

RHEL5 Driver Update Program Technical Whitepaper

Jon Masters <jcm@redhat.com>

October 9, 2007

Abstract

Red Hat Enterprise Linux 5 (RHEL5) includes a new feature known as the RHEL Driver Update Program. This ongoing project aims to allow both Red Hat and third party developers to supply updated Linux Kernel Modules (commonly referred to as “drivers”) for RHEL systems, packaged in such a fashion as to operate across a wide range of Enterprise systems, without requiring the end user (or administrator) to rebuild said drivers on every kernel update. This paper describes how the model works, from both a technical and end-user perspective, using example packages to emphasize key points. It also briefly describes some of the testing criteria that has been exercised against the RHEL5 product itself.

Project Overview

RHEL5 includes a new feature known as the Driver Update Program. This allows third parties to produce pre-compiled Linux Kernel Modules (LKMs) that are portable across a range of target RHEL5 systems. Such driver modules can be installed by end-users without regard for the specific Linux kernel version in use - so long as it was shipped by Red Hat as part of RHEL5 - and without having to redeploy driver modules whenever the kernel itself is updated on their machines. The program is helping to define a joint packaging standard between Red Hat and other vendors.

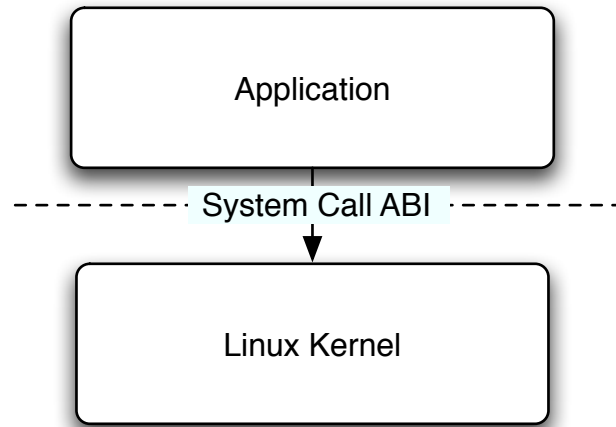
In order to achieve this decoupling between a third party supplied Linux Kernel Module (LKM) and the RHEL5 kernel itself, the existing Linux Kernel Module versioning infrastructure has been extended into the RPM packaging process, through the addition of new RPM dependencies. This modification allows RPM to make more intelligent decisions about the compatibility of pre-compiled driver modules and to resolve module dependencies in the same way that other system package dependencies are resolved. Existing tools, such as yum and up2date, can be used in order to install these third party Linux Kernel Modules without a need for the user to understand the precise mechanics of the module infrastructure itself.

Central to the Driver Update Program is the notion of kernel ABI stability within compatible RHEL5 kernels. This is discussed in the next section.

ABI Stability Considerations

The Linux kernel exists in a variety of forms and is supplied by a variety of entities, companies and (on occasion) individuals. These different parties all refer to the kernel that they develop as “Linux”. The official Linux kernel (also referred to as “upstream”, “mainline”, “mainstream” or “the mainstream Linux kernel”) is supplied by Linus Torvalds and other developers, via the <http://www.kernel.org/> website. This official Linux kernel is then standardized, undergoes stability testing (and appropriate patching) and various other processes as part of the vendor-driven activity of producing a Linux product. When Red Hat shipped RHEL5 (and all subsequent updates to the RHEL5 product thereafter), it was based upon the upstream 2.6.18 Linux kernel, but it with certain improvements made as a result of stability testing and partner involvement. RHEL5 also has what Red Hat term a stable kernel ABI. It is the stability of the Linux kernel ABI with which we are most concerned in providing compatible Linux Kernel Modules on RHEL5 systems - this is a central requirement in order for third party modules to be added to a RHEL5-based Linux system.

Kernel ABI stability is in some respects similar in concept to that of the ABI stability between software applications and system libraries, as you will learn in the following paragraphs. Once you have a thorough understanding of the importance of regular application/library ABI stability, then the discussion will turn to the specifics of the RHEL5 kernel ABI stability, as needed by the RHEL Driver Update Program.



(a) The Linux application kernel interface: System Calls

Application ABI stability

Whenever you use an end-user application written for Linux (or indeed any modern Operating System), you are reliant upon ABI stability between that pre-compiled application and any system software libraries upon which it depends. You are also reliant upon the interface between the Operating System and the application remaining consistent for many years at a time. Changes in this space should be few and very far between, in order to avoid any breakages that could affect existing software applications. The interface between the Linux kernel and user applications running on a Linux system is determined by the upstream Linux kernel development community and those involved in the development of critical system infrastructure. The goal being that development should very rarely (if ever) force users to be subject to application visible changes. Changing the ABI between the Linux kernel and regular user applications is carefully avoided. In fact, the Linux community frowns upon making such changes.

You can see an example of Linux ABI stability between regular applications and the Linux kernel in figure 1.

Applications making system calls into the kernel can be sure that the system call number (a piece of data passed between the application and the kernel during a system call operation) that they use when requesting a service of the kernel will be the exact same system call number from one kernel release to the next¹. New system calls can be added to the Linux kernel, but existing system call numbers are never recycled (if deprecated

¹On the same architecture. System call numbers can (and do) differ between different architectures upon which Linux runs, but applications are built for a specific architecture in any case.

over many years, they are instead replaced with dummy calls). The Linux kernel maintains a variety of other interfaces - procfs, sysfs, and related pseudofilesystems are examples of these - that also cannot be changed once any significant dependency has been established between a large number of applications and the kernel providing these interfaces. The number and variety of interfaces between regular userspace application and the Linux kernel continues to grow with time.

