Red Hat Driver Update Packages

Official Reference Guide

Jon Masters <jcm@redhat.com>

Version: 0.99
Date: January 6th, 2011
Contents

1 Introduction ................................................................. 7
  1.1 What are Driver Updates? ........................................... 7
  1.2 Why use Driver Updates? ........................................... 8
  1.3 Further Resources .................................................. 10
  1.4 Feedback ............................................................. 11

2 Background Requirements .............................................. 13
  2.1 Supported driver types ............................................. 13
  2.2 Driver Source Versions ............................................ 14
  2.3 Red Hat Kernel ABI ................................................ 15
     2.3.1 Tracking kernel ABI changes ................................ 17
  2.4 Build Environment ................................................ 17
     2.4.1 Installing Red Hat Enterprise Linux 6 .................... 18
     2.4.2 Configuring rpmbuild ........................................ 19
     2.4.3 Using Koji and mock ........................................ 20

3 Red Hat Kernel ABI .................................................. 23
  3.1 What is the kernel ABI? ............................................ 23
  3.2 Obtaining the official kABI lists ................................ 25
  3.3 Verifying kABI requirements ..................................... 26
     3.3.1 Using the command line ................................... 26
     3.3.2 Using the abi-check script ................................ 27
     3.3.3 Using the kabi-yum-plugins package ...................... 27
  3.4 Working with the kABI ............................................ 27
     3.4.1 Appropriate use of kernel interfaces ..................... 28
     3.4.2 Initializing and handling memory ......................... 28
  3.5 Requesting additions to kABI ................................... 29
  3.6 Kernel ABI Breakage ............................................. 31
4 Driver Update Packages
  4.1 Preparation ................................................. 35
    4.1.1 Kernel driver source ................................. 36
  4.2 Creating a Driver Update package ...................... 37
    4.2.1 The %kernel_module_package macro ................. 40
  4.3 Building a Driver Update Package ..................... 43
  4.4 Shipping an entirely new driver ..................... 44
  4.5 Updating an existing driver ........................... 44
  4.6 Replacing an existing driver .......................... 46
  4.7 Firmware .................................................. 46
  4.8 Multiple drivers and dependencies .................... 48
  4.9 Further Recommendations ............................... 49
    4.9.1 Naming Guidance .................................... 49

5 Driver Update Disks ........................................ 51
  5.1 Preparation ................................................. 51
  5.2 Installing and using ddiskit 2 ......................... 52
  5.3 The Red Hat Installer ................................... 53
  5.4 Network Driver Update Disks ............................ 53
  5.5 Multiple architecture support .......................... 54
Preface

Driver Update Packages from Red Hat implement a mechanism whereby system drivers (and other loadable Linux kernel modules) may be added to, or updated on, existing installations of Red Hat Enterprise Linux. Such drivers can be pre-compiled and distributed (along with source) in the form of regular RPM packages that users may install easily onto their systems, without requiring build tools or specialist knowledge concerning the kernel.

In addition to the capability to build and install Driver Update packages (henceforth known as “Driver Updates”), Red Hat’s related Driver Update Program provides a means for Red Hat to ship certain Open Source RHEL 6 Driver Updates it builds directly at the request of partner companies. The specifics of that program are documented elsewhere, and further information can be obtained directly by contacting Red Hat Partner Engineering.

This guide covers the mechanics of building and installing Driver Update packages, along with explanations of some of the associated pieces that are required for proper functioning of such drivers. These include a summary of the Red Hat kernel ABI (kABI), a description of the official process for requesting additions to the kABI, and the tools and processes for building Red Hat Driver Update Disks. This guide also contains some commentary on proper test procedures and deployment suggestions for Driver Updates.

Note that this is an evolving document that pertains to RHEL 6 (and later) systems. Please send all feedback to the author, referencing the version of this document in your email. In time, it is hoped that this guide will form a complete reference set of documentation for building and using both Driver Updates, and Driver Update Disks with Red Hat Enterprise Linux.

Jon Masters, Cambridge Massachusetts, January 2011.
Chapter 1

Introduction

In this chapter you will learn about Driver Updates, what they are, how they work, and what kinds of situations they are intended to be used in. You will also learn a little about the Red Hat kernel ABI, setting the stage for a later chapter devoted to the topic. Toward the end of the chapter, there are a number of references in the form of further reading items. As you read this (and other) chapters, keep in mind that a full glossary of Red Hat and Driver Update nomenclature is located at the end of this book.

Driver Updates provide many capabilities, but foremost amongst them is the ability to provide an existing, installed RHEL system with interim support for a new or improved piece of hardware that is not yet supported by the main RHEL 6 product release. Then, support for the new hardware can be added to RHEL 6 in a timely manner, through the standard update cycle. Once the RHEL 6 distribution contains built-in support for a given piece of hardware, the corresponding Driver Update can be deprecated.

1.1 What are Driver Updates?

Linux systems are designed to be modular in nature and include support for loading certain kernel functionality such as device drivers in the form of kernel modules. These modules are typically shipped as part of the system (in the Linux kernel package) and are automatically loaded when appropriate hardware devices that require them are detected. But modules can also be shipped separately from the system, either as source, or as Driver Updates.

Strictly speaking, the term Driver Updates should be Module Updates, since
updates are in fact shipped in the form of regular Linux kernel modules and are not limited solely to new device drivers. However, the majority of “Module Updates” are to device drivers, and so the term Driver Updates has become the convention. Regardless, you are able to ship updates to file system modules and certain other kernel components just as easily as to regular drivers, using the same Driver Updates process. There are, naturally, some limits to what is possible. These limits will be explained in due course.

Red Hat Driver Updates differ from driver updates on non-Red Hat systems because they do not generally need to be recompiled by the user. This is made possible by a special value-added feature in RHEL known as the Red Hat kernel ABI (kABI). The kABI provides a stable binary-level kernel interface for loadable (third party) modules that remains compatible across security and errata fixes, as well as between one minor update and the next (such as the transition from RHEL 6.0 to RHEL 6.1). Red Hat carefully develops its kernel to retain the special level of compatibility guaranteed by the kABI. You will learn more about this in a later chapter of this book.

Driver Updates take the form of regular system RPM packages that contain Linux Kernel Module(s) - files ending in the .ko filename extension. These are installed into special locations that have been reserved for updates. System policy encoded into the module loading utilities will then prioritize or replace driver updates according to various configurable options. Once an update is installed, it will be loaded in an identical fashion to any other regular system driver that is compatible with the running system kernel.

Technically, this is achieved through the use of the "udev" and "module-init-tools" utilities - the former handles kernel notification of new devices detected by the kernel, and so forth, and the latter actually loads the required functionality, applying any necessary global and driver specific policy in the process. Most users do not need to interact with these tools directly, since their function is automated. You can find more information about the mechanics and plumbing of Linux kernel modules in a variety of books, such as "Linux Device Drivers" (3rd edition, O'Reilly), and on the Internet.

1.2 Why use Driver Updates?

There are several reasons why it might be desirable or necessary to ship driver updates. These apply both in the case of Red Hat shipping a Driver
1.2. WHY USE DRIVER UPDATES?

Update directly (through the Red Hat Driver Update Program), and in the case that a third party wishes to supply an update of their own. Some of the key reasons for shipping Driver Updates include:

- A new driver (or other kernel module) is not yet provided by the regular system software and a solution is required in the interim. Red Hat supports many different devices and kernel capabilities, but not all. There are some more niche users who desire additional kernel modules that are not or cannot be shipped by Red Hat directly.

- An existing driver is out of date and does not support new hardware devices that are entering the market before the next distribution update. Most of the revenue generated by a new device entering the marketplace is generated in a relatively small amount of time, and so it is desirable to support the new hardware as quickly as possible. When the distribution is updated to support the new hardware officially, the driver update can be removed, deprecated, or it can be shipped in such a way as to automatically deprecate at that time.

- A replacement third party driver is available that adds functionality not available in the official driver. This might include a storage driver that augments the existing system-supplied driver with various extra management capabilities that are not present in the standard driver. Red Hat only supports drivers that it ships, but some users desire to install third party drivers that are supported by the third party.

There are, of course, many other reasons for shipping driver updates, since Linux is an extremely flexible Operating System with many kinds of user. The preceding list is meant only as an example of a few common situations.

One of the often cited reasons for shipping a driver update is that new hardware devices are coming to market for which updated support is required. This is the case for which the mechanism has been primarily designed, as a "stop gap" solution for supplying updates that can be used until an updated version of the Operating System is released that contains official support. This is true for all Driver Updates shipped directly by Red Hat, since they are intended to be deprecated as soon as convenient in a RHEL 6 update.

As such, many (or most) updates should be relatively short lived - perhaps just one update cycle (e.g. available for RHEL 6.0, then unnecessary in 6.1 and simply deprecated in such a way as to not interfere with the 6.1
driver). Using the Driver Update mechanism, it is possible to support new hardware devices in the months between those RHEL 6 minor releases, or for longer periods in the case that a driver is being shipped for which there is no official support (and will not be) in the distribution.

Driver Updates should only be shipped to customers when necessary. The process is generally simpler than on non-RHEL systems, due to the packaging process, and the ability to avoid rebuilding drivers frequently, but it is not a trivial undertaking to replace fundamental system device drivers, and does come with support burden. Use this document as a guide in the case that it is necessary to package a driver update, but you are strongly recommended to avoid shipping drivers for any Linux system unless necessary.

NOTE: Red Hat has an official program for shipping certain of their own Driver Updates, known as the Driver Update Program. This program fully exercises each of the same technologies documented in this guide, and provides a means for Red Hat customers to obtain an officially supported Red Hat driver that can be used in interim releases prior to official support for hardware being added in the RHEL 6 kernel. The program operates in conjunction with the Partner Engineering team, and is subject to certain requirements surrounding the planned inclusion of a driver into future RHEL releases. Contact Red Hat Partner Engineering for further information.

1.3 Further Resources

This reference provides an introduction to Driver Updates as used by Red Hat Enterprise Linux 6. It also documents some of the processes involved in building drivers, and some aspects of overall Linux kernel development. It is more than sufficient to follow the instructions in this guide if you are seeking to package a driver from source code that is already known to work with RHEL 6. For example, code you have tested on the RHEL 6 kernel.

There are a number of Red Hat resources available to those working with Driver Updates and with kernel ABI. Copies of the kernel ABI reference files, Driver Update Disk packaging tools mentioned in this guide, and various example drivers can be found at the following locations (the former for kernel ABI reference, and the later for other Driver Updates resources):

http://people.redhat.com/jcm/el6/kabi/
http://people.redhat.com/jcm/el6/dup/
Additional information is available from Red Hat Partner Engineering, and is periodically mailed to other interested parties upon request.

This guide is not intended to serve as a reference for actually developing Linux device drivers, nor for "backporting" them to the RHEL 6 kernel. However, many good references on Linux kernel development do exist. These include the following non-exhaustive list of reference books on the subject:

- Linux Device Drivers - This is a primary reference for those seeking to get involved with Linux driver development. It is available online under an Open Source license, or you can support the authors by purchasing a copy through your local bookseller.

