **Red Hat Enterprise Linux Server Debuginfo (v. 6)**.

Red Hat Enterprise Linux system packages are compiled with optimizations (gcc option **-O2**). This means that some variables will be displayed as **<optimized out>**. Stepping through code will 'jump' a little but a crash can still be analyzed. If some debugging information is missing because of the optimizations, the right variable information can be found by disassembling the code and matching it to the source manually. This is applicable only in exceptional cases and is not suitable for regular debugging.

For system packages, GDB informs the user if it is missing some debuginfo packages that limit its functionality.

```
gdb ls
[...]
Reading symbols from /usr/bin/ls...(no debugging symbols found)...done.
Missing separate debuginfos, use: debuginfo-install coreutils-8.4-
16.el6.x86_64
(gdb) q
```

If the system package to be debugged is known, use the command suggested by GDB above. It will also automatically install all the debug packages *packagename* depends on.

```
# debuginfo-install packagename
```

4.2.1. Installing Debuginfo Packages for Core Files Analysis

A core file is a representation of the memory image at the time of a process crash. For bug reporting of system program crashes, Red Hat recommends the use of the ABRT tool, explained in the *Automatic Bug Reporting Tool* chapter in the *Red Hat Deployment Guide*. If ABRT is not suitable for your purposes, the steps it automates are explained here.

If the **ulimit -c unlimited** setting is in use when a process crashes, the core file is dumped into the current directory. The core file contains only the memory areas modified by the process from the original state of disk files. In order to perform a full analysis of a crash, a core file is required to have:

- the core file itself
- the executable binary which has crashed, such as **/usr/sbin/sendmail**
- all the shared libraries loaded in the binary when it crashed

- .debug files and source files (both stored in debuginfo RPMs) for the executable and all of its loaded libraries

For a proper analysis, either the exact ***version-release.architecture*** for all the RPMs involved or the same build of your own compiled binaries is needed. At the time of the crash, the application may have already recompiled or been updated by **yum** on the disk, rendering the files inappropriate for the core file analysis.

The core file contains build-ids of all the binaries involved. For more information on build-id, see Section 3.4, “build-id Unique Identification of Binaries”. The contents of the core file can be displayed by:

```
$ eu-unstrip -n --core=./core.9814
0x400000+0x207000 2818b2009547f780a5639c904cded443e564973e@0x400284
usr/bin/sleep /usr/lib/debug/bin/sleep.debug [exe]
0x7fff26fff000+0x1000
1e2a683b7d877576970e4275d41a6aaec280795e@0x7fff26fff340 . - linux-vdso.so.1
0x35e7e00000+0x3b6000
374add1ead31ccb449779bc7ee7877de3377e5ad@0x35e7e00280 /usr/lib64/libc-
2.14.90.so /usr/lib/debug/lib64/libc-2.14.90.so.debug libc.so.6
0x35e7a00000+0x224000
3ed9e61c2b7e707ce244816335776afa2ad0307d@0x35e7a001d8 /usr/lib64/ld-
2.14.90.so /usr/lib/debug/lib64/ld-2.14.90.so.debug ld-linux-x86-64.so.2
```

The meaning of the columns in each line are:

- The in-memory address where the specific binary was mapped to (for example, **0x400000** in the first line).
- The size of the binary (for example, **+0x207000** in the first line).
- The 160-bit SHA-1 build-id of the binary (for example, **2818b2009547f780a5639c904cded443e564973e** in the first line).
- The in-memory address where the build-id bytes were stored (for example, **@0x400284** in the first line).
- The on-disk binary file, if available (for example, **usr/bin/sleep** in the first line). This was found by **eu-unstrip** for this module.
- The on-disk debuginfo file, if available (for example, **/usr/lib/debug/bin/sleep.debug**). However, best practice is to use the binary file reference instead.
- The shared library name as stored in the shared library list in the core file (for example, **libc.so.6** in the third line).

For each build-id (for example, **ab/cdef0123456789012345678901234567890123**) a symbolic link is included in its debuginfo RPM. Using the **/usr/bin/sleep** executable above as an example, the **coreutils-debuginfo** RPM contains, among other files:

```
lrwxrwxrwx 1 root root 24 Nov 29 17:07 /usr/lib/debug/.build-
id/28/18b2009547f780a5639c904cded443e564973e -> ../../../../../bin/sleep*
lrwxrwxrwx 1 root root 21 Nov 29 17:07 /usr/lib/debug/.build-
id/28/18b2009547f780a5639c904cded443e564973e.debug ->
../../bin/sleep.debug
```

In some cases (such as loading a core file), GDB does not know the name, version, or release of a ***name*-debuginfo-*version-release*.rpm** package; it only knows the build-id. In such cases,

GDB suggests a different command:

```
gdb -c ./core
[...]
Missing separate debuginfo for the main executable filename
Try: yum --disablerepo='*' --enablerepo='*debug*' install
/usr/lib/debug/.build-id/ef/dd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

The *version-release.architecture* of the binary package *packagename*-debuginfo-*version-release.architecture*.rpm must be an exact match. If it differs then GDB cannot use the debuginfo package. Even the same *version-release.architecture* from a different build leads to an incompatible debuginfo package. If GDB reports a missing debuginfo, ensure to recheck:

rpm -q *packagename* *packagename*-debuginfo

The *version-release.architecture* definitions should match.

rpm -V *packagename* *packagename*-debuginfo

This command should produce no output, except possibly modified configuration files of *packagename*, for example.

rpm -qi *packagename* *packagename*-debuginfo

The *version-release.architecture* should display matching information for Vendor, Build Date, and Build Host. For example, using a CentOS debuginfo RPM for a Red Hat Enterprise Linux RPM package will not work.

If the required build-id is known, the following command can query which RPM contains it:

```
$ repoquery --disablerepo='*' --enablerepo='*-debug*' -qf
/usr/lib/debug/.build-id/ef/dd0b5e69b0742fa5e5bad0771df4d1df2459d1
```

For example, a version of an executable which matches the core file can be installed by:

```
# yum --enablerepo='*-debug*' install $(eu-unstrip -n --core=./core.9814
| sed -e 's#^[^ ]* \(..\)\([^@ ]*\).*$#/usr/lib/debug/.build-id/\1/\2#p'
-e 's/$/.debug/')
```

Similar methods are available if the binaries are not packaged into RPMs and stored in yum repositories. It is possible to create local repositories with custom application builds by using **/usr/bin/createrepo**.

4.3. GDB

Fundamentally, like most debuggers, GDB manages the execution of compiled code in a very closely controlled environment. This environment makes possible the following fundamental mechanisms necessary to the operation of GDB:

- Inspect and modify memory within the code being debugged (for example, reading and setting variables).
- Control the execution state of the code being debugged, principally whether it's running or stopped.

- Detect the execution of particular sections of code (for example, stop running code when it reaches a specified area of interest to the programmer).
- Detect access to particular areas of memory (for example, stop running code when it accesses a specified variable).
- Execute portions of code (from an otherwise stopped program) in a controlled manner.
- Detect various programmatic asynchronous events such as signals.

The operation of these mechanisms rely mostly on information produced by a compiler. For example, to view the value of a variable, GDB has to know:

- The location of the variable in memory
- The nature of the variable

This means that displaying a double-precision floating point value requires a very different process from displaying a string of characters. For something complex like a structure, GDB has to know not only the characteristics of each individual elements in the structure, but the morphology of the structure as well.

GDB requires the following items in order to fully function:

Debug Information

Much of GDB's operations rely on a program's *debug information*. While this information generally comes from compilers, much of it is necessary only while debugging a program, that is, it is not used during the program's normal execution. For this reason, compilers do not always make that information available by default — GCC, for instance, must be explicitly instructed to provide this debugging information with the **-g** flag.

To make full use of GDB's capabilities, it is *highly advisable* to make the debug information available first to GDB. GDB can only be of *very limited* use when run against code with no available debug information.

Source Code

One of the most useful features of GDB (or any other debugger) is the ability to associate events and circumstances in program execution with their corresponding location in source code. This location normally refers to a specific line or series of lines in a source file. This, of course, would require that a program's source code be available to GDB at debug time.

4.3.1. Simple GDB

GDB literally contains dozens of commands. This section describes the most fundamental ones.

br (breakpoint)

The breakpoint command instructs GDB to halt execution upon reaching a specified point in the execution. That point can be specified a number of ways, but the most common are just as the line number in the source file, or the name of a function. Any number of breakpoints can be in effect simultaneously. This is frequently the first command issued after starting GDB.

r (run)

The **run** command starts the execution of the program. If **run** is executed with any arguments, those arguments are passed on to the executable as if the program has been started normally. Users normally issue this command after setting breakpoints.

Before an executable is started, or once the executable stops at, for example, a breakpoint, the state of many aspects of the program can be inspected. The following commands are a few of the more common ways things can be examined.

p (print)

The **print** command displays the value of the argument given, and that argument can be almost anything relevant to the program. Usually, the argument is the name of a variable of any complexity, from a simple single value to a structure. An argument can also be an expression valid in the current language, including the use of program variables and library functions, or functions defined in the program being tested.

bt (backtrace)

The **backtrace** displays the chain of function calls used up until the execution was terminated. This is useful for investigating serious bugs (such as segmentation faults) with elusive causes.

l (list)

When execution is stopped, the **list** command shows the line in the source code corresponding to where the program stopped.

The execution of a stopped program can be resumed in a number of ways. The following are the most common.

c (continue)

The **continue** command restarts the execution of the program, which will continue to execute until it encounters a breakpoint, runs into a specified or emergent condition (for example, an error), or terminates.

n (next)

Like **continue**, the **next** command also restarts execution; however, in addition to the stopping conditions implicit in the **continue** command, **next** will also halt execution at the next sequential line of code in the current source file.

s (step)

Like **next**, the **step** command also halts execution at each sequential line of code in the current source file. However, if execution is currently stopped at a source line containing a *function call*, GDB stops execution after entering the function call (rather than executing it).

fini (finish)

Like the aforementioned commands, the **finish** command resumes executions, but halts when execution returns from a function.

Finally, two essential commands:

q (quit)

This terminates the execution.

h (help)

The **help** command provides access to its extensive internal documentation. The command takes arguments: **help breakpoint** (or **h br**), for example, shows a detailed description of the **breakpoint** command. See the **help** output of each command for more detailed information.

4.3.2. Running GDB

This section will describe a basic execution of GDB, using the following simple program:

hello.c