Kernel ABI stability

Obviously, ABI stability is necessary in order to have any ability to ship pre-compiled application programs for Linux systems. ABI stability is, however, not only an issue for compatibility between regular applications and the Linux kernel. ABI stability is a critical issue for many third party KISVs (kernel ISVs) who need to ship drivers designed to work with a particular version of the Red Hat Enterprise Linux kernel. Within the RHEL kernel, this means that certain standard functions and data structures are normally relied upon by external parties. For example, whenever a third party supplied Linux Kernel Module (LKM) needs to log a diagnostic or log message into the system logs, it can use the kernel-provided `printk()` function. Because almost all Linux Kernel Modules (LKMs) require the use of the `printk()` function from time to time, its prototype (the interface between it and third party modules) cannot change without breaking a substantial number (if not all) of those third party users of that function.

The official (upstream, mainline) Linux kernel makes *absolutely no guarantees* of any kind with respect to the compatibility or non-compatibility of third party modules between one release and the next, while a vendor such as Red Hat is in a position to offer a value-added improvement upon that situation. Red Hat therefore offer a stabilized Linux kernel ABI within a given release of the RHEL product family. In this way, although it is not possible to produce a pre-built kernel module that works “on Linux 2.6”, it is possible to produce a Linux kernel module that is known to work “on Red Hat Enterprise Linux 5”. Red Hat strives to retain a stable kernel ABI across a particular product release. This means that any one kernel released for a particular RHEL version is guaranteed to retain a level of compatibility with other Red Hat kernels² released for that same version.

A module built against one particular variant of the RHEL5 kernel will be compatible (at the Linux Kernel Module level) with any updated version of the same variant of the RHEL5 kernel, although not with an arbitrarily chosen similar variant RHEL4 kernel. RHEL5 kernels come in 10 variants (more on these in the next section) and each kernel module must be built for each individual variant of the RHEL5 kernel. ABI compatibility and supported symbols are per-variant, since they will be different - for example,

²Some caveats do exist here. For example, Red Hat does not endorse the ABI stability of internal test kernels (including those made available for limited external circulation in the aim of testing), nor does it endorse the compatibility at the ABI level of certain other non-standard kernels (for example, Real-Time enhanced kernels and others that are not shipped as part of the base RHEL product).

a Xen kernel contains additional base hypervisor functions not present in a non-xen kernel. The number of variants may change over the lifetime of RHEL5. If additional variants become supported, then the overhead of building a module RPM package to support those additional kernels is very low indeed.

Tracking the RHEL5 kernel ABI is a complex operation not limited to merely the “external” interfaces being exported by the Linux kernel to its Linux Kernel Modules (LKMs). Indeed, it is necessary to insure other (more subtle) semantic changes do not also impact upon the assumptions made by existing drivers, which often come to rely upon one specific behavior in the upstream kernel only to find it change a few months down the line. The RHEL5 Driver Update Program addresses this need in two distinct ways:

- Red Hat engineers track internal semantic changes as they occur and isolate their impact upon external third parties.
- Kernel ABI meta-data is tracked in the kernel package itself in order to ensure compatible module/kernel compatibility.

Internally, within Red Hat, kernel engineers must fully evaluate every potential kernel patch against its impact upon the RHEL5 kernel ABI and how that will affect or otherwise influence known compatibility with third party drivers. Patches are only accepted once their impact upon kABI has been assessed. Indeed, it is not even possible for Red Hat engineers to build a kernel within the internal build system unless it conforms to the ABI commitments already in place. Once on a target RHEL5 system, kernel ABI compatibility between third party modules and the RHEL5 kernel is determined via new kernel package dependencies, which extend the regular Linux kernel module versioning system and make it available to the RPM package manager itself. RPM can thus determine whether a module tagged for installation is known to be compatible with the kernel in use.

Kernel Variants

The Linux kernel is available for a wide range of possible systems. These can comprise many different architectures, platforms and configurations of the kernel for such platforms. Consequently, a number of kernel variants typically exist on any RHEL system. In the case of RHEL5, there are 10 kernel variants at the time of RHEL5.1 GA - i686, i686PAE, i686xen, ia64, ia64xen, ppc64, ppc64kdump, s390x, x86_64, x86_64xen. Each of these kernels is built to a specific configuration. There are alternative kernels for each platform, while some platforms have more than one variant - for example, i686 (regular PC systems based upon Intel-compatible processors) has 3 variants in RHEL5,

depending upon the presence or lack of the PAE memory extension, and/or Xen Hypervisor. RHEL5 Driver Update Program packages must be built on the appropriate architecture against which they will be run, but those architectures supporting more than one variant can build all such variants - for example, i686 machines can build Driver Updates for i686, i686PAE and i686xen, while x86_64 machines can build for x86_64 and x86_64 only.

It is necessary to build one RHEL5 Driver Update Program RPM package for each defined RHEL5 kernel variant, if you wish to make your Linux Kernel Module (LKM) available to users of that specific kernel variant. For example, you may build for i686 and i686xen but not for i686PAE or s390x. Building need only be done once, unless it becomes necessary to update your driver at some point in the future.