- Linux Kernel in a nutshell - This book documents the processes involved in working with the kernel community, of building the kernel, and of getting more involved generally. It is not a full programming reference, but is a good higher level guide.

- Linux Kernel Development - This is a recently updated book that provides a summary of the structure of the Linux kernel, of its core algorithms, and is a good general reference.

If you are unfamiliar with the Linux kernel, with building Linux drivers, or with other technical aspects, one of these books may help you. If you are seeking to backport a driver to the RHEL 6 kernel, and lack the expertise, contact Red Hat Global Engineering Services, who may be able to provide you with paid consultancy services. If you are a Red Hat partner, you can also work with Red Hat directly to get your driver into RHEL 6.

1.4 Feedback

To provide feedback, or to report a defect in this document, with the Driver Update packaging process, Red Hat Driver Update Program, or the kernel ABI, please file a Bug in the Red Hat Bugzilla (bugzilla.redhat.com), using the "driver-update-program” component. This is the fastest, and most expedient way to ensure that your concern is addressed in a timely manner.
Chapter 2

Background Requirements

This chapter introduces the various background requirements for working with Red Hat Driver Updates, such as describing the appropriate development system configuration, and the tools that are necessary to build Driver Update packages. You will learn about the supported types of driver, which driver source code to use, and how the Red Hat kernel ABI works at a high level. You will also learn about some of the (optional) alternative approaches to building Driver Updates, including use of mock and Koji build systems.

Driver Updates take advantage of a number of Red Hat technologies, such as the kernel ABI (kABI), installer extensions, and so forth. These extend traditional Linux Kernel Modules for Enterprise computing environments by facilitating ease of packaging, distribution, and updates. But there are some limits, often caused by the lack of an official (non-Red Hat) Linux kernel ABI or stable driver interface in the "mainline" Linux kernel. The net result of this is that some drivers cannot be packaged, and some others must necessarily be re-built from time to time (an explanation follows).

2.1 Supported driver types

Linux supports modular components that add new drivers, file systems, and other functionality to the kernel. These cover the vast majority of updates that you are likely to be interested in shipping, but you should know that there are some general limits imposed by the design of the Linux kernel, or by the design of Red Hat Enterprise Linux. Those limits upon the kind of module that can be created include (but are not limited to):

- It is not possible to add new architecture, CPU, platform, or chipset
support to Linux without re-building the kernel itself. Most updates
do not need to add such low-level features to the kernel.

- It is not possible to replace an entire subsystem of the kernel. For
  example, it is not intended to replace the SCSI, networking, or wireless
  stacks (even the loadable module components).

- Driver Updates using non-kABI interfaces may need to be rebuilt and
  updated from time to time. Further information follows in the section
  of this chapter devoted to Red Hat kernel ABI.

Typically, it is possible to ship most Linux device driver kernel modules
that can be compiled against the version of the Linux kernel used in RHEL
6 (based upon 2.6.32, with some additions from subsequent kernel versions).
In the case that you have a simple device driver for a new or updated hard-
ware adapter card, there is a very good chance that you can avail yourself
of the Driver Update packaging process described in this guide.

2.2 Driver Source Versions

The Red Hat Enterprise Linux 6 kernel is based upon the official 2.6.32
kernel, but it also has some additions from Red Hat, and from later kernel
versions. Over time, new features added to the ”mainline” kernel are ”back
ported” (added to the RHEL 6 kernel), where possible. This facilitates Red
Hat’s own work to add new device drivers and capabilities to the kernel
that would otherwise require substantial modification of device drivers, but
instead allow relatively more recent driver sources to be built for RHEL 6.

Over time, the RHEL 6 kernel will deviate from the official ”upstream”
kernel in some respects, as pieces of the upstream kernel are re-written. For
example, interrupt thread handlers have changed since RHEL 6 was cre-
ated. All of this means that some device drivers will need an amount of
refactoring to work with the RHEL 6 kernel. Some will just compile and
work, others will use interfaces not present in the RHEL 6 kernel or expect
a particular behavior that differs. You or your engineers will need to decide
the best course of action to ensure your driver works with the RHEL 6 kernel.

It is assumed in this document that you already have a ”port” of a working
driver that compiles and operates using the RHEL 6 (2.6.32 derived) kernel,
and that you are now seeking to package it for release as a Driver Update.
If this is not the case, please ensure that the driver works with the RHEL 6
Red Hat kernel before proceeding with the following packaging steps. Merely ensuring that the driver compiles against an upstream 2.6.32 kernel is insufficient; it is necessary to work with the RHEL 6 kernel directly at this stage.

2.3 Red Hat Kernel ABI

Red Hat provides a powerful capability for those working on device drivers for the RHEL 6 kernel: the Red Hat kernel ABI. The kABI, which is documented further in the next chapter, provides certain guarantees that particular Linux kernel interfaces will remain stable and continue to operate in the same fashion in future updates. This means that a driver can be pre-compiled, packaged (along with source), and that it will continue to operate without the user having to install specialist compilers, and the like.

kABI is powerful, but it has limits. Driver Updates affecting certain fundamental subsystems of the kernel undergoing very active upstream development by the Linux kernel community have additional limitations imposed upon them. Since it is not always possible to predict future upstream development, Red Hat kernel engineering may consider particular kernel interfaces not to be part of the official kernel ABI. These non-kABI interfaces will never have been listed on the kernel ABI lists published by Red Hat and could therefore change in a minor product update (for example, in RHEL 6.1 or 6.2, but not in an update for the current update - e.g. 6.0). Typically, minor updates are released annually or semi-annually, meaning that affected drivers may have to be recompiled and updated on a similar timescale.

Minor releases to RHEL 6 are well planned and known about many months ahead of time. Thus, there should be more than sufficient time to schedule the rebuilding of certain Driver Updates using non-kABI interfaces to fall in line with the release of RHEL 6 minor updates. It is generally possible to test for changes to non-kABI interfaces in a pending update to RHEL 6 by the time that a beta release for that update is made. Once a beta has been released, further changes to core kernel interfaces outside of the kernel ABI in that release are considerably less likely to occur (but remain possible).

\footnote{Red Hat reserves the right to make changes to the kernel at any time. However, generally efforts are made and processes are in place not to change even non-kABI interfaces in security and "errata" updates unless necessary. Interfaces considered to be officially part of the kABI are not changed, even in minor updates, with the exception of unforeseen exceptional situations involving otherwise uncorrectable security issues, and the like.}
Examples of possible Driver Updates that may use non-kABI interfaces, and might therefore require periodic recompilation as of this writing include (but are not necessarily limited to) the following:

- **ACPI.** Certain ACPI drivers may need to be re-built in future minor update releases as interfaces like `acpi_bus_get_device` are not considered to be part of the current kernel ABI in RHEL 6.

- **ATA.** Drivers for ATA devices may need to be re-built in minor updates since the kernel `libata` layer has never been considered part of kABI.

- **Graphics.** Many DRM (3D rendering model) interfaces such as `drm_connector_init` are not considered part of RHEL 6 kABI at this time. It is therefore quite likely that you will need to re-build updated graphics drivers for each minor update of RHEL 6 (annually or semi-annually). If you are working on a graphics driver, it is highly likely that you would have an update to the driver more frequently than annually in any case.

- **ISCSI.** Many ISCSI interfaces are not considered part of kABI due to the amount of upstream development. This may change in a future update. In the interim, you may need to rebuild ISCSI drivers in future minor updates. Again, this means annually or semi-annually.

- **SAS.** Many SAS interfaces are not considered part of kABI due to the amount of upstream development. This may change in a future update. In the interim, you may need to rebuild SAS drivers in future minor updates in the same fashion as you would for ISCSI or ATA.

- **Sound.** Certain sound drivers utilizing the "HDA" interfaces may need to be rebuilt for future minor updates of RHEL 6. Generally speaking, this is not likely to be a considerable issue for you in practice.

Contact Red Hat Partner Engineering for more information about kABI limitations that may affect your specific class of driver and your specific requirements, but please do so only after referring to the following chapter (which includes information about requesting additions to the kernel ABI).

If you do not have a Partner Manager, please file a Red Hat Bugzilla "bug" (bugzilla.redhat.com) or contact the author of this document directly for advice. Please note in any case that Red Hat cannot re-engineer your driver on your behalf because it fails to meet certain kernel ABI expectations.
2.3.1 Tracking kernel ABI changes

Periodic additions are made to the kernel ABI. These are published in each RHEL 6 minor update by way of changes to the kabi-whitelists RPM package, and generally are also available at the URL given in the Further Resources section of the introductory chapter. Red Hat does not, in general, remove interfaces from the kernel ABI once they have been added.

Red Hat is working on a notification mechanism to track the requirements of certain third party modules using non-kABI interfaces and provide warning of possible incompatibilities. To obtain further information about this, contact the author of this document to be signed up to a list of interested parties. General availability of such a service may occur in the future.

*NOTE*: Red Hat Kernel ABI is not supported in Fedora, or on other Linux systems not shipped by Red Hat. This is because the "mainline" Linux kernel as distributed on kernel.org has no kernel ABI. Consequently, it is not possible to use Red Hat Driver Update packages on those systems. If you are not using a RHEL system, it may be necessary to re-compile additional drivers that have been added to your system on each system update.

2.4 Build Environment

Driver Updates allow a customer or user to have a consistent driver installation and user experience, and are automatically preserved across regular system updates. However, for proper functioning, it is necessary to use a suitable build environment, based upon RHEL 6 (and not Fedora, or any other Linux distribution), on the target architecture that will be supported (this typically means building 32-bit Intel x86 packages on a 32-bit release). It is not necessary for a customer installing a Driver Update to have such a build environment, this pertains only to development of the Driver Update.

When building a Driver Update, it is possible to choose from a variety of different kernel packages to build against. If your driver is conformant to the Red Hat kernel ABI (kABI), and if the kernel interfaces required have been on the kABI for some time, it may be possible to build a Driver Update that will also work with older releases of RHEL 6\(^2\). For example, if RHEL

\(^2\)This is considered an advanced use case, and you are welcome to contact the author of this document for advice, but typically Driver Updates are only intended for forward compatibility from the current release onward. Of course, in any case, such forward
6.2 were current, and you wanted to support RHEL 6.0 and RHEL 6.1 with the same Driver Update package, this may be possible as long as all of the required interfaces have been on the kernel ABI since RHEL 6.0.

2.4.1 Installing Red Hat Enterprise Linux 6

It is recommended to build all Driver Updates using the latest version of the RHEL 6 distribution\(^3\), even if you will be supporting older kernel versions (for which you should obtain the appropriate kernel-devel package separately, from Red Hat in that case, and install that kernel for use during builds). It is not recommended to use an old version of the RHEL 6 distribution because continual enhancements are made to the distribution that improve the build process for new Driver Updates. Unless you must use an older version, you will benefit from obtaining the latest version of RHEL 6 from Red Hat for this purpose. It is explicitly not recommended to build using beta pre-releases of RHEL 6 updates for production use at any time.