```
#include <stdio.h>

char hello[] = { "Hello, World!" };

int
main()
{
  fprintf (stdout, "%s\n", hello);
  return (0);
}
```

The following procedure illustrates the debugging process in its most basic form.

Procedure 4.1. Debugging a 'Hello World' Program

1. Compile hello.c into an executable with the debug flag set, as in:

   gcc -g -o hello hello.c

   Ensure that the resulting binary **hello** is in the same directory as **hello.c**.

2. Run **gdb** on the **hello** binary, that is, **gdb hello**.

3. After several introductory comments, **gdb** will display the default GDB prompt:

   ```
   (gdb)
   ```

4. The variable **hello** is global, so it can be seen even before the **main** procedure starts:

   ```
   gdb) p hello
   $1 = "Hello, World!"
   (gdb) p hello[0]
   $2 = 72 'H'
   (gdb) p *hello
   $3 = 72 'H'
   (gdb)
   ```

   Note that the **print** targets **hello[0]** and ***hello** require the evaluation of an expression, as does, for example, ***(hello + 1)**:

   ```
   (gdb) p *(hello + 1)
   $4 = 101 'e'
   ```

5. Next, list the source:

```
(gdb) l
1       #include <stdio.h>
2
3       char hello[] = { "Hello, World!" };
4
5       int
6       main()
7       {
8         fprintf (stdout, "%s\n", hello);
9         return (0);
10      }
```

The **list** reveals that the **fprintf** call is on line 8. Apply a breakpoint on that line and resume the code:

```
(gdb) br 8
Breakpoint 1 at 0x80483ed: file hello.c, line 8.
(gdb) r
Starting program: /home/moller/tinkering/gdb-manual/hello

Breakpoint 1, main () at hello.c:8
8         fprintf (stdout, "%s\n", hello);
```

6. Finally, use the **next** command to step past the **fprintf** call, executing it:

```
(gdb) n
Hello, World!
9         return (0);
```

The following sections describe more complex applications of GDB.

4.3.3. Conditional Breakpoints

In many real-world cases, a program may perform its task well during the first few thousand times; it may then start crashing or encountering errors during its eight thousandth iteration of the task. Debugging programs like this can be difficult, as it is hard to imagine a programmer with the patience to issue a **continue** command thousands of times just to get to the iteration that crashed.

Situations like this are common in real life, which is why GDB allows programmers to attach conditions to a breakpoint. For example, consider the following program:

simple.c

```
#include <stdio.h>

main()
{
  int i;

  for (i = 0;; i++) {
fprintf (stdout, "i = %d\n", i);
  }
}
```

To set a conditional breakpoint at the GDB prompt:

```
(gdb) br 8 if i == 8936
Breakpoint 1 at 0x80483f5: file iterations.c, line 8.
(gdb) r
```

With this condition, the program execution will eventually stop with the following output:

```
i = 8931
i = 8932
i = 8933
i = 8934
i = 8935

Breakpoint 1, main () at iterations.c:8
8           fprintf (stdout, "i = %d\n", i);
```

Inspect the breakpoint information (using **`info br`**) to review the breakpoint status:

```
(gdb) info br
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483f5 in main at iterations.c:8
        stop only if i == 8936
        breakpoint already hit 1 time
```

4.3.4. Forked Execution

Among the more challenging bugs confronting programmers is where one program (the *parent*) makes an independent copy of itself (a *fork*). That fork then creates a *child* process which, in turn, fails. Debugging the parent process may or may not be useful. Often the only way to get to the bug may be by debugging the child process, but this is not always possible.

The **`set follow-fork-mode`** feature is used to overcome this barrier allowing programmers to follow a a child process instead of the parent process.

`set follow-fork-mode parent`

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

`set follow-fork-mode child`

The new process is debugged after a fork. The parent process runs unimpeded.

`show follow-fork-mode`

Display the current debugger response to a fork call.

Use the **`set detach-on-fork`** command to debug both the parent and the child processes after a fork, or retain debugger control over them both.

`set detach-on-fork on`

The child process (or parent process, depending on the value of **`follow-fork-mode`**) will be detached and allowed to run independently. This is the default.

`set detach-on-fork off`

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of **follow-fork-mode**) is debugged as usual, while the other is suspended.

show detach-on-fork

Show whether **detach-on-fork** mode is on or off.

Consider the following program:

fork.c

```
#include <unistd.h>

int main()
{
  pid_t  pid;
  const char *name;

  pid = fork();
  if (pid == 0)
    {
      name = "I am the child";
    }
  else
    {
      name = "I am the parent";
    }
  return 0;
}
```

This program, compiled with the command **gcc -g fork.c -o fork -lpthread** and examined under GDB will show:

```
gdb ./fork
[...]
(gdb) break main
Breakpoint 1 at 0x4005dc: file fork.c, line 8.
(gdb) run
[...]
Breakpoint 1, main () at fork.c:8
8   pid = fork();
(gdb) next
Detaching after fork from child process 3840.
9   if (pid == 0)
(gdb) next
15        name = "I am the parent";
(gdb) next
17   return 0;
(gdb) print name
$1 = 0x400717 "I am the parent"
```

GDB followed the parent process and allowed the child process (process 3840) to continue execution.

The following is the same test using **set follow-fork-mode child**.

```
(gdb) set follow-fork-mode child
(gdb) break main
Breakpoint 1 at 0x4005dc: file fork.c, line 8.
(gdb) run
[...]
Breakpoint 1, main () at fork.c:8
8   pid = fork();
(gdb) next
[New process 3875]
[Thread debugging using libthread_db enabled]
[Switching to Thread 0x7ffff7fd5720 (LWP 3875)]
9   if (pid == 0)
(gdb) next
11        name = "I am the child";
(gdb) next
17   return 0;
(gdb) print name
$2 = 0x400708 "I am the child"
(gdb)
```

GDB switched to the child process here.

This can be permanent by adding the setting to the appropriate **`.gdbinit`**.

For example, if **`set follow-fork-mode ask`** is added to **`~/.gdbinit`**, then ask mode becomes the default mode.

4.3.5. Debugging Individual Threads

GDB has the ability to debug individual threads, and to manipulate and examine them independently. This functionality is not enabled by default. To do so use **`set non-stop on`** and **`set target-async on`**. These can be added to **`.gdbinit`**. Once that functionality is turned on, GDB is ready to conduct thread debugging.

For example, the following program creates two threads. These two threads, along with the original thread executing main makes a total of three threads.

three-threads.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_t thread;

void* thread3 (void* d)
{
  int count3 = 0;

  while(count3 < 1000){
    sleep(10);
    printf("Thread 3: %d\n", count3++);
  }
  return NULL;
}
```

```
void* thread2 (void* d)
{
  int count2 = 0;

  while(count2 < 1000){
    printf("Thread 2: %d\n", count2++);
  }
  return NULL;
}

int main (){

  pthread_create (&thread, NULL, thread2, NULL);
  pthread_create (&thread, NULL, thread3, NULL);

  //Thread 1
  int count1 = 0;

  while(count1 < 1000){
    printf("Thread 1: %d\n", count1++);
  }

  pthread_join(thread,NULL);
  return 0;
}
```

Compile this program in order to examine it under GDB.

```
gcc -g three-threads.c -o three-threads  -lpthread
gdb ./three-threads
```

First set breakpoints on all thread functions; thread1, thread2, and main.

```
(gdb) break thread3
Breakpoint 1 at 0x4006c0: file three-threads.c, line 9.
(gdb) break thread2
Breakpoint 2 at 0x40070c: file three-threads.c, line 20.
(gdb) break main
Breakpoint 3 at 0x40074a: file three-threads.c, line 30.
```

Then run the program.

```
(gdb) run
[...]
Breakpoint 3, main () at three-threads.c:30
30   pthread_create (&thread, NULL, thread2, NULL);
[...]
(gdb) info threads
* 1 Thread 0x7ffff7fd5720 (LWP 4620)  main () at three-threads.c:30
(gdb)
```

Note that the command **`info threads`** provides a summary of the program's threads and some details about their current state. In this case there is only one thread that has been created so far.