Module Versioning

The upstream community developed Linux kernel implements a module versioning (modversions) infrastructure, which is used by the Linux kernel itself in order to determine whether a particular pre-compiled Linux Kernel Module (LKM) is compatible with the running kernel at module load time. This module versioning infrastructure is designed to prevent system crashes and other problems brought about as a result of users attempting to load drivers built with different versions of the GNU C Compiler (GCC) and other installed system tools, but it can also be used to determine compatibility of modules built for one kernel being loaded into another. The Driver Update Program extends this existing module versioning infrastructure into the RPM packaging system itself. In a nutshell, the technical implementation of the Driver Update Program is “modversioning for RPM”, but with a few niceties added - such as the various processes and tools supplied to third parties to aid in driver update packaging.

The Linux kernel includes drivers for thousands of different devices in a typical configuration. Many of these drivers do not change from one release of the kernel to the next, and the majority of the interfaces that those drivers use within the kernel also are unlikely to change substantially from one minor kernel release to the next. Consequently, support for module versioning was added to the kernel early on. Module versioning uses various specially generated checksums to determine whether a given module is compatible with a particular Linux kernel, at load time. Each exported symbol within the kernel is checksummed and it is that checksum that is used to match a specific Linux kernel and compatible modules at load time. If the checksums don't match then the module was built against a binary incompatible Linux kernel release. If the checksums do match, then the module may be loaded by the kernel. Even matching checksums won't guarantee compatibility, because some changes cannot be tracked using module versioning alone. This is where Red Hat engineers come in. They carefully assess the semantic impact of potential patches on kernel ABI.

The meta-data necessary in order to add comprehensive checksum information to any and all Linux Kernel Modules (LKMs) is actually stored within a file called Mod-

ule.symvers in the pre-built Linux kernel source tree (see /usr/src/kernels/ on any RHEL5 system for the version of this file that is included in the Red Hat kernel-devel package, which must be installed in order to build kernel modules for use with the RHEL5 Driver Update Program packaging standard). This file contains a checksum of publicly exported kernel symbols (functions and data structures that are exported using the EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL() macros in the Linux kernel source) and it is used when building external (and also those supplied with the kernel) modules in order to ensure that the runtime Linux Kernel Module loader will be able to successfully load the built module. Linux will refuse to load a module unless its checksums match those of the running kernel in use - an additional check.

Each entry in the Module.symvers file follows the following format:

```
symbol_checksum  symbol_name  symbol_file_location  type_of_export
```

The symbol checksum is calculated during the kernel build process, taking into account the prototype of any functions and the data structures used by those functions. Here is an example checksum for the external module function bttv_sub_register provided by the bttv V4L (Video 4 Linux) driver:

```
0x15de472b bttv_sub_register drivers/media/video/bt8xx/bttv EXPORT_SYMBOL
```

Many functions are built into the kernel itself (i.e. they are built into the core kernel binary itself and are not part of any externally loadable Linux Kernel Module(s)). Their entries look a little different:

```
0xa043d94b struct_module vmlinux EXPORT_SYMBOL
```

The struct_module function has a checksum of 0xa043d94b in this particular kernel build and it is provided by the core kernel (vmlinux), not a loadable module. The symbol is exported using a regular EXPORT_SYMBOL macro (in kernel/module.c) as opposed to one of the variants - such as EXPORT_SYMBOL_GPL - used in other instances. Many of these symbols that are built into the kernel are shown as being within the vmlinux kernel image itself - the scripts that will process the kernel sources and produce RPM dependency information automatically take this into account.

Module versioning in action

You can see module versioning in action by using the special flag `--dump-modversions` with the `modprobe` utility, provided by the `module-init-tools` package on RHEL5 systems. In the following example, the module version dependency information for the `ext3` filesystem driver (normally loaded on most RHEL5 systems) can be seen:

```
$ /sbin/modprobe --dump-modversions \  
  /lib/modules/2.6.18-1.2876.el5xen/kernel/fs/ext3/ext3.ko  
0x9a7513c8 struct_module  
0x179e8de8 mb_cache_entry_find_next  
0xecac7e8b kmem_cache_destroy  
0x5a34a45c __kmalloc  
0x91b56d0e security_ops  
0x3922d605 _spin_trylock  
0x6b1b67d3 __bdevname  
0xfde9c3a7 sb_min_blocksize  
0x090eaf0b generic_getxattr  
0x42f8c19a up_read  
0xa50a8a5e journal_restart  
...
```

RHEL5 Linux Kernel Modules (LKMs) have the file extension “.ko”, referring to Kernel Object. These module files are loadable by compatible Linux RHEL5 Linux kernels, as determined by the module loading system built into the kernel. Each symbol checksum in the output from the `modprobe` command is compared internally with the checksum of those functions that are supplied by the RHEL5 kernel before a given module is linked into the running RHEL5 kernel at load time. Information about checksum dependencies is stored within a special “.modinfo” section of the RHEL5 kernel “.ko” file. For interested readers, this is simply another regular section in an ELF file - kernel modules are special ELF executables.