To begin installation, obtain a copy of Red Hat Enterprise Linux 6. If you are a partner or a third party with an existing relationship with Red Hat, this may be possible by contacting your partner manager for assistance. You may use a virtual machine to perform builds if they will be tested separately (it is recommended to have a second system available in any case, to test install-time use of your Driver Update package). Ensure that you elect to install the Development packages during installation, as these will be required later on (they provide the kernel source and related build tools).

Once the system is installed, verify that you have at least the following packages installed on your new RHEL 6 system:

- createrepo (required when building Driver Update Disks)
- kabi-whitelists (required as a reference for official kABI)
- kabi-yum-plugins (optional enforcing plugin to mandate kABI)
- kernel-devel (older releases of RHEL had multiple possible kernel "variants" such as kernel-xen-devel, but these do not exist in RHEL 6).

\(^3\)This means the tools, utilities, and other "user space" components of RHEL 6. For example, if RHEL 6.2 is the current release of RHEL 6, you might install an older kernel development package for building Driver Updates, but the distribution is still RHEL 6.2.
2.4. BUILD ENVIRONMENT

- redhat-rpm-config

You can use the following command to verify that a package is installed:

```bash
rpm -q PACKAGE_NAME
```

For example, to verify that the "createrepo" package is installed:

```bash
rpm -q createrepo
```

If it is not installed, you can use the "yum" command to install it:

```bash
yum install createrepo
```

It is also possible to use graphical package management tools on the desktop to verify that particular packages are installed on your system.

Congratulations. You are now ready to begin building your Driver Update packages on RHEL 6. Be sure to keep your RHEL 6 system up to date with fixes for bugs and security problems (including bugs affecting the building of Driver Update packages) by subscribing it to the Red Hat Network (RHN). If you have a Red Hat Partner Manager, it is also advisable to contact them and let them know that you are working on a Driver Update of your own. There may be updates to this and other documentation that they can inform you about, and Red Hat can add your driver to an internal list of known Driver Updates, along with your general contact information.

2.4.2 Configuring rpmbuild

To actually build Driver Update RPM packages (in the absence of a more complicated Koji or mock based solution), it is necessary to configure the rpmbuild environment. By default, the rpmbuild command will use the system level /usr/src directories for building RPMs. This requires special permissions and is often undesirable. Instead, you should generally build Driver Update RPMs as a regular system user, in a working directory.

As an example, create the following directories within your home directory (the "$" character delimits the standard system prompt in this example):

```bash
$ mkdir -p ~/rpm/BUILD
$ mkdir -p ~/rpm/BUILDROOT
$ mkdir -p ~/rpm/RPMS
$ mkdir -p ~/rpm/SOURCES
$ mkdir -p ~/rpm/SPECS
$ mkdir -p ~/rpm/SRPMS
```
Then, configure RPM to use this "rpm" directory rather than the global system-wide default build location by creating your own .rpmmacros file (end the "cat" command by typing CTRL-D on your keyboard):

```
$ cat > ~/.rpmmacros
%home %(echo $HOME)
%_topdir %{home}/rpm
```

After performing this modification to the local user RPM configuration, you will be able to create RPM "SPEC" files in the rpm/SPECS directory and have the resultant RPM packages after an rpmbuild run be created in the appropriate subdirectories of the "rpm" directory that was just made.

The following section provides optional material of interest to certain developers supporting very large production systems that will build many different drivers and packages. It may not be relevant to your needs.

### 2.4.3 Using Koji and mock

It is possible to use the mock "chroot" build utility or the Koji build system that is build upon it and has been popularized by its use in Fedora (Red Hat uses an internal version of Koji called Brew for building its own RHEL 6 packages). Mock is a utility that creates a "chroot" environment containing all of the correctly versioned build dependencies for a given package, and produces binary output packages. You can find more information about mock by visiting its website at http://www.fedoraproject.org/wiki/Projects/Mock, or about Koji by visiting its website at https://fedorahosted.org/koji.

The benefit to using mock in large scale production (such as within an environment like Red Hat itself) is that it generates consistent builds every time, irrespective of local changes that might happen to packages on a given installed system (updates, local admin changes, etc.). This allows a build to be totally isolated from many typical influences. But this being said, most third parties will build on a specifically installed RHEL 6 system, often favoring virtualization as a means to provide a dedicated build system.

You might install a local instance of Koji if your company or organization will be building many different RPM packages, in addition to Driver Updates. This will provide you with a convenient web interface for submitting your builds and for watching the results of a build. While convenient, it is not necessary to have such a web-based environment to build Driver
2.4. BUILD ENVIRONMENT

Updates, and a regular build environment on a local workstation or virtual machine should be more than sufficient in most cases. In addition, a local environment allows a more hands-on level of interaction with individual builds.

If you are working on community supported Driver Update packages and want to avail of the Koji facility in Fedora, it is also possible to compile drivers as part of the "EPEL" (Extra Packages for Enterprise Linux) project.
CHAPTER 2. BACKGROUND REQUIREMENTS
Chapter 3

Red Hat Kernel ABI

Red Hat provide a value-add to Red Hat Enterprise Linux in the form of a stable kernel ABI (kABI). In this chapter, you will learn what a stable kernel ABI means (and what it does not), and will discover some of the tools and processes available to you when working on the RHEL kernel. You will also learn how to request additions to the kernel ABI using Red Hat Buzilla.

Your experience with kABI and requesting additions to it should not slow down your driver development, which can continue in parallel. It is anticipated that you will skim this chapter on a first reading, proceed to packaging in the next, and then return to use this chapter as a reference in ensuring kABI compliance. After all, you cannot truly know all of your kABI needs until you have a working driver that has been tested on RHEL 6.

3.1 What is the kernel ABI?

An Application Binary Interface (ABI) is a binary-level implementation of a specific Application Programming Interface (API), such as that provided by the various exported (for third party module use) functions of the kernel. Unlike an API, an ABI includes strict versioning information, structure layout, function calling and register use, and the like. An ABI is used by pre-compiled code (such as third party drivers) to ensure that it will operate as intended. A stable ABI means just that, an ABI (or portions thereof in the case of the kernel) that does not become incompatible over time.

Linux kernels already feature support for tracking the binary versions of various interface functions, known as symbol exports. Each interface that
CHAPTER 3. RED HAT KERNEL ABI

will be used by loadable kernel modules is marked as such using a macro called EXPORT_SYMBOL (which may actually take the form of a similarly named macro variant). For example, the kernel contains a function known as printk, used by modules and other kernel code to write to the system logs. During kernel build, special scripts calculate a checksum of the interface exposed by the printk function. When module files are later compiled, automated dependencies can be created upon these checksums.

In most Linux kernels, interfaces (and thus the interface checksums) will change between one build and the next as developers make various changes to the underlying code. Red Hat kernel ABI modifies the internal development process such that incompatible changes are not allowed to those interfaces that are determined to form part of the desired kernel ABI. Not every interface within the kernel (of which there are over 8,000) is considered to be part of the kernel ABI, since some interfaces are intended only for use within the kernel itself, are never really exposed to the outside, or have alternatives better suited for use by third party modules. Some other interfaces occasionally cannot be supported as part of kABI, even after a request is made to consider them as such. This was briefly mentioned in the last chapter and affects a few interfaces under heavy upstream development.

The Red Hat kernel ABI is represented in a series of lists of kernel interfaces (symbols) that are considered to be stable (and are deliberately maintained as such) for third party driver use. This facilitates an ability to compile a driver once and run it on a wide range of RHEL 6 systems, without having to frequently rebuild the driver from source code or even to have build tools installed on the end user’s system. As far as the user is concerned, they can install one package that provides a driver, and that driver keeps working. Internally, the kernel ABI lists are tracked to include the specific binary version of the interface as it was when it was first added to the kABI. These binary checksums form part of the RPM dependency data both on the RHEL 6 kernel package, and on Driver Updates built for it.

NOTE: Interfaces (symbols) not on the kernel ABI lists are allowed to change in minor updates to RHEL 6. This facilitates Red Hat kernel engineering in further developing the RHEL 6 kernel. When an interface is added to the kernel ABI, it is added as of a particular kernel release (typically at the granularity of a minor update release in which it is first included). The addition does not apply retrospectively to older kernels and so there may be different versions of the same symbol already deployed. Your driver is only
3.2 OBTAINING THE OFFICIAL KABI LISTS

guaranteed to be compatible with systems running a version of the kernel at or after the point that an interface became an official part of kABI.

Red Hat maintains a variety of internal tools and processes that preserve the kernel ABI (kABI). Red Hat kernel engineers must be aware of the existing semantics of the kernel and of the kernel ABI when making changes in minor product updates so that they do not (inadvertently) affect compatibility with existing third party driver modules. This takes skill and effort, but the payoff is that you as a third party packaging a driver do not have to worry about the RHEL 6 kernel being a moving target to develop against.

3.2 Obtaining the official kABI lists

The official reference for the Red Hat kernel ABI is the kabi-whitelists RPM package that comes with Red Hat Enterprise Linux 6. If you do not have this package installed, you can install it using the following command:

```
yum install kabi-whitelists
```

The package provides various files in the /lib/modules/kabi directory of your RHEL 6 filesystem. These have names such as the following:

- `kabi_whitelist_i686` - The kernel ABI for 32-bit PC systems
- `kabi_whitelist_ppc64` - The kernel ABI for POWER systems
- `kabi_whitelist_s390x` - The kernel ABI for System 390 systems
- `kabi_whitelist_x86_64` - The kernel ABI for 64-bit PC systems

Each of these kABI reference files targets a specific RHEL 6 supported architecture, such as the Intel PC or the IBM POWER. The files consist of a list of kernel symbols that are "on the kernel ABI" as of the date of the kabi-whitelists package. Important changes to the kABI will be documented within this package in the future, for those who are interested to know when a specific symbol was added to the kernel ABI\(^1\). Generally speaking, if an interface is added to the kernel ABI, it is added for all architectures for which it is available, although exceptions do occur. This means that you should verify your kABI use for all architectures you will be supporting.

\(^1\)Look for a "README" file that will be added in future minor updates to RHEL 6.
A canned version of the kernel ABI is also maintained at the following URL: http://people.redhat.com/jcm/el6/kabi/. In the event of any discrepancy between the version contained on that website and the content of the kabi-whitelists RPM package, the content of the package shall prevail as being the canonical reference. Please contact Red Hat if you have any concerns.

You can read the kABI reference files using a simple text editor, and use them as a reference in determining whether your driver is compliant with the Red Hat kernel ABI. Tools to aid with that are documented in the next section on kABI verification, entitled "Verifying kABI requirements".

### 3.3 Verifying kABI requirements

Now that you know what the kernel ABI is, you may be interested to determine whether your driver(s) is/are compliant with the Red Hat kernel ABI. The following tools and techniques can be used in order to quickly ascertain this information. A later section will provide information about requesting additions to the kernel ABI, if that is subsequently required.