Continue execution some more.

```
(gdb) next
[New Thread 0x7ffff7fd3710 (LWP 4687)]
31   pthread_create (&thread, NULL, thread3, NULL);
(gdb)
Breakpoint 2, thread2 (d=0x0) at three-threads.c:20
20   int count2 = 0;
next
[New Thread 0x7ffff75d2710 (LWP 4688)]
34   int count1 = 0;
(gdb)
Breakpoint 1, thread3 (d=0x0) at three-threads.c:9
9   int count3 = 0;
info threads
  3 Thread 0x7ffff75d2710 (LWP 4688)  thread3 (d=0x0) at three-
threads.c:9
  2 Thread 0x7ffff7fd3710 (LWP 4687)  thread2 (d=0x0) at three-
threads.c:20
* 1 Thread 0x7ffff7fd5720 (LWP 4620)  main () at three-threads.c:34
```

Here, two more threads are created. The star indicates the thread currently under focus. Also, the newly created threads have hit the breakpoint set for them in their initialization functions. Namely, thread2() and thread3().

To begin real thread debugging, use the **`thread <thread number>`** command to switch the focus to another thread.

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff7fd3710 (LWP 4687))]#0  thread2
(d=0x0)
    at three-threads.c:20
20   int count2 = 0;
(gdb) list
15   return NULL;
16 }
17
18 void* thread2 (void* d)
19 {
20   int count2 = 0;
21
22   while(count2 < 1000){
23     printf("Thread 2: %d\n", count2++);
24   }
```

Thread 2 stopped at line 20 in its function thread2().

```
(gdb) next
22   while(count2 < 1000){
(gdb) print count2
$1 = 0
(gdb) next
23     printf("Thread 2: %d\n", count2++);
(gdb) next
```

```
Thread 2: 0
22   while(count2 < 1000){
(gdb) next
23     printf("Thread 2: %d\n", count2++);
(gdb) print count2
$2 = 1
(gdb) info threads
  3 Thread 0x7ffff75d2710 (LWP 4688)  thread3 (d=0x0) at three-
threads.c:9
* 2 Thread 0x7ffff7fd3710 (LWP 4687)  thread2 (d=0x0) at three-
threads.c:23
  1 Thread 0x7ffff7fd5720 (LWP 4620)  main () at three-threads.c:34
(gdb)
```

Above, a few lines of thread2 printed the counter count2 and left thread 2 at line 23 as is seen by the output of 'info threads'.

Now thread3.

```
(gdb) thread 3
[Switching to thread 3 (Thread 0x7ffff75d2710 (LWP 4688))]#0  thread3
(d=0x0)
    at three-threads.c:9
9   int count3 = 0;
(gdb) list
4
5 pthread_t thread;
6
7 void* thread3 (void* d)
8 {
9   int count3 = 0;
10
11   while(count3 < 1000){
12     sleep(10);
13     printf("Thread 3: %d\n", count3++);
(gdb)
```

Thread three is a little different in that it has a sleep statement and executes slowly. Think of it as a representation of an uninteresting IO thread. Because this thread is uninteresting, continue its execution uninterrupted, using the **`continue`**.

```
(gdb) continue &
(gdb) Thread 3: 0
Thread 3: 1
Thread 3: 2
Thread 3: 3
```

Take note of the *&* at the end of the **`continue`**. This allows the GDB prompt to return so other commands can be executed. Using the **`interrupt`**, execution can be stopped should thread 3 become interesting again.

```
(gdb) interrupt
[Thread 0x7ffff75d2710 (LWP 4688)] #3 stopped.
0x000000343f4a6a6d in nanosleep () at ../sysdeps/unix/syscall-
template.S:82
```

```
82 T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
```

It is also possible to go back to the original main thread and examine it some more.

```
(gdb) thread 1
[Switching to thread 1 (Thread 0x7ffff7fd5720 (LWP 4620))]#0  main ()
    at three-threads.c:34
34   int count1 = 0;
(gdb) next
36   while(count1 < 1000){
(gdb) next
37     printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 0
36   while(count1 < 1000){
(gdb) next
37     printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 1
36   while(count1 < 1000){
(gdb) next
37     printf("Thread 1: %d\n", count1++);
(gdb) next
Thread 1: 2
36   while(count1 < 1000){
(gdb) print count1
$3 = 3
(gdb) info threads
  3 Thread 0x7ffff75d2710 (LWP 4688)  0x000000343f4a6a6d in nanosleep ()
    at ../sysdeps/unix/syscall-template.S:82
  2 Thread 0x7ffff7fd3710 (LWP 4687)  thread2 (d=0x0) at three-
threads.c:23
* 1 Thread 0x7ffff7fd5720 (LWP 4620)  main () at three-threads.c:36
(gdb)
```

As can be seen from the output of info threads, the other threads are where they were left, unaffected by the debugging of thread 1.

4.3.6. Alternative User Interfaces for GDB

GDB uses the command line as its default interface. However, it also has an API called *machine interface* (MI). MI allows IDE developers to create other user interfaces to GDB.

Some examples of these interfaces are:

Eclipse (CDT)

A graphical debugger interface integrated with the Eclipse development environment. More information can be found at the *Eclipse website*.

Nemiver

A graphical debugger interface which is well suited to the GNOME Desktop Environment. More information can be found at the *Nemiver website*

Emacs

A GDB interface which is integrated with the emacs. More information can be found at the *Emacs website*

4.3.7. GDB Documentation

For more detailed information about GDB, see the GDB manual:

http://sources.redhat.com/gdb/current/onlinedocs/gdb.html

Also, the commands **`info gdb`** and **`man gdb`** will provide more concise information that is up to date with the installed version of gdb.

4.4. Variable Tracking at Assignments

Variable Tracking at Assignments (VTA) is a new infrastructure included in GCC used to improve variable tracking during optimizations. This allows GCC to produce more precise, meaningful, and useful debugging information for GDB, SystemTap, and other debugging tools.

When GCC compiles code with optimizations enabled, variables are renamed, moved around, or even removed altogether. As such, optimized compiling can cause a debugger to report that some variables have been <optimized out>. With VTA enabled, optimized code is internally annotated to ensure that optimization passes to transparently keep track of each variable's value, regardless of whether the variable is moved or removed. The effect of this is more parameter and variable values available, even for the optimized (**`gcc -O2 -g`** built) code. It also displays the <optimized out> message less.

VTA's benefits are more pronounced when debugging applications with inlined functions. Without VTA, optimization could completely remove some arguments of an inlined function, preventing the debugger from inspecting its value. With VTA, optimization will still happen, and appropriate debugging information will be generated for any missing arguments.

VTA is enabled by default when compiling code with optimizations and debugging information enabled (that is, **`gcc -O -g`** or, more commonly, **`gcc -O2 -g`**). To disable VTA during such builds, add the **`-fno-var-tracking-assignments`**. In addition, the VTA infrastructure includes the new **`gcc`** option **`-fcompare-debug`**. This option tests code compiled by GCC with debug information and without debug information: the test passes if the two binaries are identical. This test ensures that executable code is not affected by any debugging options, which further ensures that there are no hidden bugs in the debug code. Note that **`-fcompare-debug`** adds significant cost in compilation time. See **`man gcc`** for details about this option.

For more information about the infrastructure and development of VTA, see *A Plan to Fix Local Variable Debug Information in GCC*, available at the following link:

http://gcc.gnu.org/wiki/Var_Tracking_Assignments

A slide deck version of this whitepaper is also available at http://people.redhat.com/aoliva/papers/vta/slides.pdf.

4.5. Python Pretty-Printers

The GDB command **`print`** outputs comprehensive debugging information for a target application. GDB aims to provide as much debugging data as it can to users; however, this means that for highly complex programs the amount of data can become very cryptic.

In addition, GDB does not provide any tools that help decipher GDB **`print`** output. GDB does not even empower users to easily create tools that can help decipher program data. This makes the practice of reading and understanding debugging data quite arcane, particularly for large, complex projects.

For most developers, the only way to customize GDB **`print`** output (and make it more meaningful) is to revise and recompile GDB. However, very few developers can actually do this. Further, this practice will not scale well, particularly if the developer must also debug other programs that are heterogeneous and contain equally complex debugging data.

To address this, the Red Hat Enterprise Linux version of GDB is now compatible with Python *pretty-printers*. This allows the retrieval of more meaningful debugging data by leaving the introspection, printing, and formatting logic to a *third-party* Python script.

Compatibility with Python pretty-printers gives you the chance to truly customize GDB output as you see fit. This makes GDB a more viable debugging solution to a wider range of projects, since you now have the flexibility to *adapt* GDB output as required, and with greater ease. Further, developers with intimate knowledge of a project and a specific programming language are best qualified in deciding what kind of output is meaningful, allowing them to improve the usefulness of that output.

The Python pretty-printers implementation allows users to automatically inspect, format, and print program data according to specification. These specifications are written as rules implemented via Python scripts. This offers the following benefits:

Safe

To pass program data to a set of registered Python pretty-printers, the GDB development team added *hooks* to the GDB printing code. These hooks were implemented with safety in mind: the built-in GDB printing code is still intact, allowing it to serve as a default fallback printing logic. As such, if no specialized printers are available, GDB will still print debugging data the way it always did. This ensures that GDB is backwards-compatible; users who do not require pretty-printers can still continue using GDB.

Highly Customizable

This new "Python-scripted" approach allows users to distill as much knowledge as required into specific printers. As such, a project can have an entire library of printer scripts that parses program data in a unique manner specific to its user's requirements. There is no limit to the number of printers a user can build for a specific project; what's more, being able to customize debugging data script by script offers users an easier way to re-use and re-purpose printer scripts — or even a whole library of them.

Easy to Learn

The best part about this approach is its lower barrier to entry. Python scripting is comparatively easy to learn and has a large library of free documentation available online. In addition, most programmers already have basic to intermediate experience in Python scripting, or in scripting in general.

Here is a small example of a pretty printer. Consider the following C++ program:

fruit.cc