The full `ext3` symbol listing on this particular system (running a Xen variant of the RHEL5 kernel) is 232 lines long, encompassing a wide range of kernel symbols needed by `ext3`. The Linux kernel will ensure that all symbols needed by this `ext3` filesystem module are fully resolved prior to completing the load of a module - unresolved checksums (caused by a module being used against an incompatible kernel) count as unresolved symbol dependencies and will prevent a module from loading, resulting in an error message being logged (and, quite possibly, an inability to boot the system in question - since `ext3` is usually loaded within the early `initrd` image on RHEL5). The module versioning infrastructure exists in order to ensure that an incompatible module is not loaded, so the error situation described should never occur in real life on production systems.

Module versioning limitations

Module versioning is very useful in a variety of situations, but it does have drawbacks. Since it relies upon every change to the kernel source code affecting the binary ABI between internal components, it is possible for a more subtle change to internal kernel semantics (but retaining the same kernel ABI) to remain untracked by the module versioning system alone. Therefore, the RHEL5 Driver Update Program also relies upon the highly skilled developers at Red Hat who remain ever on guard for subtle semantic changes affecting third party Linux Kernel Modules (LKMs).

You can help to mitigate compatibility issues by consulting third parties for updated drivers before reporting a bug against the RHEL5 kernel and encouraging third party driver authors to maintain a rigorous driver testing process as new releases become available.

RPM Enhancements

In order to support making KABI meta information available to the RPM software itself, RHEL5 features an updated set of RPM packaging macros. You can use these macros when building your own RHEL5 kernel drivers in order to produce your own RPM based driver updates, suitable for installation on a wide range of RHEL5 target systems. The packaging process extends the aforementioned module versioning infrastructure such that module versioning information is available in the form of regular RPM package dependencies, resolvable at installation time. In order to achieve this, various modifications have been made to the RPM packaging process for kernel modules (when packaged, these are commonly referred to, in Red Hat nomenclature, as “kmods”). These modifications include the addition of kABI tracking dependencies, as well as a few cosmetic enhancements, and a general purpose wrapper macro.

Kernel dependencies

The RHEL5 kernel build process introduces several additional stages, during which additional kernel dependency information is added to the RHEL5 kernel package. Here is an example subset of additional kernel dependencies:

```
$ rpm -q --provides kernel-2.6.18-1.3002.el5xen
kernel = 2.6.18
kernel-i686 = 2.6.18-1.3002.el5xen
...
kernel(rhel5_vmlinux_ga) = 16e199f9f0700b2723a9334d48044dcd0a02790b
kernel(rhel5_drivers_pci_hotplug_ga) = f2d3da1edcba66c7dc0b09377b3e0b4ccdfd98aa
```

```
kernel(rhel5_drivers_base_ga) = 2e3adfd0ee21ca2a1d4d6ca181ec73a6f0221543
kernel(rhel5_arch_i386_mach_xen_ga) = f140ac8f99498655de6d2128067bff4946e3cbcf
kernel(rhel5_net_sunrpc_ga) = 24164573208ac3a8706912194cb786857d2222d6
kernel(rhel5_drivers_leds_ga) = 7f2fc85b83543489b4f76aae3cf7d89f8d59f437
kernel(rhel5_sound_pci_ac97_ga) = f82c2204168fd64cff0fb979fa48f7834d183449
...
```

There are around 110-120 of these additional dependency groupings in the current RHEL5 kernel package. Each dependency is a checksum representing a specific collection of exported kernel symbols within the Linux kernel, as defined by a kernel ABI whitelist. The whitelist defines which kernel symbols will form which kernel dependency groupings and thus how the checksum will be calculated. Therefore, it is not possible to remove symbols from the RHEL5 kABI whitelists once a kernel has been built using and packaging those symbols, without possibly (and highly likely) changing the overall checksum of the group within which a symbol is contained - this would break compatibility with all existing pre-built drivers using those kernel symbols. You will learn more about the RHEL5 kABI whitelists later in this paper, and at driverupdateprogram.com.

In the example RHEL5 kernel RPM package dependency output, you can see the “kernel(rhel5_drivers_base_ga)” dependency - which includes a number of symbols defined within the “drivers/base” subdirectory of the Linux kernel source. A similar dependency, “kernel(rhel5_module_ga)” contains an intentionally much more limited set of symbols - just `printk()`, and `struct_module` - since these are known to be needed by almost all kernel modules in use (it was, therefore, decided to split these fundamental dependencies into a special symbol grouping). The actual kernel package RPM dependency information is generated by a script within the RHEL5 kernel sources (this script is called `kabitoool`), which parses the large per-kernel-variant kernel ABI whitelists of kernel symbols and outputs these, grouped according to the groupings within the data file. You can see example sets of kernel symbol whitelist data via the whitelist files located on the driverupdateprogram.com website.

kabitoool

The RHEL5 kernel source includes a list of kernel symbols - these files are always prefixed “kabi_whitelist”, and may have a postfix, depending upon the type of source being used - that determines which kernel symbols will be used by the RHEL5 Driver Update Program. Each individual kernel symbol is listed according to the desired RPM grouping in the resultant kernel package. For example, here is an extract from one possible definition of the “kernel(rhel5_kernel_ga)” dependency (grouping) in the `kabi_whitelist (kabi_whitelist_i686)`:

```
[rhel5_kernel_ga]
```

```
kill_proc
kthread_should_stop
_read_lock_irqsave
utrace_regset
timespec_trunc
printk_ratelimit
__symbol_put
call_rcu_bh
blocking_notifier_chain_unregister
request_dma
register_kprobe
_spin_unlock_irq
__kfifo_get
interruptible_sleep_on
kfifo_free
do_exit
```

Each of the RHEL5 kernel ABI whitelists was generated by taking a list of all kernel symbols and extracting from that list all those symbols used by known in-kernel modules, as well as third party supplied modules for which Red Hat has data. The goal is to provide a minimal whitelist that grows over time if necessary, rather than immediately including every kernel symbol that may have no actual real world users. This both helps to reduce risk to Red Hat that a later security issue could compromise the kernel ABI, and also helps to define a sensible set of intended third party kernel ABI.