#### 3.3.1 Using the command line

It is possible to check for the symbols used by a particular Linux kernel driver (module) that has been built against a particular RHEL 6 kernel using the pre-installed "modprobe" command that is part of the (pre-installed) "module-init-tools" system package on your RHEL 6 system.

To see a list of symbols required by a pre-compiled kernel module file (whether part of kernel ABI or not), you can use the following command:

```bash
modprobe --dump-modversions /path/to/driver/module.ko
```

This will list all of the kernel symbols required of the driver module, along with the version that is required (which comes first in the output). Note that it is important to give the path to a particular compiled kernel module when using this option, which differs from some regular uses of the modprobe command in which the path to the module is determined automatically.

If you have already built a Driver Update package (as documented in the next chapter), and are now interested in kABI compliance, you can also use the following RPM command to list each of its kernel dependencies:
3.4. WORKING WITH THE KABI

```bash
rpm -qp --requires /path/to/kmod-driver-update.rpm
```

The output will include (amongst other details) a list of kernel symbols required by your driver. You engineers will easily recognize the appropriate symbol requirements output data from this command.

With the appropriate symbol requirements obtained either through modprobe, or through RPM, you can verify these symbols against the official kernel ABI lists. You can also use either of the abi-check or kabi-yum-plugins Yum plugin package to automate this check against the latest installed version of the kABI reference lists on your build system.

### 3.3.2 Using the abi-check script

Red Hat supplies a script called `abi_check.py` on the website referenced in the Introduction under “Further Resources” that can be used to automatically verify compatibility between a kernel module and the RHEL 6 kernel ABI. That script is presently being updated to operate correctly with RHEL 6 and should be available shortly. It is a simple standalone python script.

### 3.3.3 Using the kabi-yum-plugins package

Red Hat supplies an (optional) Yum package manager plugin known as kabi-yum-plugins (plural because there may be additional functionality added in the future). The intention of this package is to allow certain users to mandate that only kABI compliant drivers will be installed on their systems, but it can also serve a useful verification step if you are seeking the same.

After installing the package, you will receive a warning in the system log (syslog) files whenever a package that fails a kABI check is installed. It is also possible to configure the package (using its configuration file in the `/etc` directory) to automatically fail to install in the case that a Driver Update package does not conform to kernel ABI. This is not the default, since it is quite common for Driver Updates to be installed that require one or more kernel symbols that are not on the official kABI reference lists.

### 3.4 Working with the kABI

The implementation of a stable kernel ABI in Red Hat Enterprise Linux provides you and your engineers with many benefits. You are able to know that
certain kernel interfaces will not change, and can expect certain behavior to be preserved even in the face of future kernel code changes. The kernel ABI process does place some constraints upon you, as a driver author, however.

3.4.1 Appropriate use of kernel interfaces

The kernel ABI can only be correctly preserved if third party drivers are written to make use of it following typical conventions for Linux kernel code. This means that you must use correct locking primitives, must always call any kernel-provided initialization function, and generally cannot write your driver to intentionally work contrary to typical Linux kernel conventions. If you would like advice, please contact the author of this document. Depending upon the nature of your inquiry, we may be able to help you.

NOTE: The RHEL 6 kernel receives updates from time to time, especially in minor releases. These updates can make changes that will affect existing modules when they are re-compiled, for example introducing a new, replacement, or updated interface dependency, or other change. This is not considered to be a "kernel ABI break" ("kABI break") because the existing, pre-compiled driver would have continued to operate were it not recompiled. It is entirely reasonable that recompiling a driver may introduce small changes. To avoid any negative experiences associated with on the fly rebuilds, refrain from shipping tools that recompile drivers on user systems\(^2\), and use the Driver Updates packaging process to ship your drivers.

3.4.2 Initializing and handling memory

The kernel ABI locks down certain kernel structures and does not allow them to change in incompatible ways. Even adding a new field to the end of an existing structure can affect kernel ABI because many structures are embedded within other structures, and a change to one will affect the other. Certain sophisticated mechanisms can be employed to work around this, and of course it is easier to add a new field to the end of a non-embedded structure. Red Hat sometimes does this in minor update releases.

You should always use the correct allocation mechanism for the memory regions and structures you intend to use. If there is a core kernel provided function that is generally always used to allocate a particular structure, then

\(^2\)These seem attractive, but are highly unsupportable on a large scale, and are strongly discouraged for many reasons, not the least of which is repeatability of code compilation.
it is reasonable for Red Hat kernel engineers to assume that you are using
that function when they need to add new fields to existing structures. A
similar situation applies in freeing structures in the most appropriate way at
all times. Failing to follow this advice can result in compatibility problems.

NOTE: Some structures contain members with names like rh_pad. These
have been added for the benefit of Red Hat’s kernel engineers, for future
possible use in avoiding breaks to the official kernel ABI. These are not
available for use by your driver, at all. Please do not use them as they are
reserved exclusively for the use of Red Hat. Additionally, please do ensure
that, wherever possible you initialize structures with a call to memset, or
use a kernel function such as kzalloc to ensure structures are zero initialized,
according to the appropriate best practice for the kernel subsystem involved.

3.5 Requesting additions to kABI

Sometimes, a particular kernel interface that is required by your kernel driver
is not considered to be part of the kernel ABI at the time you are wanting
to avail yourself of it. This can occur for any number of reasons, including
(but not necessarily limited to) the following:

• The interface (kernel symbol) you require is not considered stable by
Red Hat. This means that there is significant activity in this area
of the kernel upstream (in the “mainline” kernel.org kernel) and it
is not clear yet what changes may be required in a future update.
Typically, such symbols are stabilized within a few minor updates,
but we may not have an accurate timeframe for this due to the way in
which upstream development occurs independently of such controls.

• The interface has been added in a RHEL 6 kernel update (in a minor
release) in order to enable a new feature that has also been added to
the kernel. It may be that Red Hat has yet to receive a request for
this new kernel interface to be added to the kernel ABI\(^3\). Sometimes,
a relatively small change to the kernel in a minor update can add
(for example) a new string manipulation function that becomes widely

\(^3\)However, a new tracking system is being developed to flag these up for consideration
automatically, which should help but will not eliminate the need for you to request the
addition of a new interface to the kernel ABI. Even with a system flagging up potentially
problematic interfaces (symbols), it is still often necessary to know that they are actually
being used due to the otherwise large number of false positives that could be triggered.
used. In such cases, it is usually straightforward to update the kABI. This is a good reason for working with pre-release betas of RHEL 6 updates to ensure that your driver does not (inadvertently) require such an interface addition, recalling that such use does not need to be explicit and may happen by way of other called interfaces or macros in an existing driver code base not recently modified for RHEL 6.

- The interface has been explicitly flagged as unsuitable for third party driver use by the Red Hat kernel engineering team. It may be that there is a preferred interface that should be used instead. In this case, Red Hat may be able to make some recommendations for changes to your driver that will allow it to make use of official kABI.

There are occasionally other reasons to exclude kernel interfaces from the kernel ABI. Remember, the official "mainline" Linux kernel does not have a kernel ABI, and this is an addition made by Red Hat. Consequently, it is not possible to support all desired situations or uses of kABI. You are free to use interfaces not considered part of kABI, but you may have to rebuild your driver for subsequent minor releases of RHEL 6 (on an annual or semi-annual basis perhaps - but generally not for security or errata updates).

To request an addition to the kernel ABI, you should contact Red Hat Partner Engineering. They will direct you to file a bug in the Red Hat Bugzilla (bugzilla.redhat.com), containing a list of kernel symbols (interfaces) that you would like have added to the kernel ABI\(^4\), along with a justification for each of them, a summary of the purpose and function of your driver, and any necessary contact details. It is not necessary that you supply Red Hat with source for your driver directly\(^5\), since you are responsible for your own licensing obligations when it comes time to actually ship your product.

NOTE: The appropriate Bugzilla template for filing a kernel ABI request against the "Driver Update Program" component in Bugzilla is available from your Red Hat Partner Manager. If you do not have a Red Hat Partner Manager, please file a bug against the "Driver Update Program" component

\(^4\) kernel ABI is not always architecture specific insomuch that particular interfaces may be common to many different architectures. However, for various reasons, it is necessary that you supply information about your intended target platform.

\(^5\) Driver packaging work often takes place concurrent with ongoing driver development, and of course may involve drivers for hardware that has not been announced. We understand that you may not wish to share details or source for unreleased products, but we do require certain information in your request.
in Bugzilla, under the RHEL 6 product. Include as much detail as you can about your driver and your request will be reviewed. You may be asked to fill in a template request form at that time if further information is required.

Once a request has been made to add a symbol to the kernel ABI, it will be passed to internal review and considered for inclusion into the kabi-whitelists package (and official kABI) as of the next minor update release. For example, if you request an addition to kABI in RHEL 6.0 then it will be reviewed for inclusion into RHEL 6.1. This does not mean that you cannot use the affected symbol in the interim, only that it is not guaranteed against future changes in a minor update until it is added to the official kABI.

3.6 Kernel ABI Breakage

Red Hat generally undertakes not to break the kernel ABI for RHEL 6. Once a kernel interface (symbol) has been added to a kABI reference list, it will not be changed for the duration of RHEL 6, with the exception that very rare and special circumstances may force such action. For example, a critical security problem that cannot be resolved without breaking kABI would be an example of a situation in which we would introduce kABI breakage. At that time, Red Hat would generally inform partners of the problem.

It is not considered kABI breakage that re-compiled drivers occasionally have newer symbolic dependencies or other subtle changes caused by changes to the RHEL 6 kernel source since the last build (another reason not to build on the target user system). It is also not considered kABI breakage when interfaces not on the kABI (not in the kabi-whitelists package) are changed in kernel updates. Although Red Hat generally endeavors not to make changes even to non-kABI interfaces in regular security and errata updates, some changes are (rarely) required. If you believe you have experienced genuine kABI breakage, please contact Red Hat immediately.
Chapter 4

Driver Update Packages

Red Hat Driver Updates are a special form of RPM package that contain one or more Linux kernel module(s) that is/are compatible with the RHEL 6 kernel. These packages are sometimes referred to as "kmods" because they contain the word "kmod" in the name, although the official name is "Driver Update". During the development of RHEL 6, an effort was made to standardize the packaging of Driver Updates between distributions, and so there are some (minor) differences from the packaging used in RHEL5.

An example Driver Update package may be named:

kmod-sym53c8xx-2.2.3.rhtest60-1.el6.x86_64.rpm

This example targets the "sym53c8xx", a popular SCSI controller that is emulated by the "KVM" virtual machine hypervisor, and thus proves to be a widely usable and also useful test case for Driver Updates. You can obtain this package from the website location given in the "Further Resources" section of the Introduction. Driver Updates always begin with "kmod", and always contain an extension that includes the el6 release and the system architecture that the package is being provided for. It is explicitly not recommended to attempt to change this naming convention in your packages.