```
enum Fruits {Orange, Apple, Banana};

class Fruit
{
```

```
  int fruit;

 public:
  Fruit (int f)
    {
      fruit = f;
    }
};

int main()
{
  Fruit myFruit(Apple);
  return 0;              // line 17
}
```

This is compiled with the command **`g++ -g fruit.cc -o fruit`**. Now, examine this program with GDB.

```
gdb ./fruit
[...]
(gdb) break 17
Breakpoint 1 at 0x40056d: file fruit.cc, line 17.
(gdb) run

Breakpoint 1, main () at fruit.cc:17
17   return 0;              // line 17
(gdb) print myFruit
$1 = {fruit = 1}
```

The output of **`{fruit = 1}`** is correct because that is the internal representation of 'fruit' in the data structure 'Fruit'. However, this is not easily read by humans as it is difficult to tell which fruit the integer 1 represents.

To solve this problem, write the following pretty printer:

```
fruit.py

class FruitPrinter:
    def __init__(self, val):
        self.val = val

    def to_string (self):
        fruit = self.val['fruit']

        if (fruit == 0):
            name = "Orange"
        elif (fruit == 1):
            name = "Apple"
        elif (fruit == 2):
            name = "Banana"
        else:
            name = "unknown"
        return "Our fruit is " + name

def lookup_type (val):
    if str(val.type) == 'Fruit':
```

```
        return FruitPrinter(val)
    return None

gdb.pretty_printers.append (lookup_type)
```

Examine this printer from the bottom up.

The line **gdb.pretty_printers.append (lookup_type)** adds the function **lookup_type** to GDB's list of printer lookup functions.

The function **lookup_type** is responsible for examining the type of object to be printed, and returning an appropriate pretty printer. The object is passed by GDB in the parameter ***val***. **val.type** is an attribute which represents the type of the pretty printer.

FruitPrinter is where the actual work is done. More specifically in the **to_string** function of that Class. In this function, the integer **fruit** is retrieved using the python dictionary syntax **self.val['fruit']**. Then the name is determined using that value. The string returned by this function is the string that will be printed to the user.

After creating **fruit.py**, it must then be loaded into GDB with the following command:

```
(gdb) python execfile("fruit.py")
```

The *GDB and Python Pretty-Printers* whitepaper provides more details on this feature. This whitepaper also includes details and examples on how to write your own Python pretty-printer as well as how to import it into GDB. See the following link for more information:

http://sourceware.org/gdb/onlinedocs/gdb/Pretty-Printing.html

4.6. ftrace

The **ftrace** framework provides users with several tracing capabilities, accessible through an interface much simpler than SystemTap's. This framework uses a set of virtual files in the **debugfs** file system; these files enable specific tracers. The **ftrace** function tracer outputs each function called in the kernel in real time; other tracers within the **ftrace** framework can also be used to analyze wakeup latency, task switches, kernel events, and the like.

You can also add new tracers for **ftrace**, making it a flexible solution for analyzing kernel events. The **ftrace** framework is useful for debugging or analyzing latencies and performance issues that take place outside of user-space. Unlike other profilers documented in this guide, **ftrace** is a built-in feature of the kernel.

4.6.1. Using ftrace

The Red Hat Enterprise Linux kernels have been configured with the **CONFIG_FTRACE=y** option. This option provides the interfaces required by **ftrace**. To use **ftrace**, mount the **debugfs** file system as follows:

```
mount -t debugfs nodev /sys/kernel/debug
```

All the **ftrace** utilities are located in **/sys/kernel/debug/tracing/**. View the **/sys/kernel/debug/tracing/available_tracers** file to find out what tracers are available for your kernel:

```
cat /sys/kernel/debug/tracing/available_tracers
```

```
power wakeup irqsoff function sysprof sched_switch initcall nop
```

To use a specific tracer, write it to **/sys/kernel/debug/tracing/current_tracer**. For example, **wakeup** traces and records the maximum time it takes for the highest-priority task to be scheduled after the task wakes up. To use it:

```
echo wakeup > /sys/kernel/debug/tracing/current_tracer
```

To start or stop tracing, write to **/sys/kernel/debug/tracing/tracing_on**, as in:

echo 1 > /sys/kernel/debug/tracing/tracing_on (enables tracing)

echo 0 > /sys/kernel/debug/tracing/tracing_on (disables tracing)

The results of the trace can be viewed from the following files:

/sys/kernel/debug/tracing/trace

This file contains human-readable trace output.

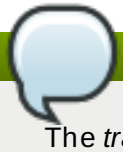
/sys/kernel/debug/tracing/trace_pipe

This file contains the same output as **/sys/kernel/debug/tracing/trace**, but is meant to be piped into a command. Unlike **/sys/kernel/debug/tracing/trace**, reading from this file consumes its output.

4.6.2. ftrace Documentation

The **ftrace** framework is fully documented in the following files:

- *ftrace - Function Tracer*: **file:///usr/share/doc/kernel-doc-*version*/Documentation/trace/ftrace.txt**
- *function tracer guts*: **file:///usr/share/doc/kernel-doc-*version*/Documentation/trace/ftrace-design.txt**



Note

The *trace-cmd* package provides a tool of the same name that can be a useful alternative to **ftrace**. Further information is available on the **trace-cmd** man page.

Chapter 5. Monitoring Performance

Developers profile programs to focus attention on the areas of the program that have the largest impact on performance. The types of data collected include what section of the program consumes the most processor time, and where memory is allocated. Profiling collects data from the actual program execution. Thus, the quality of the data collect is influenced by the actual tasks being performed by the program. The tasks performed during profiling should be representative of actual use; this ensures that problems arising from realistic use of the program are addressed during development.

Red Hat Enterprise Linux includes a number of different tools (**Valgrind**, **OProfile**, **perf**, and **SystemTap**) to collect profiling data. Each tool is suitable for performing specific types of profile runs, as described in the following sections.

5.1. Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools that can be used to profile applications in detail. **Valgrind** tools are generally used to automatically detect many memory management and threading problems. The **Valgrind** suite also includes tools that allow the building of new profiling tools as required.

Valgrind provides instrumentation for user-space binaries to check for errors, such as the use of uninitialized memory, improper allocation/freeing of memory, and improper arguments for systemcalls. Its profiling tools can be used by normal users on most binaries; however, compared to other profilers, **Valgrind** profile runs are significantly slower. To profile a binary, **Valgrind** rewrites its executable and instruments the rewritten binary. **Valgrind**'s tools are most useful for looking for memory-related issues in user-space programs; it is not suitable for debugging time-specific issues or kernel-space instrumentation/debugging.

Previously, **Valgrind** did not support IBM System z architecture. However, as of 6.1, this support has been added, meaning **Valgrind** now supports all hardware architectures that are supported by Red Hat Enterprise Linux 6.x.

5.1.1. Valgrind Tools

The **Valgrind** suite is composed of the following tools:

memcheck

This tool detects memory management problems in programs by checking all reads from and writes to memory and intercepting all system calls to **malloc**, **new**, **free**, and **delete**. **memcheck** is perhaps the most used **Valgrind** tool, as memory management problems can be difficult to detect using other means. Such problems often remain undetected for long periods, eventually causing crashes that are difficult to diagnose.

cachegrind

cachegrind is a cache profiler that accurately pinpoints sources of cache misses in code by performing a detailed simulation of the I1, D1 and L2 caches in the CPU. It shows the number of cache misses, memory references, and instructions accruing to each line of source code; **cachegrind** also provides per-function, per-module, and whole-program summaries, and can even show counts for each individual machine instructions.

callgrind

Like **cachegrind**, **callgrind** can model cache behavior. However, the main purpose of **callgrind** is to record callgraphs data for the executed code.

massif

massif is a heap profiler; it measures how much heap memory a program uses, providing information on heap blocks, heap administration overheads, and stack sizes. Heap profilers are useful in finding ways to reduce heap memory usage. On systems that use virtual memory, programs with optimized heap memory usage are less likely to run out of memory, and may be faster as they require less paging.

helgrind

In programs that use the POSIX pthreads threading primitives, **helgrind** detects synchronization errors. Such errors are:

- Misuses of the POSIX pthreads API
- Potential deadlocks arising from lock ordering problems
- Data races (that is, accessing memory without adequate locking)

Valgrind also allows you to develop your own profiling tools. In line with this, **Valgrind** includes the **lackey** tool, which is a sample that can be used as a template for generating your own tools.

5.1.2. Using Valgrind

The **valgrind** package and its dependencies install all the necessary tools for performing a **Valgrind** profile run. To profile a program with **Valgrind**, use:

```
valgrind --tool=toolname program
```

See Section 5.1.1, “Valgrind Tools” for a list of arguments for ***toolname***. In addition to the suite of **Valgrind** tools, **none** is also a valid argument for ***toolname***; this argument allows you to run a program under **Valgrind** without performing any profiling. This is useful for debugging or benchmarking **Valgrind** itself.

You can also instruct **Valgrind** to send all of its information to a specific file. To do so, use the option **--log-file=*filename***. For example, to check the memory usage of the executable file **hello** and send profile information to **output**, use:

```
valgrind --tool=memcheck --log-file=output hello
```

See Section 5.1.3, “Valgrind Documentation” for more information on **Valgrind**, along with other available documentation on the **Valgrind** suite of tools.

5.1.3. Valgrind Documentation

For more extensive information on **Valgrind**, see **man valgrind**. Red Hat Enterprise Linux 6 also provides a comprehensive *Valgrind Documentation* book, available as PDF and HTML in:

- **file:///usr/share/doc/valgrind-*version*/valgrind_manual.pdf**
- **file:///usr/share/doc/valgrind-*version*/html/index.html**

The *Valgrind Integration User Guide* in the Eclipse **Help Contents** also provides detailed information on the setup and usage of the **Valgrind** plug-in for Eclipse. This guide is provided by the **eclipse-valgrind** package.

5.2. OProfile

OProfile is a system-wide Linux profiler, capable of running at low overhead. It consists of a kernel driver and a daemon for collecting raw sample data, along with a suite of tools for parsing that data into meaningful information. OProfile is generally used by developers to determine which sections of code consume the most amount of CPU time, and why.

During a profile run, OProfile uses the processor's performance monitoring hardware. **Valgrind** rewrites the binary of an application, and in turn instruments it. OProfile, on the other hand,profiles a running application as-is. It sets up the performance monitoring hardware to take a sample every *x* number of events (for example, cache misses or branch instructions). Each sample also contains information on where it occurred in the program.

OProfile's profiling methods consume less resources than **Valgrind**. However, OProfile requires root privileges. OProfile is useful for finding "hot-spots" in code, and looking for their causes (for example, poor cache performance, branch mispredictions).

Using OProfile involves starting the OProfile daemon (**oprofiled**), running the program to be profiled, collecting the system profile data, and parsing it into a more understandable format. OProfile provides several tools for every step of this process.

5.2.1. OProfile Tools

The most useful OProfile commands include the following:

operf

New in Red Hat Enterprise Linux 7, **operf** uses the Linux Performance Events subsystem, and so can completely replace use of the **opcontrol** daemon. See the *Red Hat Enterprise Linux 7 System Administrator's Guide* for further details.

opcontrol

This tool is used to start/stop the OProfile daemon and configure a profile session.

opreport

The **opreport** command outputs binary image summaries, or per-symbol data, from OProfile profiling sessions.

opannotate

The **opannotate** command outputs annotated source and/or assembly from the profile data of an OProfile session.

oparchive

The **oparchive** command generates a directory populated with executable, debug, and OProfile sample files. This directory can be moved to another machine (via **tar**), where it can be analyzed offline.

opgprof

Like **opreport**, the **opgprof** command outputs profile data for a given binary image from an OProfile session. The output of **opgprof** is in **gprof** format.

For a complete list of OProfile commands, see **man oprofile**. For detailed information on each OProfile command, see its corresponding **man** page. See Section 5.2.4, “OProfile Documentation” for other available documentation on OProfile.

5.2.2. Using OProfile

The **oprofile** package and its dependencies install all the necessary utilities for executing OProfile. To instruct OProfile to profile all the applications running on the system and to group the samples for the shared libraries with the application using the library, run the following command:

```
# opcontrol --no-vmlinux --separate=library --start
```

You can also start the OProfile daemon without collecting system data. To do so, use the option **--start-daemon**. The **--stop** option halts data collection, while **--shutdown** terminates the OProfile daemon.

Use **opreport**, **opannotate**, or **opgprof** to display the collected profiling data. By default, the data collected by the OProfile daemon is stored in **/var/lib/oprofile/samples/**.

OProfile conflict with Performance Counters for Linux (PCL) tools

Both OProfile and Performance Counters for Linux (PCL) use the same hardware Performance Monitoring Unit (PMU). If the PCL or the NMI watchdog timer are using the hardware PMU, a message like the following occurs when starting OProfile:

```
# opcontrol --start
Using default event: CPU_CLK_UNHALTED:100000:0:1:1
Error: counter 0 not available nmi_watchdog using this resource ? Try:
opcontrol --deinit
echo 0 > /proc/sys/kernel/nmi_watchdog
```

Stop any **perf** commands running on the system, then turn off the NMI watchdog and reload the OProfile kernel driver with the following commands:

```
# opcontrol --deinit
```

```
# echo 0 > /proc/sys/kernel/nmi_watchdog
```

5.2.3. OProfile in Red Hat Enterprise Linux 7

OProfile 0.9.8 has been released for Red Hat Enterprise Linux 7. This is an alpha version but has proven stable for many users. With the 0.9.8 release OProfile can now also be used to profile specific individual processes.

5.2.3.1. New Features

A new **operf** program is now available that allows non-root users to profile single processes. This can also be used for system-wide profiling, but in this case root authority is required. This capability requires a kernel version of 2.6.31 or greater.

OProfile also supports a number of new processors:

- Tilera tile64

- Tilera tilepro
- Tilera tile-gx
- IBM System z10
- IBM System z196
- Intel Ivy Bridge
- ARMv7 Cortex-A5
- ARMv7 Cortex-A15
- ARMv7 Cortex-A7

5.2.3.2. Incompatibilities with the Previous Release

OProfile 0.9.8 has some incompatibilities with the previous release:

- Support for pre-2.6 kernels has been removed.
- With the removal of pre-2.6 support, the **--with-kernel-support** configure option is no longer needed nor valid.
- Sample header mtime field has changed to u64.
- The **configure.in** file has been renamed to **configure.ac**. This should a transparent change.

5.2.3.3. Known Problems and Limitiations

OProfile 0.9.8 has a few known problems and limitations. These are:

- AMD Instruction Based Sampling (IBS) is not currently supported with the new operf program. Use the **legacy** opcontrol commands for IBS profiling.
- When using **operf** to profile multiple events, the absolute number of events recorded will usually be substantially fewer than expected. This is due to a bug in the Linux kernel's Performance Events Subsystem that was fixed between Linux kernel version 3.1 and 3.5.
- If NMI watchdog is not disabled on x86_64 systems, **opcontrol** may fail to allocate the hardware performance counters it needs. The progress of this issue can be followed in bugzilla at https://bugzilla.redhat.com/show_bug.cgi?id=683176.
- Many Alpha ev67 events do not work. The progress of this issue can be followed in bugzilla https://bugzilla.redhat.com/show_bug.cgi?id=931875.

5.2.4. OProfile Documentation

For a more extensive information on OProfile, see **man oprofile**. Red Hat Enterprise Linux also provides two comprehensive guides to OProfile in **file:///usr/share/doc/oprofile-*version*/**:

OProfile Manual

A comprehensive manual with detailed instructions on the setup and use of OProfile is found at **file:///usr/share/doc/oprofile-*version*/oprofile.html**

OProfile Internals

Documentation on the internal workings of OProfile, useful for programmers interested in contributing to the OProfile upstream, can be found at **file:///usr/share/doc/oprofile-*version*/internals.html**

The *OProfile Integration User Guide* in the Eclipse **Help Contents** also provides detailed information on the setup and usage of the OProfile plug-in for Eclipse. This guide is provided by the **eclipse-oprofile** package.

5.3. SystemTap

SystemTap is a useful instrumentation platform for probing running processes and kernel activity on the Linux system. To execute a probe:

1. Write *SystemTap scripts* that specify which system events (for example, virtual file system reads, packet transmissions) should trigger specified actions (for example, print, parse, or otherwise manipulate data).
2. SystemTap translates the script into a C program, which it compiles into a kernel module.
3. SystemTap loads the kernel module to perform the actual probe.

SystemTap scripts are useful for monitoring system operation and diagnosing system issues with minimal intrusion into the normal operation of the system. You can quickly instrument running system test hypotheses without having to recompile and re-install instrumented code. To compile a SystemTap script that probes *kernel-space*, SystemTap uses information from three different *kernel information packages*:

- **kernel-*variant*-devel-*version***
- **kernel-*variant*-debuginfo-*version***
- **kernel-debuginfo-common-*arch*-*version***

These kernel information packages must match the kernel to be probed. In addition, to compile SystemTap scripts for multiple kernels, the kernel information packages of each kernel must also be installed.

An important new feature has been added as of Red Hat Enterprise Linux 6.1: the **--remote** option. This allows users to build the SystemTap module locally, and then execute it remotely via SSH. The syntax to use this is **--remote [USER@]HOSTNAME**; set the execution target to the specified SSH host, optionally using a different username. This option may be repeated to target multiple execution targets. Passes 1-4 are completed locally as normal to build the script, and then pass 5 copies the module to the target and runs it.

5.3.1. DynInst with SystemTap 2.0

SystemTap 2.0 introduces experamental support for running instrumentation using the *DynInst* system. DynInst is a pure-userspace binary manipulation library that allows programs to modify other running programs. It does this by inserting highly efficient instrumentation or other modifications. SystemTap 2.0 and later can use this as a backend to run a restricted class of scripts. In exchange for the restrictions, the instrumentation runs fast and entirely in user-space with no root access or kernel module operations required. The restrictions are evolving but are tighter than those for unprivileged user probing that relies on cryptography, kernel modules, and membership in special groups.

To use this experamental backend, add an extra **--runtime** option on the stap command line:

```
$ stap --runtime=stapdyn script.stp -c command
```

If the script requires facilities beyond those available with DynInst, SystemTap will advise. If this is the case, the standard kernel-module-based backends will have to be used with the **--runtime** option omitted.

5.3.2. SystemTap Compile Server

SystemTap in Red Hat Enterprise Linux 7 supports a *compile server and client* deployment. With this setup, the kernel information packages of *all* client systems in the network are installed on just one compile server host (or a few). When a client system attempts to compile a kernel module from a SystemTap script, it remotely accesses the kernel information it requires from the centralized compile server host.

A properly configured and maintained SystemTap compile server host offers the following benefits:

- The system administrator can verify the integrity of kernel information packages before making the packages available to users.
- The identity of a compile server can be authenticated using the *Secure Socket Layer* (SSL). SSL provides an encrypted network connection that prevents eavesdropping or tampering during transmission.
- Individual users can run their own servers and authorize them for their own use as trusted.
- System administrators can authorize one or more servers on the network as trusted for use by all users.
- A server that has not been explicitly authorized is ignored, preventing any server impersonations and similar attacks.

5.3.3. SystemTap Support for Unprivileged Users

For security purposes, users in an enterprise setting are rarely given privileged (that is, root or **sudo**) access to their own machines. In addition, full SystemTap functionality should also be restricted to privileged users, as this can provide the ability to completely take control of a system.

SystemTap in Red Hat Enterprise Linux 7 features a new option to the SystemTap client: **--unprivileged**. This option allows an unprivileged user to run **stap**. Of course, several restrictions apply to unprivileged users that attempt to run **stap**.

Note

An unprivileged user is a member of the group **stapusr** but is not a member of the group **stapdev** (and is not root).

Before loading any kernel modules created by unprivileged users, SystemTap verifies the integrity of the module using standard digital (cryptographic) signing techniques. Each time the **--unprivileged** option is used, the server checks the script against the constraints imposed for unprivileged users. If the checks are successful, the server compiles the script and signs the resulting

module using a self-generated certificate. When the client attempts to load the module, **staprun** first verifies the signature of the module by checking it against a database of trusted signing certificates maintained and authorized by root.

Once a signed kernel module is successfully verified, **staprun** is assured that:

- The module was created using a trusted systemtap server implementation.
- The module was compiled using the **--unprivileged** option.
- The module meets the restrictions required for use by an unprivileged user.
- The module has not been tampered with since it was created.

5.3.4. SSL and Certificate Management

SystemTap in Red Hat Enterprise Linux 7 implements authentication and security via certificates and public/private key pairs. It is the responsibility of the system administrator to add the credentials (that is, certificates) of compile servers to a database of trusted servers. SystemTap uses this database to verify the identity of a compile server that the client attempts to access. Likewise, SystemTap also uses this method to verify kernel modules created by compile servers using the **--unprivileged** option.

5.3.4.1. Authorizing Compile Servers for Connection

The first time a compile server is started on a server host, the compile server automatically generates a certificate. This certificate verifies the compile server's identity during SSL authentication and module signing.

In order for clients to access the compile server (whether on the same server host or from a client machine), the system administrator must add the compile server's certificate to a database of trusted servers. Each client host intending to use compile servers maintains such a database. This allows individual users to customize their database of trusted servers, which can include a list of compile servers authorized for their own use only.

5.3.4.2. Authorizing Compile Servers for Module Signing (for Unprivileged Users)

Unprivileged users can only load signed, authorized SystemTap kernel modules. For modules to be recognized as such, they have to be created by a compile server whose certificate appears in a database of trusted signers; this database must be maintained on each host where the module will be loaded.

5.3.4.3. Automatic Authorization

Servers started using the **stap-server** initscript are automatically authorized to receive connections from all clients on the same host.

Servers started by other means are automatically authorized to receive connections from clients on the same host run by the user who started the server. This was implemented with convenience in mind; users are automatically authorized to connect to a server they started themselves, provided that both client and server are running on the same host.

Whenever root starts a compile server, *all* clients running on the same host automatically recognize the server as authorized. However, Red Hat advises that you refrain from doing so.

Similarly, a compile server initiated through **stap-server** is automatically authorized as a trusted signer on the host in which it runs. If the compile server was initiated through other means, it is not automatically authorized as such.

5.3.5. SystemTap Documentation

For more detailed information about SystemTap, see the following books (also provided by Red Hat):

- *SystemTap Beginner's Guide*
- *SystemTap Tapset Reference*

The *SystemTap Beginner's Guide* and *SystemTap Tapset Reference* are also available locally when you install the **systemtap** package:

- **file:///usr/share/doc/systemtap-*version*/SystemTap_Beginners_Guide/index.html**
- **file:///usr/share/doc/systemtap-*version*/SystemTap_Beginners_Guide.pdf**
- **file:///usr/share/doc/systemtap-*version*/tapsets/index.html**
- **file:///usr/share/doc/systemtap-*version*/tapsets.pdf**

Section 5.3.2, “SystemTap Compile Server”, Section 5.3.3, “SystemTap Support for Unprivileged Users”, and Section 5.3.4, “ SSL and Certificate Management” are all excerpts from the *SystemTap Support for Unprivileged Users and Server Client Deployment* whitepaper. This whitepaper also provides more details on each feature, along with a case study to help illustrate their application in a real-world environment.

5.4. Performance Counters for Linux (PCL) Tools and perf

Performance Counters for Linux (PCL) is a new kernel-based subsystem that provides a framework for collecting and analyzing performance data. These events will vary based on the performance monitoring hardware and the software configuration of the system. Red Hat Enterprise Linux 6 includes this kernel subsystem to collect data and the user-space tool **perf** to analyze the collected performance data.

The PCL subsystem can be used to measure hardware events, including retired instructions and processor clock cycles. It can also measure software events, including major page faults and context switches. For example, PCL counters can compute the *Instructions Per Clock* (IPC) from a process's counts of instructions retired and processor clock cycles. A low IPC ratio indicates the code makes poor use of the CPU. Other hardware events can also be used to diagnose poor CPU performance.

Performance counters can also be configured to record samples. The relative frequency of samples can be used to identify which regions of code have the greatest impact on performance.

5.4.1. Perf Tool Commands

Useful **perf** commands include the following:

perf stat

This **perf** command provides overall statistics for common performance events, including instructions executed and clock cycles consumed. Options allow selection of events other than the default measurement events.

perf record

This **perf** command records performance data into a file which can be later analyzed using **perf report**.

perf report

This **perf** command reads the performance data from a file and analyzes the recorded data.

perf list

This **perf** command lists the events available on a particular machine. These events will vary based on the performance monitoring hardware and the software configuration of the system.

Use **perf help** to obtain a complete list of **perf** commands. To retrieve **man** page information on each **perf** command, use **perf help *command***.

5.4.2. Using Perf

Using the basic PCL infrastructure for collecting statistics or samples of program execution is relatively straightforward. This section provides simple examples of overall statistics and sampling.

To collect statistics on **make** and its children, use the following command:

```
# perf stat -- make all
```

The **perf** command collects a number of different hardware and software counters. It then prints the following information:

```
Performance counter stats for 'make all':

  244011.782059  task-clock-msecs         #      0.925 CPUs
          53328  context-switches          #      0.000 M/sec
            515  CPU-migrations            #      0.000 M/sec
        1843121  page-faults               #      0.008 M/sec
   789702529782  cycles                    #   3236.330 M/sec
  1050912611378  instructions              #      1.331 IPC
   275538938708  branches                  #   1129.203 M/sec
     2888756216  branch-misses             #      1.048 %
     4343060367  cache-references          #     17.799 M/sec
      428257037  cache-misses              #      1.755 M/sec

  263.779192511  seconds time elapsed
```

The **perf** tool can also record samples. For example, to record data on the **make** command and its children, use:

```
# perf record -- make all
```

This prints out the file in which the samples are stored, along with the number of samples collected:

```
[ perf record: Woken up 42 times to write data ]
[ perf record: Captured and wrote 9.753 MB perf.data (~426109 samples) ]
```

As of Red Hat Enterprise Linux 6.4, a new functionality to the **{}** group syntax has been added that allows the creation of event groups based on the way they are specified on the command line.

The current **--group** or **-g** options remain the same; if it is specified for record, stat, or top command, all the specified events become members of a single group with the first event as a group leader.

The new **{}** group syntax allows the creation of a group like:

```
# perf record -e '{cycles, faults}' ls
```

The above results in a single event group containing *cycles* and *faults* events, with the *cycles* event as the group leader.

All groups are created with regards to threads and CPUs. As such, recording an event group within two threads on a server with four CPUs will create eight separate groups.

It is possible to use a standard event modifier for a group. This spans over all events in the group and updates each event modifier settings.

```
# perf record -r '{faults:k,cache-references}:p'
```

The above command results in the **:kp** modifier being used for *faults*, and the **:p** modifier being used for the *cache-references* event.

Performance Counters for Linux (PCL) Tools conflict with OProfile

Both OProfile and Performance Counters for Linux (PCL) use the same hardware Performance Monitoring Unit (PMU). If OProfile is currently running while attempting to use the PCL **perf** command, an error message like the following occurs when starting OProfile:

```
Error: open_counter returned with 16 (Device or resource busy).
/usr/bin/dmesg may provide additional information.

Fatal: Not all events could be opened.
```

To use the **perf** command, first shut down OProfile:

```
# opcontrol --deinit
```

You can then analyze **perf.data** to determine the relative frequency of samples. The report output includes the command, object, and function for the samples. Use **perf report** to output an analysis of **perf.data**. For example, the following command produces a report of the executable that consumes the most time:

```
# perf report --sort=comm
```

The resulting output:

```
# Samples: 1083783860000
#
# Overhead          Command
# ........  ...............
#
    48.19%         xsltproc
```

```
    44.48%        pdfxmltex
     6.01%             make
     0.95%             perl
     0.17%       kernel-doc
     0.05%          xmllint
     0.05%              cc1
     0.03%               cp
     0.01%            xmlto
     0.01%               sh
     0.01%          docproc
     0.01%               ld
     0.01%              gcc
     0.00%               rm
     0.00%              sed
     0.00%   git-diff-files
     0.00%             bash
     0.00%   git-diff-index
```

The column on the left shows the relative frequency of the samples. This output shows that **make** spends most of this time in **xsltproc** and the **pdfxmltex**. To reduce the time for the **make** to complete, focus on **xsltproc** and **pdfxmltex**. To list the functions executed by **xsltproc**, run:

```
# perf report -n --comm=xsltproc
```

This generates:

```
comm: xsltproc
# Samples: 472520675377
#
# Overhead  Samples                    Shared Object  Symbol
# ........ ..........  ..............................  ......
#
    45.54%215179861044  libxml2.so.2.7.6               [.]
xmlXPathCmpNodesExt
    11.63%54959620202  libxml2.so.2.7.6               [.]
xmlXPathNodeSetAdd__internal_alias
     8.60%40634845107  libxml2.so.2.7.6               [.]
xmlXPathCompOpEval
     4.63%21864091080  libxml2.so.2.7.6               [.]
xmlXPathReleaseObject
     2.73%12919672281  libxml2.so.2.7.6               [.]
xmlXPathNodeSetSort__internal_alias
     2.60%12271959697  libxml2.so.2.7.6               [.] valuePop
     2.41%11379910918  libxml2.so.2.7.6               [.]
xmlXPathIsNaN__internal_alias
     2.19%10340901937  libxml2.so.2.7.6               [.]
valuePush__internal_alias
```

Chapter 6. Writing Documentation

Red Hat_Enterprise Linux 7 offers the **Doxygen** tool for generating documentation from source code and for writing standalone documentation.

6.1. Doxygen

Doxygen is a documentation tool that creates reference material both online in HTML and offline in Latex. It does this from a set of documented source files which makes it easy to keep the documentation consistent and correct with the source code.

6.1.1. Doxygen Supported Output and Languages

Doxygen has support for output in:

- RTF (MS Word)
- PostScript
- Hyperlinked PDF
- Compressed HTML
- Unix man pages

Doxygen supports the following programming languages:

- C
- C++
- C#
- Objective -C
- IDL
- Java
- VHDL
- PHP
- Python
- Fortran
- D

6.1.2. Getting Started

Doxygen uses a configuration file to determine its settings, therefore it is paramount that this be created correctly. Each project requires its own configuration file. The most painless way to create the configuration file is with the command **doxygen -g *config-file***. This creates a template configuration file that can be easily edited. The variable *config-file* is the name of the configuration file.

If it is committed from the command it is called Doxyfile by default. Another useful option while creating the configuration file is the use of a minus sign (**-**) as the file name. This is useful for scripting as it will cause Doxygen to attempt to read the configuration file from standard input (**stdin**).

The configuration file consists of a number of variables and tags, similar to a simple Makefile. For example:

```
TAGNAME = VALUE1 VALUE2...
```

For the most part these can be left alone but should it be required to edit them see the *configuration page* of the Doxygen documentation website for an extensive explanation of all the tags available. There is also a GUI interface called **doxywizard**. If this is the preferred method of editing then documentation for this function can be found on the *Doxywizard usage page* of the Doxygen documentation website.

There are eight tags that are useful to become familiar with.

INPUT

For small projects consisting mainly of C or C++ source and header files it is not required to change anything. However, if the project is large and consists of a source directory or tree, then assign the root directory or directories to the INPUT tag.

FILE_PATTERNS

File patterns (for example, ***.cpp** or ***.h**) can be added to this tag allowing only files that match one of the patterns to be parsed.

RECURSIVE

Setting this to **yes** will allow recursive parsing of a source tree.

EXCLUDE and EXCLUDE_PATTERNS

These are used to further fine-tune the files that are parsed by adding file patterns to avoid. For example, to omit all **test** directories from a source tree, use **EXCLUDE_PATTERNS = */test/***.

EXTRACT_ALL

When this is set to **yes**, doxygen will pretend that everything in the source files is documented to give an idea of how a fully documented project would look. However, warnings regarding undocumented members will not be generated in this mode; set it back to **no** when finished to correct this.

SOURCE_BROWSER and INLINE_SOURCES

By setting the **SOURCE_BROWSER** tag to **yes** doxygen will generate a cross-reference to analyze a piece of software's definition in its source files with the documentation existing about it. These sources can also be included in the documentation by setting **INLINE_SOURCES** to **yes**.

6.1.3. Running Doxygen

Running **doxygen *config-file*** creates **html**, **rtf**, **latex**, **xml**, and / or **man** directories in whichever directory doxygen is started in, containing the documentation for the corresponding filetype.

HTML OUTPUT

This documentation can be viewed with a HTML browser that supports cascading style sheets (CSS), as well as DHTML and Javascript for some sections. Point the browser (for example, Mozilla, Safari, Konqueror, or Internet Explorer 6) to the **index.html** in the **html** directory.

LaTeX OUTPUT

Doxygen writes a **Makefile** into the **latex** directory in order to make it easy to first compile the Latex documentation. To do this, use a recent teTeX distribution. What is contained in this directory depends on whether the **USE_PDFLATEX** is set to **no**. Where this is true, typing **make** while in the **latex** directory generates **refman.dvi**. This can then be viewed with **xdvi** or converted to **refman.ps** by typing **make ps**. Note that this requires **dvips**.

There are a number of commands that may be useful. The command **make ps_2on1** prints two pages on one physical page. It is also possible to convert to a PDF if a ghostscript interpreter is installed by using the command **make pdf**. Another valid command is **make pdf_2on1**. When doing this set **PDF_HYPERLINKS** and **USE_PDFLATEX** tags to **yes** as this will cause **Makefile** will only contain a target to build **refman.pdf** directly.

RTF OUTPUT

This file is designed to import into Microsoft Word by combining the RTF output into a single file: **refman.rtf**. Some information is encoded using fields but this can be shown by selecting all (**CTRL+A** or Edit -> select all) and then right-click and select the **toggle fields** option from the drop down menu.

XML OUTPUT

The output into the **xml** directory consists of a number of files, each compound gathered by doxygen, as well as an **index.xml**. An XSLT script, **combine.xslt**, is also created that is used to combine all the XML files into a single file. Along with this, two XML schema files are created, **index.xsd** for the index file, and **compound.xsd** for the compound files, which describe the possible elements, their attributes, and how they are structured.

MAN PAGE OUTPUT

The documentation from the **man** directory can be viewed with the **man** program after ensuring the **manpath** has the correct man directory in the man path. Be aware that due to limitations with the man page format, information such as diagrams, cross-references and formulas will be lost.

6.1.4. Documenting the Sources

There are three main steps to document the sources.

1. First, ensure that **EXTRACT_ALL** is set to **no** so warnings are correctly generated and documentation is built properly. This allows doxygen to create documentation for documented members, files, classes and namespaces.

2. There are two ways this documentation can be created:

   A *special* documentation block

   This comment block, containing additional marking so Doxygen knows it is part of the documentation, is in either C or C++. It consists of a brief description, or a detailed description. Both of these are optional. What is not optional, however, is the *in body* description. This then links together all the comment blocks found in the body of the method or function.

Note

While more than one brief or detailed description is allowed, this is not recommended as the order is not specified.

The following will detail the ways in which a comment block can be marked as a detailed description:

- C-style comment block, starting with two asterisks (*) in the JavaDoc style.