The `kabi_whitelist` file is processed by a kernel package build script - `kabitool` - whose job is to locate the symbol named in the whitelist file and reconcile that against the actual symbol as listed in the `Module.symvers` file. An in-memory table of symbols is computed, and the checksums of individual functions are subsequently grouped together in files named after the grouping defined in the whitelist file. Each output grouping file is then checksummed to create the kernel package RPM dependency for that group. Effectively, each kernel package RPM dependency is a “checksum of checksums” and changes whenever a component symbol within the group changes. The kernel checksums are MD5, whereas the larger checksums used in the RPM packaging process happen to be SHA1. There was no particular reasoning behind this differentiation in hash use.

Since RHEL5 kernel ABI is grouped according to kernel subsystem, theoretical ABI changes to certain subsystems may not break compatibility with every third party driver module - only those that use the particular set of symbols that actually had changed, should such a situation ever actually occur in real life. The RHEL5 kernel ABI whitelist information is grouped together into files, known as `symsets`, in the kernel-devel package and used whenever third party drivers are built against a RHEL5 system - it is this data that helps to automate the process of creating `kABI` group checksum-related RPM dependency information (providing the mapping between symbols in third party drivers and RPM dependencies).

symsets

Each kernel ABI dependency group, or set, is written out to a text file. This file lives in the RHEL5 kernel-devel package (for example in /usr/src/kernels/2.6.18-kernel_version) and is needed whenever a third party module will be built against a RHEL5 kernel and packaged into an RPM.

Here are some examples of symsets files:

```
rhel5_init_ga.1ddb4c6ff8388416c794ccdda5a038dfc5f68b3
rhel5_kernel_ga.3da5aac618143daaab8bdecae3c5ef13162126a0
rhel5_kernel_irq_ga.8f9d9be5872245513862ebf5c69d1c7e099adbcf
rhel5_kernel_power_ga.3c2c37d553ebecf99e6d147387f3dd4b5f5df7b7
rhel5_lib_ga.dd010fa49d7dfc0f898d4793bca5397886949ee8
rhel5_lib_zlib_deflate_ga.8c9c879eb1c516e805a067a60c667d4c04ca8d81
rhel5_lib_zlib_inflate_ga.c45d3f9e25fd94604c33c5af840f2f01677d821c
rhel5_mm_ga.f5959af65c341e625618542f366c7ec97d14afe4
rhel5_net_ga.0997486955f99f146458434d560a2aac97202bd0
```

Each file is named after the RHEL5 kernel ABI dependency grouping that it represents, with the group checksum appended. The files are collectively stored within an archive - symsets-kernel_version.tar.gz - that is contained within the corresponding kernel-devel package for the kernel to which it relates (and is only required to be installed at build time, production systems do not need to have the kernel-devel package installed or any other build tools beyond those normally present on a particular RHEL5 system). The RPM build scripts make extensive use of these symset files when building RHEL5 Driver Update Program packages (via the specially added post-processing scripts in the redhat-rpm-config package, a package that contains Red Hat's RPM macros) in order to match any necessary kernel dependencies against symbols required of kernel modules.

Each of the component symsets files has a similar layout. Here you can see part of the content of the "rhel5_kernel_ga" group contained within the file rhel5_kernel_ga.3da5aac618143daaab8bdecae3c5ef1316

```
0xda85dfc7 _read_lock_irqsave kernel/built-in.o
0x6e9dd606 __symbol_put kernel/built-in.o
0x2a6f6519 kfifo_free kernel/built-in.o
0x7a6eaabd register_posix_clock kernel/built-in.o
0xb1c3a01a oops_in_progress kernel/built-in.o
0x7ff10ccf raw_notifier_call_chain kernel/built-in.o
0xcc7fa952 local_bh_enable_ip kernel/built-in.o
0x176fffc3 dma_spin_lock kernel/built-in.o
0x2920d619 flush_signals kernel/built-in.o
0x60a13e90 rcu_barrier kernel/built-in.o
```

```
0x6483655c param_get_short kernel/built-in.o
0x999fe899 set_current_groups kernel/built-in.o
0xd3099f8c sysctl_ms_jiffies kernel/built-in.o
0x3482fcb1 down_write kernel/built-in.o
0x93a6e0b2 io_schedule kernel/built-in.o
```

There are many thousands of “public” symbols exported by a typical Linux kernel and the RHEL5 kernel is no different in that respect. One reason for grouping the symbols together is to reduce the complexity of tracking many thousands of different symbols at the packaging level when really only an overall picture of compatibility is required at installation time. As part of the grouping exercise, many symbols that are known to be unneeded by third party modules were removed - over time, it is hoped that the actual third party module-visible exports within the upstream Linux kernel can be reduced in accordance with this trend, through the addition of a special new export macro tailor made for this cleanup exercise.