A particular Driver Update package provides drivers that are built against a certain release of the RHEL 6 kernel. In the case of the example package, it was built against the 2.6.32-71.el6 "GA"1 RHEL 6.0 kernel, as can be

---

1"General Availability", another term for the initially released kernel that came with RHEL 6. Updates to this kernel in the form of errata and security fixes also have a version of 2.6.32-71, but may have an additional sub-version number appended to identify the particular fix that has been applied.
ascertained by using the following command:

```
rpm -qpi kmod-sym53c8xx-2.2.3.rhtest60-1.el6.x86_64.rpm
```

The output includes the following:

```
This package provides the sym53c8xx kernel modules built for
the Linux kernel 2.6.32-71.el6.x86_64 for the x86_64
family of processors.
```

The package can be installed on the 2.6.32-71.el6.x86_64 kernel that orig-
inally came with RHEL 6.0, and also updates to RHEL 6.0 in the form
of security and errata fixes. When the Driver Update package is installed,
special system scripts will detect all compatible RHEL 6 kernels, and will
ensure that the module is made available to those kernels automatically,
whether they include the original 2.6.32-71 kernel release or not. When
subsequent kernel updates are applied to the system, the modules will be
similarly preserved, as long as those kernel updates are compatible. This
happens automatically and does not require any user involvement.²

A Driver Update package contains a number of files, which can be displayed
using the following command:

```
rpm -qpl kmod-sym53c8xx-2.2.3.rhtest60-1.el6.x86_64.rpm
```

The output may resemble the following:

```
/etc/depmod.d/sym53c8xx.conf
/lib/modules/2.6.32-71.el6.x86_64
/lib/modules/2.6.32-71.el6.x86_64/extra
/lib/modules/2.6.32-71.el6.x86_64/extra/sym53c8xx
/lib/modules/2.6.32-71.el6.x86_64/extra/sym53c8xx/sym53c8xx.ko
```

The example includes the following two files (the remainder are directories
that are "owned" by the package for the purpose of system installed package
file ownership tracking - facilitating ease of removal, and so forth):

²This includes the building of initramfs "ramdisks", handling of firmware files, and so
forth. It is not necessary and is explicitly not intended for you to in any way rebuild
these images in your driver packages, run scripts at system startup that attempt to do
so, or otherwise modify the system beyond providing the driver module(s). If it appears
necessary to take these steps, please file a bug or request further information from Red
Hat concerning best practices for the building and distribution of Driver Updates.
4.1. **PREPARATION**

- sym53c8xx.conf - This is a system-level configuration file for the "depmod" module utility that specifies the relative priority of this driver compared with the existing system driver. This is needed when replacing drivers that are already provided by Red Hat Enterprise Linux.

- sym53c8xx.ko - This is the actual Linux kernel module, installed under the "extra" module subdirectory of the kernel it was built against. This special location is recognized and reserved for Driver Updates. Compatible kernels will automatically have symbolic links created within "weak-updates" subdirectories of their module directories.

Installing the Driver Update is a simple matter of running a regular RPM command, either on the command line, or using the graphical interface. As you will learn later, Driver Updates may also be installed during system installation by means of Driver Update Disks, and may also be distributed using Yum repositories (using the "yum install" command, or the interface).

The following section describes how to build your own Driver Update, assuming that you have already configured your system as described in chapter 2 ("Background Requirements"). If you have not, please return to that chapter for instructions now, and proceed once your build system is ready.

### 4.1 Preparation

To begin building a Driver Update package, ensure that you have read the chapter "Background Requirements" and that you have configured RPM as described to use the local user "rpm" directory for its output. Additionally, you should already have a working Linux kernel driver module to test. If you do not have one, download one of the examples from the "Further Resources" section of the Introduction and use that as working source instead.

In the following section, you will create a reference SPEC file for your driver and then build it from source. It is recommended to actually create these files for yourself, especially if it is the first time that you have ever made an RPM package. This will aid your understanding. However, if you run into any problems, or if you are pressed for time, you can also install the example Source RPM (SRPM) package from the website previously referenced. A Source RPM is a special kind of RPM package that really contains only files to build an RPM, and it doesn’t require special privileges to install it to a location such as was configured by the .rpmmacros file in chapter 2.
To install the sym53c8xx-2.2.3.rhtest60-1.el6.src.rpm SRPM (notice the lack of "kmod" for the SRPM), you could type the following:

```
rpm -ivh sym53c8xx-2.2.3.rhtest60-1.el6.src.rpm
```

When the rpm command is run, it uses the .rpmmacros file that you created previously, and recognizes that it is working on a source package. Thus you do not require any special system privileges to "install" (which really, for source packages, simply means to unpack the package file) such source RPMs. The files from the SRPM will be placed into the rpm directories previously created under "Background Requirements" in chapter 2. After installing the SRPM you would see a file named sym53c8xx.spec in the rpm/SPECS directory, along with the source tarball and supporting files in the rpm/SOURCES directory as discussed in the following section.

Before proceeding with packaging, you may like to have a general RPM packaging reference available. There is an official RPM book called "Maximum RPM" available on the rpm.org website, and some further useful information contained within the Fedora "How to create an RPM package" tutorial guide located on the Fedora project website:

```
http://fedoraproject.org/wiki/How_to_create_an_RPM_package
```

These documents cover everything about building RPM packages, except the specific %kernel_module_package Driver Updates RPM macros that are described in the following section.

### 4.1.1 Kernel driver source

It is assumed that you have your kernel module source contained within a regular UNIX "tarball" (in this case, actually a bzip2 "tarball") of the form sym53c8xx-2.2.3.rhtest.tar.bz2 (the name of course varying for your module), and that this is already installed into the rpm/SOURCES directory that RPM will search when attempting to locate your specified source. The example tarball actually contains the following module source code files:

```
sym53c8xx-2.2.3.rhtest60/sym_hipd.h
sym53c8xx-2.2.3.rhtest60/sym_hipd.c
sym53c8xx-2.2.3.rhtest60/Makefile
sym53c8xx-2.2.3.rhtest60/sym_fw.h
sym53c8xx-2.2.3.rhtest60/sym_fw2.h
```
4.2. CREATING A DRIVER UPDATE PACKAGE

sym53c8xx-2.2.3.rhtest60/sym_m misc.h
sym53c8xx-2.2.3.rhtest60/sym_defs.h
sym53c8xx-2.2.3.rhtest60/sym_nvram.h
sym53c8xx-2.2.3.rhtest60/sym_fw1.h
sym53c8xx-2.2.3.rhtest60/sym_glue.h
sym53c8xx-2.2.3.rhtest60/sym_glue.c
sym53c8xx-2.2.3.rhtest60/sym_nvram.c
sym53c8xx-2.2.3.rhtest60/sym53c8xx.h
sym53c8xx-2.2.3.rhtest60/sym_fw.c
sym53c8xx-2.2.3.rhtest60/sym_malloc.c

These are regular Linux kernel source files, with the addition of a Makefile to drive the kernel’s "Kbuild" system when the RPM source runs “make”. The example Makefile contains only the following content:

# Makefile for the NCR/SYMBIOS/LSI 53C8XX PCI SCSI controllers driver.

sym53c8xx-objs := sym_fw.o sym_glue.o sym_hipd.o sym_malloc.o sym_nvram.o
obj-m := sym53c8xx.o

When running the kernel build system, this states that the sym53cx8xx driver module is built from various other object files, and is itself a module. More information about the format and process of building external kernel modules is documented in the "Documentation" subdirectory of the Linux kernel sources, which is included within RHEL 6 (but is not unique to RHEL 6, this is standard Linux kernel practice).

4.2 Creating a Driver Update package

Driver Updates are built from a SRPM (Source RPM) that contains a tar-ball archive of the Linux kernel module source, and an RPM "SPEC" file that describes how to build the resultant binary RPM package from the SRPM. Typically, the Source RPM is built automatically at the same time as the binary RPM package, using the same rpmbuild command. You might already have used the SRPM output from a previous build to save time installing the example files, but you could just as easily have placed them into the rpm/SPECS and rpm/SOURCES directories without using the SRPM.

RPM works by using various macros in a SPEC file to inform the build of the name of the package, its license, source files, and so on. In the case of Driver Updates, a special set of RPM macros allows you to avoid some of
CHAPTER 4. DRIVER UPDATE PACKAGES

the specifics of the Driver Updates mechanism and produce clean, portable
Driver Update packages. The special macros will be discussed when they
are used. Most of the following example uses completely standard RPM
macros as documented in the references given previously.

To build your Driver Update package, begin by creating a "SPEC" file in
the rpm/SPECS directory. Use the same name as your driver. The exam-
ple SPEC file that follows is named "sym53c8xx.spec". Then, populate the
SPEC file with content from the following example:

Name: sym53c8xx
Version: 2.2.3.rhtest60
Release: 1%{?dist}
Summary: Example RHEL 6 Driver Update Program package

Group: System/Kernel
License: GPLv2
URL: http://www.kernel.org/
Source0: %{name}-%{version}.tar.bz2
Source1: %{name}.files
Source2: %{name}.conf
BuildRoot: %(mktemp -ud %{_tmppath}/%{name}-%{version}-%{release}-XXXXXX)
BuildRequires: %kernel_module_package_buildreqs

# Uncomment to build "debug" packages
# %kernel_module_package -f %{SOURCE1} default debug

# Build only for standard kernel variant(s)
%kernel_module_package -f %{SOURCE1} default

%description
An example RHEL 6 Driver Update package.

%prep
%setup
set -- *
mkdir source
mv "$@" source/
mkdir obj
4.2. CREATING A DRIVER UPDATE PACKAGE

%build
for flavor in %flavors_to_build; do
  rm -rf obj/$flavor
  cp -r source obj/$flavor
  make -C %{kernel_source $flavor} M=$PWD/obj/$flavor
done

%install
export INSTALL_MOD_PATH=$RPM_BUILD_ROOT
export INSTALL_MOD_DIR=extra/%{name}
for flavor in %flavors_to_build; do
  make -C %{kernel_source $flavor} modules_install \
    M=$PWD/obj/$flavor
done

install -m 644 -D %{SOURCE2} $RPM_BUILD_ROOT/etc/depmod.d/%{name}.conf

%clean
rm -rf $RPM_BUILD_ROOT

%changelog
* Thu Dec 23 2010 Jon Masters <jcm@redhat.com>
  - An example Driver Update Package

The SPEC file contains various entirely standard RPM "preamble" data, such as the "Name", "Version", "Release", "Summary", licensing information, and so on. The Source lines reference source files contained within the rpm/SOURCES directory, including the driver source tarball (Source0), a file that contains a list of files to be included in the Driver Update package (Source1) - done this way in order to aid the special macros used next do some special processing, but not typical of most RPM packages - and a configuration file for the "depmod" system utility (Source2).

Most of the "magic" of Driver Update packaging happens by way of the special %kernel_module_package RPM macro. This will be documented shortly. The remaining sections of the RPM SPEC file are standard (with the exception of the use of the %flavors_to_build and %kernel_source macros about to be explained), including the %prep, %build, and %install sections. These unpack the source for your kernel module from the source "tarball" archive, build it as an external module against the currently installed kernel-devel
package, and install the result into a special temporary build-time directory through the setting of special Linux kernel "Kbuild" environment variables INSTALL_MOD_PATH, and so on. These environment variables are part of standard upstream building practices for Linux kernel modules built outside of the kernel, whether Driver Updates or otherwise.