```
/**
 * ... documentation ...
 */
```

- C-style comment block using the Qt style, consisting of an exclamation mark (!) instead of an extra asterisk.

```
/*!
 * ... documentation ...
 */
```

- The beginning asterisks on the documentation lines can be left out in both cases if that is preferred.
- A blank beginning and end line in C++ also acceptable, with either three forward slashes or two forward slashes and an exclamation mark.

```
///
/// ... documentation
///
```

or

```
//!
//! ... documentation ...
//!
```

- Alternatively, in order to make the comment blocks more visible a line of asterisks or forward slashes can be used.

```
//////////////////////////////////////////////////
/// ... documentation ...
//////////////////////////////////////////////////
```

or

```
/*********************************************//**
 * ... documentation ...
 **********************************************/
```

Note that the two forwards slashes at the end of the normal comment block start a special comment block.

There are three ways to add a brief description to documentation.

- To add a brief description use **\brief** above one of the comment blocks. This brief section ends at the end of the paragraph and any further paragraphs are the detailed descriptions.

```
/*! \brief brief documentation.
 *         brief documentation.
 *
 *  detailed documentation.
 */
```

- By setting **JAVADOC_AUTOBRIEF** to **yes**, the brief description will only last until the first dot followed by a space or new line. Consequentially limiting the brief description to a single sentence.

```
/** Brief documentation. Detailed documentation continues
* from here.
 */
```

  This can also be used with the above mentioned three-slash comment blocks (*///*).

- The third option is to use a special C++ style comment, ensuring this does not span more than one line.

```
/// Brief documentation.
/** Detailed documentation. */
```

  or

```
//! Brief documentation.

//! Detailed documentation //! starts here
```

  The blank line in the above example is required to separate the brief description and the detailed description, and **JAVADOC_AUTOBRIEF** must to be set to **no**.

Examples of how a documented piece of C++ code using the Qt style can be found on the *Doxygen documentation website*

It is also possible to have the documentation after members of a file, struct, union, class, or enum. To do this add a < marker in the comment block.\

```
int var; /*!< detailed description after the member */
```

Or in a Qt style as:

```
int var; /**< detailed description after the member */
```

or

```
int var; //!< detailed description after the member
         //!<
```

or

```
int var; ///< detailed description after the member
         ///<
```

For brief descriptions after a member use:

```
int var; //!< brief description after the member
```

or

```
int var; ///< brief description after the member
```

Examples of these and how the HTML is produced can be viewed on the *Doxygen documentation website*

Documentation at other places

While it is preferable to place documentation in front of the code it is documenting, at times it is only possible to put it in a different location, especially if a file is to be documented; after all it is impossible to place the documentation in front of a file. This is best avoided unless it is absolutely necessary as it can lead to some duplication of information.

To do this it is important to have a structural command inside the documentation block. Structural commands start with a backslash (\) or an at-sign (@) for JavaDoc and are followed by one or more parameters.

```
/*! \class Test
    \brief A test class.

    A more detailed description of class.
 */
```

In the above example the command **\class** is used. This indicates that the comment block contains documentation for the class 'Test'. Others are:

- **\struct**: document a C-struct
- **\union**: document a union
- **\enum**: document an enumeration type
- **\fn**: document a function
- **\var**: document a variable, typedef, or enum value
- **\def**: document a #define
- **\typedef**: document a type definition
- **\file**: document a file
- **\namespace**: document a namespace
- **\package**: document a Java package
- **\interface**: document an IDL interface

3. Next, the contents of a special documentation block is parsed before being written to the HTML and / Latex output directories. This includes:

    a. Special commands are executed.

    b. Any white space and asterisks (*) are removed.

    c. Blank lines are taken as new paragraphs.

    d. Words are linked to their corresponding documentation. Where the word is preceded by a percent sign (%) the percent sign is removed and the word remains.

    e. Where certain patterns are found in the text, links to members are created. Examples of this can be found on the *automatic link generation page* on the Doxygen documentation website.

    f. When the documentation is for Latex, HTML tags are interpreted and converted to Latex equivalents. A list of supported HTML tags can be found on the *HTML commands page* on the Doxygen documentation website.

6.1.5. Resources

More information can be found on the Doxygen website.

- *Doxygen homepage*
- *Doxygen introduction*
- *Doxygen documentation*
- *Output formats*

Appendix

A.1. mallopt

mallopt is a library call that allows a program to change the behavior of the malloc memory allocator.

> **Example A.1. Allocator heuristics**
>
> An allocator has heuristics to determine long versus short lived objects. For the former, it attempts to allocate with **mmap**. For the later, it attempts to allocate with **sbrk**.
>
> In order to override these heuristics, set **M_MMAP_THRESHOLD**.

In multi-threaded applications, the allocator creates multiple *arenas* in response to lock contention in existing arenas. This can improve the performance significantly for some multi-threaded applications at the cost of an increase in memory usage. To keep this under control, limit the number of arenas that can be created by using the **mallopt** interface.

The allocator has limits on the number of arenas it can create. For 32bit targets, it will create 2 * # core arenas; for 64bit targets, it will create 8 * # core arenas. **mallopt** allows the developer to override those limits.

> **Example A.2. mallopt**
>
> To ensure no more than eight arenas are created, issue the following library call:
>
> ```
> mallopt (M_ARENA_MAX, 8);
> ```

The first argument for **mallopt** can be:

- M_MXFAST
- M_TRIM_THRESHOLD
- M_TOP_PAD
- M_MMAP_THRESHOLD
- M_MMAP_MAX
- M_CHECK_ACTION
- M_PERTURB
- M_ARENA_TEST
- M_ARENA_MAX

Specific definitions for the above can be found at http://www.makelinux.net/man/3/M/mallopt.

malloc_trim

malloc_trim is a library call that requests the allocator return any unused memory back to the operating system. This is normally automatic when an object is freed. However, in some cases when freeing small objects, **glibc** might not immediately release the memory back to the operating system. It does this so that the free memory can be used to satisfy upcoming memory allocation requests as it is expensive to allocate from and release memory back to the operating system.

malloc_stats

malloc_stats is used to dump information about the allocator's internal state to **stderr**. Using **mallinfo** is similar to this, but it places the state into a structure instead.

Further Information

More information on **mallopt** can be found at http://www.makelinux.net/man/3/M/mallopt and http://www.gnu.org/software/libc/manual/html_node/Malloc-Tunable-Parameters.html.

Revision History

Revision 1-8	**Thu Feb 19 2015**	**Robert Krátký**

Build for 7.1 GA release.

Revision 1-6	**Fri Dec 06 2014**	**Robert Krátký**

Update to sort order on the Red Hat Customer Portal.

Revision 1-4	**Wed Nov 11 2014**	**Robert Krátký**

Build for 7.0 GA release.

Index

U

V