Module dependencies

Linux Kernel Modules (LKMs) built using the RHEL5 Driver Update Program are commonly referred to as kmods. They are built just like any other RPM package - by using a regular RPM SPEC file. The standard system RPM scripts have been modified to add dependency information whenever kernel modules are detected (whenever a .ko file is being built via RPM). The result is that RHEL5 kmod dependencies look like this:

```
$ rpm -qp --requires rpm/RPMS/x86_64/kmod-redhat-example-2.0-0.x86_64.rpm
rpmlib(VersionedDependencies) <= 3.0.3-1
redhat-example-kmod-common >= 1.0
/sbin/depmod
/sbin/depmod
/bin/sh
/bin/sh
/bin/sh
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rpmlib(CompressedFileNames) <= 3.0.4-1
kernel(rhel5_kernel_ga) = 80eb4f3bb28f48a6c959596f7dd1bf8bc3526b1a
```

RPM simply has to resolve the regular “kernel(kernel)” dependency against the currently installed Linux kernel before the module is installed.

Module-init-tools modifications

The RHEL5 Driver Update Program takes advantage of a few modifications to the module-init-tools package. RHEL5 Driver Update Program modules are built to install within the `/lib/modules` subdirectory of the kernel against which they were compiled, regardless of its installed state. For example, if a module `foo` is built against kernel `2.6.18-1.2740.el5` then it will be installed in:

```
/lib/modules/2.6.18-1.2740.el5/extra/foo.ko
```

The “extra” subdirectory is part of the `kmod` specification and is the standard subdirectory for extra (additional) packaged kernel modules that are made independently of the modules shipping in the kernel itself. Once the driver module is installed, another script (`weak-modules`) will examine it, and automatically create symbolic links within the `/lib/modules` subdirectories of other, compatible Linux kernels that are already installed on the target system. These compatibility links are located within another subdirectory, “weak-updates”, as in the following example:

```
/lib/modules/2.6.18-1.2741.el5/weak-updates/foo.ko -> /lib/modules/2.6.18-1.2740.el5
```

The module `foo.ko`, built against kernel `2.6.18-1.2741.el5`, is compatible with kernel `2.6.18-1.2740.el5` and so a symbolic link has been created within the `/lib/modules` hierarchy, pointing from one kernel module tree to the other. The system level tools, provided within the standard Linux `module-init-tools` package were modified during the course of RHEL5 to handle these symbolic links and to automatically follow them whenever loading Linux Kernel Modules (LKMs).

It is also possible for third party RHEL5 Driver Update Program module authors to override the priority of different subdirectory search orderings when an external module exists that shares the name of a built-in kernel module. By default, RHEL5 will follow the search ordering defined in `/etc/depmod.conf.dist`, which instructs it that - unless a third party module specifically ships with a configuration file containing the override option - it must always prefer modules supplied by the RHEL5 kernel, unless the driver update happens to have been built for exactly the build in use. In practice, this means that you cannot normally rely on your third party module to replace a driver supplied by Red Hat. The intention here is that driver updates are a stop-gap mechanism, and that once the driver is in the RHEL5 kernel, that should be used instead. This supports the more typical use case of a new driver being supplied first as an update, and later entering into the RHEL5 kernel - the in-kernel driver usurping the update.

Obviously, sometimes third parties really do want to override a driver that was supplied in the RHEL kernel. This is possible, through the use of additional module configuration files. Red Hat explicitly does not support or endorse this activity, since Red Hat

generally prefers to correct problems with Red Hat supplied drivers through kernel errata or other similar mechanisms. But there are times when third parties really do want to override RHEL5 supplied in-kernel drivers, and for those times we have provided such a capability in the RHEL5 version of the depmod command.

For example, you can override the QLogic qla2xxx driver supplied in RHEL5 with the following `/etc/depmod.d/qla_example.conf` special configuration file to module-init-tools (this file should be supplied in the driver update kmod package itself):

```
#
# qla_example.conf
#
# override default search ordering for kmod packaging, and always use the
# externally supplied third party replacement module.
override qla2xxx * weak-updates/your_qla2xxx
```

This instructs module-init-tools that it should always prefer a module named `qla2xxx`, that is located in a `weak-updates` path component, over that which might be supplied elsewhere. The “`your_qla2xxx`” refers to the fact that driver update modules are always located in a subdirectory, by convention based upon the name of its containing RPM package. You can find documentation on module overrides in the `depmod.conf` man page.

Current Featureset

In the current implementation of the RHEL5 Driver Update Program as present in RHEL5, provision is made for the packaging of third party Linux Kernel Modules, including kernel ABI dependency information along with the Linux Kernel Modules (LKMs) themselves. The individual updates (the kmod RPMs) can be installed and removed using regular RPM commands, and there is additionally support for transparent handling of kernel upgrades such that drivers continue to be available to newly updated Linux kernels. The Driver Update Disk mechanism used during installation time was also modified in RHEL5.1 in order to support the automatic installation of corresponding driver update RPM packages at install time.

The following base functionality is provided in the current implementation:

- Installation of a new Driver Update Program package.
- Installing/upgrade of the RHEL5 kernel package³.

³Red Hat kernels are not “upgraded” in the traditional sense. A new kernel package is always installed before an old one is removed. This is handled by package management tools, such as yum.