*NOTE*: RPM packages have various conventions for naming. A later section of this chapter details some of the additional conventions for driver updates. You are encouraged to name the "Version" according to the version given in the driver source. In the example, 2.2.3.rhtest60 refers to a Red Hat internal test version of the 2.2.3 version of the sym53c8xx driver as shipped in RHEL 6. The Release must always be incremented for each new build of the same version of the package to distinguish one build from the next.\(^3\)

The following contains mostly optional documentation on the `%kernel_module_package` macro. On a first reading, you can skim most of this information, focusing only on the general use of the macro shown in the example, and then proceed to the following section on building the RPM. For those who are curious, or who have more advanced packaging requirements, the following may be of more interest. Additional documentation on the design and implementation of this macro may be added in an appendix to a future version of this guide.

### 4.2.1 The `%kernel_module_package` macro

It was previously noted that most of the "magic" behind building Driver Update Packages is actually abstracted by the special RPM macro that was added for the purpose. This `%kernel_module_package` macro has been standardized between several Linux distributions (hence the slightly non-intuitive name) and has some hidden complexities behind what appears to be a relatively simple usage (non-standard parameters are not shown):

```
%kernel_module_package [ -f filelist ] [ -p preamble ]
                        [ -s script ] [ -x ] flavor1 flavor2
```

The macro requires only one mandatory parameter, and that is the name of the flavor (or kernel variant) to build against. In RHEL 6, typically, there

\(^3\)It is extremely important not to ever remove the "{%dist}" component of the "Release". This is responsible for generating the ".el6" in the package name, which is required for RHEL 6 packages. It may seem tempting to modify this to "el6_0", "el6_my_driver", or another name, but this will result in user experience problems later. Please do not change or otherwise affect this dist tag macro use, which is required.
4.2. CREATING A DRIVER UPDATE PACKAGE

is only one kernel variant (in RHEL5 there was also "xen", and so forth),
which in this context (again for standardization) is known as the "default".
Thus, the following is a minimal valid call to the macro:

`%kernel_module_package default`

Many practical Driver Updates will want to configure the module loader
in order to replace or override an existing module with the Driver Update.
Thus, it is very common to use the macro as in the example:

`%kernel_module_package -f %{SOURCE1} default`

With "%SOURCE1" being replaced by Source1, which is the filelist file
containing a list of all files that will be included in the driver update:

```
%defattr(644,root,root,755)
/lib/modules/%2-%1
/etc/depmod.d/sym53c8xx.conf
```

This allows for the inclusion of the "sym53c8xx.conf" file that configures the
module loading system. If you do not need to configure the module loader
(for example because you are shipping an entirely new driver), and you do
not otherwise have files to add (such as firmware) beyond the driver itself,
you do need this step and could skip the sym53c8xx.files Source entirely.

Use of a filelist overrides the default, implied filelist that is generated in
the absence of this parameter, and thus explains why the supplied filelist
must contain additional entries (as shown in the example) in addition to
those that must be added for the module configuration. You might won-
der why you cannot use a normal %files section as in other RPM packages.
This is because the %kernel_module_package macro actually creates what are
known as sub-packages, possibly more than one binary RPM package begin-
ning with the "kmod-" prefix. Since sub-packages use a different filelist, any
%files section would actually attempt to include files in the wrong location.4

Here are other common parameters to the %kernel_module_package macro:

- `-f filelist` - A file containing a list of files to be included in the Driver
  Update package. By default, kernel modules are included in Driver

---

4There are various technical reasons why the packaging of Driver Updates is designed
in this way. The preferred solution is to simply apply the -f parameter as necessary. If
you interested in learning more, consult an RPM reference and read about the precise
implementation of RPM sub-packages to see why the %files section is seemingly ignored.
Update Packages. However, configuration and firmware files are not included by default, and so a filelist (containing a list of all the files that will be in the package, not just the additional files) of the form given in the example must be included when supplying additional files. You can use the example as an easy reference.

- \texttt{-p preamble} - This is rarely used, for example when customizing ”\texttt{\%pre}” and ”\texttt{\%post}” scripts that run before and after installation of a Driver Update. Generally speaking, Driver Updates should not perform a lot of system scripting (and you should not use these directives unless you understand how sub-packages work, and are able to apply them appropriately). Such scripts should instead be placed inside a higher level general package that has a shared dependency with the Driver Update and will cause it to be also installed. The general package can also contain any system tools and utilities that are needed to configure the driver. This allows the driver to be easily updated independently.

- \texttt{-s script} - A means to replace the internal ”\texttt{kmodtool}” script used by the Driver Update infrastructure in emergency situations wherein it is not possible to otherwise use the system-provided default. This is very rarely used in RHEL 6. It is intended only for use when a bug is found in the general packaging and a specific workaround is deployed for use in a particular Driver Update package Source RPM in the interim.

- \texttt{-x inversion} - This inverts the variants (”flavors”) of the kernel that will be built, causing any not specified to be targeted. This is a very rarely used option unlikely to be of interest in RHEL 6.

The use of the word ”default” with the \texttt{%kernel\_module\_package} macro arises because the same macro is present in multiple Linux distributions (it having undergone an agreed standardization), but RHEL does not have a specific name for the standard kernel variant. Hence, it was agreed to use ”default” for this macro for compatibility. In RHEL5, there were other variants like ”xen”, but RHEL 6 doesn’t need them. The ”default”, and other possible specified variants are referred to as ”flavors” in the case of the macros (again, for cross-distribution compatibility reasons - these are generally known as ”variants” in Red Hat kernel nomenclature). The \texttt{%flavors\_to\_build} and \texttt{%kernel\_source} macros encapsulate these cross-distribution differences in names and locations and are used during the build stage of the RPM packages, in the SPEC file. You can simply follow the example provided.
4.3. BUILDING A DRIVER UPDATE PACKAGE

The Driver Update macros generate special sub-package RPMs as output, which are related to the package defined in the SPEC file, but have slightly different names (for example kmod-foo binary RPMs instead of foo-kmod at the SRPM level). This makes no difference to you, unless you are attempting advanced modification of the RPM packages to include special "pre" or "post" scripts, in which case addition of a standard %pre or %post section in the SPEC file will not result in any changes to the binary. To use such scripts, investigate the %kernel_module_package "/-p" parameter.

4.3 Building a Driver Update Package

Once you have created a Driver Update Package by making a SPEC file, and by including source, as well as any optional configuration files and filelists, you are ready to actually build the binary Driver Update Package RPM. To do this, you will make use of the rpmbuild command, as in the following:

```
$ rpmbuild -ba rpm/SPECS/sym53c8xx.spec
```

This will cause the rpmbuild tool to parse the SPEC file and create both a binary (RPM), and a Source RPM (SRPM) through the use of the "/-ba" parameter (build all). You could also use:

```
$ rpmbuild -bb rpm/SPECS/sym53c8xx.spec
```

To build only the binary RPM package(s), and:

```
$ rpmbuild -bs rpm/SPECS/sym53c8xx.spec
```

To build only the Source RPM (SRPM) package.

The output from the build process will be placed in the rpm/RPMS and rpm/SRPMS directories, according to the build options that were given. You can install the resulting rpm package(s) just like any other Driver Update package. It is recommended to test your newly built driver update on a second test system, where you can use either an rpm command:

```
$ rpm -ivh kmod-sym53c8xx-2.2.3.rhtest60-1.el6.x86_64.rpm
```

Or a graphical tool to install the Driver Update.
4.4 Shipping an entirely new driver

When shipping a driver that has never appeared in the distribution before, you do not need to include an equivalent to the sym53c8xx.files Source given in the example. This is because the module loader does not need to be configured beyond its defaults in the case of an entirely new device driver. The default behavior is to load your new module, which does not conflict with an existing system module, and so no configuration is required.

NOTE: If your driver does replace functionality already provided by an existing driver, it is important to ensure that your driver will take precedence for affected hardware devices. You are in that case actually replacing an existing driver and need to ensure it will be used by the system driver utilities. See the section ”Replacing an existing driver” later in this chapter.

4.5 Updating an existing driver

Updating an existing driver follows exactly the process described in the "Building a Driver Update" section of this chapter. The sym53c8xx driver in the example updates the existing driver supplied by the kernel. The Driver Update includes a sym53c8xx.conf file containing the following commands for the "depmod" module management utility:

```bash
override sym53c8xx 2.6.32-* weak-updates/sym53c8xx
```

The "override" command, which is documented in the "depmod" manpage takes a module name (sym53c8xx, without the ".ko" extension), a kernel version wildcard to match against, and a subdirectory containing the preferred path to the module. The "weak-updates/sym53c8xx" reference to "weak" relates to the fact that driver updates are "weakly" coupled to the kernel they load against, as opposed to the drivers and other modules that came shipped with the kernel, which by default take precedence as a matter of system policy.\(^5\). Driver Updates are always installed into a (specified at build time) subdirectory of the system "extra" directory provided by the exact kernel they were built against, in this case named "sym53c8xx"\(^6\).

---

\(^5\)This policy can be changed at a global system level in the top-level depmod configuration file, but must never be changed by a third party driver. If your driver attempts to change global system policy, other than just for itself in its own configuration file, you risk seriously compromising system supportability for your users.

\(^6\)This is true whether or not the kernel the Driver Update was built against is actually installed. If the build kernel is not installed, the directory under /lib/modules for that
4.5. **UPDATING AN EXISTING DRIVER**

In the example, the sym53c8xx driver built against 2.6.32-71.el6 will be installed into the following location (whether the 2.6.32-71.el6 kernel is actually installed on the target system or not does not matter):

```
/lib/modules/2.7.32-71.el6.x86_64/extra/sym53c8xx/sym53c8xx.ko
```

The system will subsequently automatically install symbolic links to this driver into the "weak-updates" directories of additional, compatible kernels. For example, the sym53c8xx driver might actually be installed on a system running a 2.6.32-72.el6 kernel, in which case the previously compiled against 2.6.32-71.el6 sym53c8xx.ko module would still be placed into:

```
/lib/modules/2.6.32-71.el6.x86_64/extra/sym53c8xx/sym53c8xx.ko
```

The `/lib/modules/2.6.32-71.el6.x86_64` directory may otherwise be empty if the kernel is not installed. Since the driver in this example is compatible with 2.6.32-72.el6, which is actually the kernel installed, a symbolic link is automatically added by the system into the 2.6.32-72.el6 kernel:

```
/lib/modules/2.6.32-72.el6.x86_64/weak-updates/sym53c8xx/sym53c8xx.ko
```

This symbolic link points to the sym53c8xx.ko module file installed under the 2.6.32-71.el6 directory that it was built against. This behavior was chosen in order to remain broadly faithful to the way in which Linux kernel modules are otherwise installed, but may seem confusing on first inspection. It is this "weak-updates" directory that is being implicitly referenced in the configuration examples cited in this section, informing the module utilities that the compatible module should replace the standard module that came with the kernel\(^7\) (in this case in a separate kernel sub-directory).

**NOTE:** Actual system module loading policy is to favor local admin user updates in an "updates" directory (e.g. `/lib/modules/2.6.32-71.el6.x86_64/updates`), then "extra" modules, then kernel supplied modules, and finally "weak-updates" (the location of compatible Driver Updates). This ordering means that the most specific module for a given kernel always wins, but it also means that in the absence of a "depmod" configuration file such as `sym53c8xx.conf`, your Driver Update may appear to work normally if it is only tested with kernel version will contain only the Driver Update files. This is intentional.

\(^7\)Other distributions favor third party modules by default, but since Red Hat does not support third party modules, the default is to always use the kernel supplied official Red Hat driver if one is available
the exact same version of the kernel that it was built against, subsequently not being used when the kernel is updated. Always use a "depmod" configuration file in the case that you are overriding modules.

### 4.6 Replacing an existing driver

Replacing an existing driver is similar to providing an update for the driver, except that the existing driver must be disabled in the process. To do this, first ensure that your new driver provides the same basic functionality as the driver you are replacing and that it can (for example) support the existing hardware that the user has installed on their system, especially during system boot. Once you are confident that your driver really is a simple replacement, you need to disable the old driver. Add a new file

```bash
/etc/modprobe.d/your_module.conf
```

This configuration file should be listed in the `sym53c8xx.files` Source (obviously replacing "sym53c8xx.files" with the actual name of your driver) and, separately, that configuration file should actually be included as its own "Source" entry in the Driver Update SPEC file. The content of the file should include the following "modprobe" commands (not "depmod", because this "blacklisting" of the original driver module happens at module load time instead of at module installation time, as in the case of "depmod"):

```bash
blacklist original_driver
```

The blacklist command takes a module name (without the ".ko") and will prevent module utilities from attempting to load the old module unless they are manually instructed to do so. This means that during system boot, and at other times, the existing driver will not be automatically loaded by the system software. If your new driver supports the same hardware that the old driver did, and the user has this hardware installed, your replacement driver will load instead of the original system driver. Don’t forget that this change will be reverted when your driver is removed - as is correct. Your driver should not somehow otherwise attempt to in any other way blacklist an existing driver, especially not in a way not automatically revertible.

### 4.7 Firmware

Many drivers require firmware data be loaded into a hardware device before it can be properly used. It is common to use a built-in Linux kernel
mechanism, known as request_firmware to cause a named firmware file to be loaded from one of several possible system firmware locations:

- /lib/firmware/updates/KERNEL_VERSION
- /lib/firmware/updates
- /lib/firmware/KERNEL_VERSION
- /lib/firmware

For example, a firmware file called my_firmware.bin may be requested within the kernel module using request_firmware("my_firmware.bin"). The kernel will actually request the firmware file be loaded by means of a special process involving an event being sent to the system "udev" service. That service will search the paths previously shown, in the order specified.

When you supply firmware updates, especially for firmware files likely to already exist on the system (now, or in a future kernel update), it is strongly recommended to locate them in /lib/firmware/updates/. Failure to do this could result in systems unable to install kernel updates if those updates add a conflicting firmware file of their own. For example, in the case of the example firmware file name, the following location would be used:

/lib/firmware/updates/my_firmware.bin

Firmware files installed in this location will not conflict (at the system, or RPM packaging levels) with existing firmware installed in the system, and will always take priority. You should not install firmware under a specific KERNEL_VERSION directory (for example, 2.6.32-71.el6) because this will preclude the firmware from being located when used on any other kernel. The Driver Update process relies upon using the same driver for many different kernels, so use the generic location to ensure firmware files are loaded.

After adding a firmware file, you should include appropriate RPM packaging SPEC file syntax to copy this firmware file into the appropriate buildroot location during the building of your RPM package. For example, you could use the following within the %install section of your SPEC file:

```
install -m 644 my_firmware.bin \
$RPM_BUILD_ROOT/lib/firmware/updates/my_firmware.bin
```

Be sure to add the my_firmware.bin file location (without the buildroot $RPM_BUILD_ROOT prefix) to your RPM installed files filelist:
/lib/firmware/updates/my_firmware.bin

If you do not have a filelist, refer to the main sym53c8xx example shown previously, and in particular the Source file named sym53c8xx.files.

4.8 Multiple drivers and dependencies

Driver Updates can themselves depend upon other Driver Updates for correct operation, and in this way, complex dependencies of Driver Update packages and modules can be created. There are three ways that multiple drivers can be shipped and installed onto a system (others may exist, but they are not documented in this guide):

- Separately - drivers that are only loosely related, such as different classes of driver for independent functions of a Converged Network Adapter or CNA, should be shipped as separate Driver Update packages, one for each independent function. Since these drivers do not require one another, the user should install each separately in order to access network, storage, and other functions of the CNA. If you must have a dependency between these drivers, create another "top level" RPM package that depends upon the drivers and have the user install that package. This approach is strongly recommended.

- One package - drivers that are strongly related to one another or require each other for the normal functioning of any part, should be shipped in one Driver Update package. Follow the packaging guide contained within this document, substituting single module kernel source for your multiple module kernel source. Ensure that your modules are installed to a single directory as in the sym53c8xx example, but there can be more than one file with a ".ko" extension within that subdirectory. Also ensure that there are depmod "override" configuration entries in place within any depmod configuration file for any modules that replace existing system provided modules, one entry for each module involved, in the format previously explained.

- Separate dependent packages - it is possible to package multiple Driver Updates that depend upon one another at both the module level (the built-in versioning) and at the RPM packaging level, for automated dependency resolution and installation of dependent packages.

The last case (that of separate, but dependent packages) is the most complex. To achieve this kind of packaging, it is necessary to begin with your
base module upon which the other(s) will depend. Built this module, preserving a copy of the "Module.symvers" file that is automatically generated within the build directory. This "Module.symvers" differs from the one contained with the source for the kernel itself, since it refers only to your module. The kernel Kbuild build system notices that the module provides its own symbols (interfaces) for other modules and generates this file automatically at build time, just as it does when compiling the full kernel source.

Copy the Module.symvers file into the build environment for any dependent module(s) so that the build process can be aware of these dependent kernel symbols and create dependency information automatically. It is possible also to automate the process of obtaining the Module.symvers file from your various packages (for example, with some RPM scripting), in order to handle symbol version changes in your driver. If you do not wish to take this extra (and more complicated) step, remember to update the reference Module.symvers files within your dependent driver source whenever you change dependent symbol(s) (interface(s)).

For further information about this more advanced form of multiple module packaging, refer to the example contained within the kernel "Documentation" directory.

### 4.9 Further Recommendations

#### 4.9.1 Naming Guidance

When building Driver Updates, it is generally best practice to use a package name that matches the driver you are planning to ship. For example, if your driver is called "whizzbang" (with a kernel module name of "whizzbang.ko"), and the product it powers is called "The Uber Whizzbang Super Deluxe 9000", you should name the Driver Update according to the name of the driver. This means that the following example of a Driver Update name:

Name: uber-whizzbang-super-deluxe-9000

Is incorrect. This is against conventional naming practice. Instead, it should be named according to the name of the Linux kernel module provided:

Name: whizzbang

It is allowed to differentiate your driver by including your company name or other distinguishing identifier (e.g. acme-whizzbang) - especially in the case
that there may be several versions of the driver available (for example, if the driver pertains to a popular storage device that is provided by multiple vendors, with different driver versions) but the name of the kernel module ".ko" file should generally always match the package name. Of course, this always excludes the "kmod" prefix, which is added automatically.

An exception to this naming policy applies when targeting a narrow range of RHEL 6 kernels. If your module makes extensive use of non-kABI interfaces and will require a rebuild for subsequent RHEL 6 minor updates, it is recommended to name it according to the current release, and include a depmod configuration file that limits it to apply only to this release. For example, in the case that your driver is to be released for RHEL 6.0:

**Name: whizzbang-rhel60**

The "rhel60" specifies that this is intended for RHEL 6.0 systems, and the included "depmod" configuration file (in /etc/depmod.d/whizzbang-rhel60.conf) would limit the driver "overrides" statement to 2.6.32-71.*, which covers all errata and security updates for RHEL 6.0, but not RHEL 6.1 systems (since such updates are released with a base version of the -71 kernel). When a rebuild is required in RHEL 6.1, then it can be named appropriately for use with RHEL 6.1 if it is still minor version specific:

**Name: whizzbang-rhel61**

This will be treated as an entirely separate package, avoiding any conflict with the previous RHEL 6.0 package and allowing the user to simultaneously use both (for example if there is a need to boot into the older kernel to diagnose a problem, or for some other reason). If you know that your Driver Update will be frequently rebased, targeted to a specific minor update, and will not tie in well with kABI, you are strongly recommended to adopt an appropriate naming convention, as described here. Older packages for prior releases take up little disk space and can be removed later.
Chapter 5

Driver Update Disks

Driver Updates that must be available during system installation (for example, in the case of a new storage or network adapter required for correct installation of the target system) can be supplied in "disk" format. A driver disk is simply a special form of ISO (CD/DVD) image that may be burned to a CD or DVD, installed from a USB stick, transferred to a target system using PXE boot (and then used as an image file by the installer), and so forth. It is very rare to think of a Driver Update Disk as being a physical "disk", although the name comes obviously from historical usage as such.

Red Hat Driver Update Disks are created using a special tool called ddiskit. RHEL 6 systems use newer 2.x series versions of ddiskit to create a driver disk image in "type 3" format. This is the most recent disk format, which is essentially just a Yum repository of Driver Updates on a disk, complete with a description file. There is more to it, but just barely. This is intentional. Over time, Driver Update Disks have evolved toward the goal of simply being a bunch of RPM packages on removable (or otherwise) media.

5.1 Preparation

Driver Update Disks are containers for Driver Update packages. The following sections assume that you have already followed the directions in the previous chapter, and have made a Driver Update package that operates standalone. If you have not yet built a Driver Update, you should do this before continuing. It is easier to separately build the Driver Update first.
5.2 Installing and using ddiskit 2

Red Hat Enterprise Linux 6 systems require version 2.x of ddiskit in order to build RHEL 6 compatible Driver Update Disks. As of this writing, version 2.5 is current, and is available at the website referenced in "Further Resources" of the Introduction chapter of this guide. You should download the latest version of the ddiskit 2.x utility from the website referenced, and extract it by using a command similar to:

```
tar xvfj ddiskit-v2.5.tar.bz2
```

Before using ddiskit, ensure that you have installed all of its dependencies, by following the instructions that were given under "Background Requirements" in chapter 2. Specifically, ddiskit requires that you install the createrepo package, using a command similar to:

```
$ yum install createrepo
```

ddiskit contains a series of files, such as the typical README, ChangeLog, COPYING, and INSTALL files. It contains a Makefile and is driven using Make, which reads rules from a Build.rules file, but all it really does is to script the process of building a few Driver Update RPM packages and assemble them into a compatible yum repository within a CD image file.