- Removal of an existing Driver Update Program package.
- Removal of a RHEL5 kernel package.

Additionally, depending upon the individual driver update, upgrades of individual updates to newer versions are also possible. For those drivers that affect the system at boot time - so-called bootpath drivers - there is automatic support for rebuilding the initrd images if the Driver Update Program detects that a newly installed driver update (or one used across a kernel upgrade) requires such an action to occur. You need do nothing additional in your packaging efforts in order to ensure that system bootpath handling takes place, since the scripts will automatically determine that your module is needed in the initrd (if this is indeed the case), by consulting the standard system initrd generation tools.

Installing a Driver Update

RHEL5 Driver Update Program packages sometimes come in pairs - the kmod itself and an accompanying “common” package that contains any userland dependencies, which should be satisfied prior to installation. Although it is no longer required, it is typical that, the “common” package include such components as documentation, firmware files, X Window System X.org graphics drivers, and anything else not an actual kernel module that is being shipped alongside the Driver Update Program package itself. The common package is automatically added as a kmod RPM dependency when using the standardized RHEL5 Driver Update Program packaging macros in a standard RHEL5 Driver Update Program SPEC file - although it is possible to build RPMs without this dependency (as is the case with several currently shipping RHEL5 driver updates).

You can install a RHEL5 Driver Update Program package as follows:

```
$ sudo rpm -ivh rpm/RPMS/i686/kmod-redhat-example-1.0-0.i686.rpm \
    rpm/RPMS/noarch/redhat-example-kmod-common-1.0-1.noarch.rpm
Preparing... ##### [100%]
1:redhat-example-kmod-com##### [ 50%]
2:kmod-redhat-example ##### [100%]
```

The driver module will be installed and symbolic links will be automatically added to compatible kernels installed on the target RHEL5 system. This happens automatically as part of the RPM post-install action via a call to the weak-modules (/sbin/weak-modules) script, with appropriate parameters (this is transparent to the end user) of the system). The system initrd will be updated, if it is so necessary.

Removing a Driver Update

In order to remove a driver update, it is necessary to remove both the Driver Update Program module, as well as any paired common package (if one exists). To do this, use the regular RPM package removal command:

```
$ sudo rpm -e kmod-redhat-example redhat-example-kmod-common
```

The Driver Update Program module will be removed and symbolic links in compatible kernels will be removed. If another (older) module provides a compatible driver update, it will replace the removed driver update in order to provide as seamless as possible experience for the user.

Installing a New Kernel Package

The RHEL5 Driver Update Program handles kernel updates in a transparent fashion. Adding a new kernel package will result in a call to the weak-modules (/sbin/weak-modules) script, which will create compatible symbolic links as necessary for any existing installed update packages. This was demonstrated with the symbolics links created in the section describing changes effected to the module-init-tools system package.

Removing a Kernel Package

The RHEL5 Driver Update Program handles removing kernel packages in a transparent fashion. Removing a kernel package will result in a call to the weak-modules (/sbin/weak-modules) script, which will remove compatibility symbolic links and ensure the removed kernel does not have any files remaining in /lib/modules/kernel_version post-uninstall. This is important because the symbolic links are not tracked in the system RPM database so will not automatically be removed at kernel uninstall time.

Upgrading Kernel Packages

In general, it is not possible to perform a regular “rpm -Uvh” type of upgrade on an installed kernel. This is because it is desirable always to keep at least one existing kernel installed before trying to use an updated one. Attempting to perform this kind of upgrade is not advised in this document - but do consult the RHEL5 documentation for further information. The Driver Update Program does not affect the existing situation in this regard.

Cross-vendor Packaging

Building your own Driver Update Program kmods has been streamlined, thanks to some joint work that has taken place between Red Hat and other Linux vendors. The result of this effort is a set of RPM macros that you can use in your own RPM SPEC files to quickly build driver updates, without any need to be concerned with the precise mechanics behind the process. An alternative packaging system is available for more advanced uses but is not documented in this version of the RHEL5 Driver Update Program whitepaper - if you require more advanced control over package building, see the kerneldrivers.org website.

A typical Driver Update Program SPEC file is very small indeed:

```
Name: redhat-example
License: GPL
Group: System/Kernel
Summary: Example RHEL5 Driver Update Program Package
Version: 1.5
Release: 0
Source0: %name-%version.tar.bz2
BuildRoot: %{_tmppath}/%{name}-%{version}-build
BuildRequires: %kernel_module_package_buildreqs
%kernel_module_package
%description
This is an example RHEL5 Driver Update Program Package.
%prep
%setup
set -- *
mkdir source
mv "$@" source/
mkdir obj
%build
export EXTRA_CFLAGS='-DVERSION=\"%version\"'
for flavor in %flavors_to_build ; do
rm -rf obj/$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
```

```
%clean
rm -rf %{buildroot}
%changelog
* Tue Oct 9 2007 Jon Masters <jcm@redhat.com>
- Updated original examples
```

The SPEC file makes use of the `%kernel_module_package` macro to do most of the necessary heavy lifting required and otherwise contains the regular meta-data associated with any other RPM package. This SPEC file is broadly compatible with other Linux vendors, although some small cosmetic differences may exist (these can be wrapped in conditional inclusion statements if a single SPEC file is desired).

The next section explains the actual packaging process in more detail.