During build, of a driver disk, each of the subdirectories listed in the "subdir" file is recursively entered and the Driver Update rpm files within are built in exactly the same way as was done manually using rpmbuild from within the "rpm" directory created in the last chapter. Once the build is completed, ddiskit copies the RPMs into a newly created yum repository (a term for a software package repository), and copies this into a driver disk image file, after adding a simple driver disk description file named "rhdd3" that contains some text related to the version 3 format.

To build your own Driver Update Disk, refer to one of the existing examples, such as that of sym53c8xx. Place the Source and SPEC files from your standalone Driver Update that you created previously into a new sub-directory with a layout similar to that of the examples. Then, type make:

```
$ make
```

Once you have successfully performed a ddiskit built, you will find that a Driver Update Disk image file has been produced:
images/dd.iso.gz

This is a regular CD/DVD ISO image file that can be burned to a CD/DVD (by unzipping the file and using CD or DVD burning software to write it to the media), or in the case of a USB stick, copied (as is, not using a command like "dd") onto the filesystem. During installation, when booted with the optional "dd" appended to the installer command line, it will prompt you to insert any "driver disk" that you may have. At that time, you can insert the CD, DVD, or USB stick. The installer will load your Driver Update automatically, and install the Driver Update package onto the target system.

5.3 The Red Hat Installer

Anaconda, the Red Hat Installer, supports loading Driver Update Disks from both physical media, and over the network. The standard RHEL 6 installation guide includes examples of the various possibilities, and sources accepted. If you do not require use of hardware during installation, but merely want to install a pre-existing Driver Update package (not a Driver Update Disk), you can of course also avail yourself of the Red Hat Kickstart scripting process to script the installation of any RPM package.

Refer to the RHEL 6 installation guide for the process of using Driver Update Disks with the current release. After installation, determine that your Driver Update was installed by using the modinfo command. For example, in the case of the sym53c8xx example:

```bash
$ modinfo sym53c8xx
```

This would list as the driver location the path to your update.

5.4 Network Driver Update Disks

Driver Updates that provide support for network adapters are not necessarily confined to installation using physical media. Special support exists for loading such drivers over the network, early in the installation, using the same PXE environment that is used to load the installer itself. To use this process, a special secondary initramfs image is created that the installer will automatically merge with its own (it is not supported to manually rebuilt the installer one), and Driver Update Disk data is loaded from the combined image. Further information on this particular form of Driver Update Disk installation is available on the Red Hat Access website at http://access.redhat.com.
5.5 Multiple architecture support

Most Driver Update Disks target a specific system architecture (or in fact, a specific affected system that requires a Driver Update). However, there is technically support contained within the Red Hat Installer to load a Driver Update Disk that contains Driver Update packages built for several different architectures. This is a rarely used feature, and so is not supported by the ddiskit utility directly at this stage (though this may be added).

If you would like to build a Driver Update Disk for multiple architectures, first run the ddiskit tool on each different architecture. Then, locate the "disk" output directory within each instance of ddiskit. You will notice that each contains only a file named "rhdd3" and a directory named for the current architecture\(^1\). Each of the "rhdd3" files will be identical.

Create one unified "disk" directory containing a single copy of the "rhdd3" file, and the other architecture directories. For example:

```
disk/rhdd3
disk/i386->i686
disk/i686
disk/x86_64
```

Then, manually build the images/dd.iso.gz file:

```
mkisofs -R -o images/dd.iso disk
gzip -9 images/dd.iso
```

The output file should be tested on multiple architectures.

\(^1\)An exception is that the i686 contains a directory named i686 and a symbolic link to it, named i386. Both are required in order for the Driver Update Disk to function.
This book makes use of a number of distinct terms related to the Linux kernel, Red Hat Enterprise Linux 6 (RHEL 6), the RHEL 6 kernel, and other products and technologies. That nomenclature is summarized here.

The following terms pertain to Red Hat Enterprise Linux:

- **Red Hat Enterprise Linux** - An "Enterprise Linux" distribution from Red Hat, based upon a large number of Open Source technologies, such as the upstream Linux kernel (explained herein), and its associated tools. Red Hat Enterprise Linux is officially always referred to as "Red Hat Enterprise Linux", but is sometimes shortened to RHEL in casual use. That latter abbreviation is used in portions of this text.

- **Red Hat Enterprise Linux 6** - The latest version of Red Hat Enterprise Linux as of the writing of this document. This release is based upon the 2.6.32 upstream Linux kernel release from kernel.org, with various additional enhancements from Red Hat, and from the Open Source community. Some features from 2.6.33, 2.6.34, and later Linux kernels have been "backported" to the RHEL 6 kernel, and some others have been disabled by means of configuration. The latter includes rarely used and obscure device drivers that create additional support burden without sufficient customer demand for inclusion.

The following terms pertain to the Red Hat Driver Update Program and the "Driver Updates" technology described in this document:

- **Driver Update Program** - The official Red Hat program allowing for officially sanctioned and officially distributed Operating System driver updates from Red Hat. This is a value-added partner program that provides a means to support certain hardware platforms between regularly scheduled Operating System update releases. It typically comes into play as part of a forthcoming system certification since Red Hat
do not support third party drivers (even those built using the "Driver Updates" infrastructure, although those using the Driver Updates infrastructure obtain certain other benefits of driver compatibility).

- Driver Updates - These are individual Linux device drivers, contained in an enhanced RPM package format that provides greater compatibility against a wide range of RHEL 6 kernel updates. It is typically not necessary to rebuild Driver Updates as the kernel itself is updated on an installed system. Instead, the user can simply install a driver package and expect their hardware to continue to operate automatically across regularly scheduled system updates.

- Kernel ABI (kABI, pronounced "kay-ABI") - This is a value-added feature of Red Hat Enterprise Linux. It refers to a stable kernel ABI between the RHEL 6 kernel and certain third party device drivers, and is the means by which compatibility is ensured across updates to the distribution. This technology is not present in non-Enterprise kernels, such as mainline. There is no official "Linux Driver Model" of the form present in certain other Operating Systems, such as Microsoft Windows, because the upstream kernel community considers this an undue burden on innovation. The kernel ABI stability allows a compromise to be provided for RHEL 6 users, who are able to use pre-built RHEL 6 drivers across updates, without recompiling.

- Kmod (pronounced "kay-mod") - This is a legacy term used to refer to Driver Updates. It is sometimes still used in conversations because the package file names for Driver Updates contain the prefix "kmod-". For consistency, this and other documents now conventionally use the term "Driver Update" to refer to an individual driver package, or "Driver Updates" and "Driver Update Program" to refer to the concept of applying packaged drivers and their application in the specific Red Hat partner program that builds upon Driver Update technology.

- Minor release - A minor release (or minor update) to RHEL 6 occurs annually or semi-annually and results in a version increase, such as from RHEL 6.0 to RHEL 6.1. This differs from other regularly scheduled system errata and security updates that take place within a release. Errata and security updates applied to the current release are sometimes referred to as "async", "Z-stream" or EUS updates. Red Hat undertakes not to change any kernel interfaces wherever possible in errata or security updates, but reserves the right to make changes
to non-kABI interfaces in minor updates. When such changes occur, third parties typically have many weeks or months to update their drivers during beta cycles, and so forth. Additionally, Red Hat is working on a kernel ABI notification program for its partners.

The following Linux kernel related terms are used periodically within this document. Each of these are terms used by the wider Linux community, and are not specific to Red Hat’s products or technologies:

- **Backport** - This refers to the action of taking pieces of Linux kernel code and adapting them for use within a different kernel release, such as from the "mainline" to RHEL 6 kernel. The term is known as a "backport" because the mainline kernel undergoes very heavy developmental churn in comparison with the (older version based) RHEL 6 kernel that receives a subset of those new features developed in mainline. The RHEL 6 kernel receives greater testing and is generally more stable than mainline, but the differences between the kernels create a need for backports. Examples of backporting include adapting Interrupt routines for the changes between kernel 2.6.32 and mainline.

- **(Device) Driver** - A driver is a Linux Kernel Module (a run-time loadable extension to the Linux kernel that forms a seamless part after loading) that supplies code to drive or control a specific hardware device. The vast majority of Linux kernel modules are in fact device drivers, which is why terms such as "Driver Update" are used in place of the (more technically correct) "Module Updates". Device Drivers are typically loaded automatically by the system, either during system boot, or as devices are "hotplugged" (added or removed) thereafter. Advanced users and system administrators sometimes use system utilities such as "modprobe" to cause additional drivers to be loaded and unloaded from a particular system. End users typically have no involvement in the actual loading and unloading of kernel drivers.

- **Linux Kernel Module** - This is a more general term for extensions that can be loaded into a running Linux kernel (such as that supplied in RHEL 6), of which device drivers form a subset. Depending upon context, "Linux kernel module" can be either a reference to module source, or to pre-compiled code. Kernel modules extend the capabilities of the Linux kernel at run time, through the addition of certain features, such as support for new hardware devices (device drivers). Not all pieces of the kernel can be updated or altered using kernel
modules, and not all possible updates are supported by Red Hat in
RHEL 6. For example, it is technically impossible (for all practical
non-research purposes) to provide new chipset or CPU support within
a Linux kernel module. Pre-compiled kernel modules are managed
using utilities such as those contained within the module-init-tools
package. These include the "depmod" and "modprobe" commands.
Typically, users do not need to operate these tools independently of
the automated processes within the system, although some advanced
users and administrators will occasionally wish to apply customized
configurations.

- Mainline kernel - This refers to the kernel shipped by Linus Torvalds
via the kernel.org website. The mainline Linux kernel forms the basis
for the RHEL 6 kernel, which also includes a number of additional
features. Many of the features within a specific version of the RHEL 6
kernel come from releases of the mainline kernel that took place after
RHEL 6 was released. These features are said to have been "back-
ported", as explained earlier in this section. The RHEL 6 kernel has
passed additional stress tests, quality control, and beta cycles that
do not apply to mainline. Driver Updates shipped by Red Hat typical-
ly must already be "in mainline" before they are backported to the
RHEL 6 kernel, tested, and then released. This allows such updates
to be easily included as a built-in component of future RHEL 6 minor
updates, since Driver Updates are intended as an interim measure.

- Official kernel - This is an alternative reference to "mainline". In this
context, "official" refers to the fact that the kernel comes from Linus
Torvalds. The RHEL 6 kernel is of course also an official component
shipped by Red Hat, but use of the word official in this document
follows the standard convention in the wider Linux kernel community.

- Upstream kernel - This is an alternative reference to "mainline". It
is said to be upstream in analogy to a flowing river, with distribution
vendors such as Red Hat serving as the downstream. This broadly
describes the kernel development workflow in which engineers (many of
them working for Red Hat) first push new features into the upstream
kernel, then integrate them into various products and distributions
downstream. The workflow ensures that an "upstream first" policy
is generally enforced, which reduces maintenance burden for distribu-
tors and guarantees new features are well tested in more experimental
distributions (such as Fedora) before being released to customers.