Creating a kmod package

To create your own kmod package, begin by opening a new SPEC file (or use the example available on the kerneldrivers.org site). Give it a useful name, for example “redhat-example.spec” and add the necessary boilerplate metadata to the beginning of the file:

```
Name: redhat-example
License: GPL
Group: System/Kernel
Summary: Example RHEL5 Driver Update Program Package
Version: 1.5
Release: 0
Source0: %name-%version.tar.bz2
```

You should name your kmod according to the kernel module that it provides, including your company name if there is a potential for conflict with another RHEL5 Driver Update Program package installed on a customer system. For example, if your kmod contains a driver for the Intel e1000 network adapter NIC, called `e1000.ko`, then your kmod should include in its name `e1000`. The name you choose will be automatically prepended with the word “kmod-” in the final RPM packaged form in order to distinguish that this is a kmod package. You should also have a source file (named something like `redhat-example.tar.bz2`) that is accessible and unpacks into a directory suitable for building using the Linux kernel Kbuild infrastructure - any regular 2.6 Linux kernel “out of tree” module source should suffice for this (if you don’t have one available, you can use the example Red Hat test packages source as a guide to creating your own). The license field should be updated as appropriate to the driver in use.

Driver Update Program packages have some additional build requirements (such as the kernel-devel package) that differ between distributions, so the `%kernel_module_package_buildreqs` macro was created. You should add the following line to every Driver Update Program SPEC file:

```
BuildRequires: %kernel_module_package_buildreqs
```

%kernel_module_package

Most of the magic is encapsulated behind the `%kernel_module_package` macro. This is a standardized macro, which accepts several (optional) parameters, followed by a list of kernel flavors to build against:

- `-n` The name of the package. Can be used to override the name specified with “Name” in the SPEC file.
- `-v` A version number. Can be used to override the version specified with “Version” in the SPEC file.
- `-r` A release number. Can be used to override the release specified with “Release” in the SPEC file.
- `-s` A script used to generate the necessary RPM header output for the package. Defaults to `kmodtool`. You may need to replace the system-supplied version of `kmodtool` in various situations - for example, if you wish to remove the automatic dependency upon a common package, or if you intend to create a Driver Update Disk (an alternative version of `kmodtool` is available in the Driver Update Disk sources).
- `-f` A file containing a list of files that will form part of this package. Defaults to internally calculated.
- `-x` Invert the kernel flavor selection.
- `-p` A file containing alternative preamble for the RPM. Defaults to internal defaults.

For example, to specify that the Driver Update Program package should not be built for `kdump` kernels, you can specify:

```
%kernel_module_package -x kdump
```

Your SPEC file should also always include a description, used to document the purpose of your driver:

```
%description  
This is an example Kernel Module Package.
```

Setting up the module sources

To begin building your driver update, it is necessary to configure some module sources. This simply sets up your local build environment in order to build the RHEL5 Driver Update Program package on your system. Add this to the RPM SPEC file:

```
%prep
%setup
set -- *
mkdir source
mv "$@" source/
mkdir obj
```

Building the module sources

Building the driver update requires that each individual flavor be built (for example “default” (the UP/SMP kernel), “kdump”, “xen”), one at a time, in a loop similar to the following:

```
%build
export EXTRA_CFLAGS='-DVERSION=\"%version\"'
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
```

Note that %flavors_to_build is used in place of the specific kernel flavor (kernel variant). This is because specific kernel flavors (variants) vary from one distribution (and from one platform) to the next. The standard macro - defined by the standard RHEL5 distribution - allows you to abstract this difference and it’s partner - %kernel_source - allows you to quickly locate the actual location of the module sources, in a portable fashion. Across several different Linux distributions that support such compatible macro based packaging.

Installing the module sources

Installing the driver update binaries into the build root, so that they are properly available to be included in the resultant kmod requires another call to the kernel build system - in fact once for each flavor:

```
%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
```

Note again the call to %kernel_source in place of hard coding the location of the Linux kernel sources on your specific system.

Cleaning the module sources

You should always add a directive to any SPEC file, handling cleanup:

```
%clean
rm -rf %{buildroot}
```

Keeping track of changes

Finally, it's important to keep a changelog of your driver update package:

```
%changelog
* Tue Oct 9 2007 Jon Masters <jcm@redhat.com>
- Updated original examples
```

Alternative packaging process

Sometimes, the cross-vendor packaging process is less suited to a particular packaging situation. There is an older packaging process that is still in use, which is closer to the original Fedora kmod specification. In the spirit of promoting a cross-vendor standardized process, this document does not explicitly encourage or endorse that process, but you can find examples on the driverupdateprogram.com website. For example, the IPW3945 driver that ships in the base RHEL5 distribution makes use of the original packaging process.

Notes

Red Hat is supplying the RHEL5 Driver Update Program as a means for third parties to build and supply compatible updates to the RHEL5 kernel. Red Hat is not providing a mechanism for the delivery of such updates at this time, and are instead deferring such delivery to those parties who wish to supply Linux Kernel Modules (LKMs). There are a variety of ways in which to supply these, including custom yum repositories as well as individual RPM packages located on a website or physical medium for factory pre-install or end user installation on RHEL5 systems. Although Red Hat does not provide a mechanism for partner distribution via RHN at present, Red Hat is interested in hearing from partners with specific requests.