Kernel Module Packages Manual for CODE 10

March 6, 2006

This document specifies the requirements for rpm packages that contain kernel modules, and describes the processes surrounding those packages including building, signing, installing, and upgrading. A complete example is given and explained.

This version of the Kernel Module Packages Manual applies to the Novell/SUSE CODE 10 code base, which includes SUSE Linux 10.1, SLES 10 (SUSE Linux Enterprise Server), as well as all products based on SLES 10. An second version of this document for CODE 9 is available as well; the CODE 9 version covers SLES 9 and SUSE Linux 10.0.

A few issues related to the new package manager and dependency resolver library still need to be tested and documented here. The relevant places in this document are highlighted like this paragraph.

Introduction

The Linux kernel supports adding functionality at runtime through kernel loadable modules. It includes more than 1500 modules, about 75 percent of which are hardware drivers. These modules are shipped as part of the kernel packages. In some cases it is desirable to add additional modules or replace existing ones. (For example, a driver for a particular storage controller that was not available at the time of product release might be added later in order to support new hardware.)

Kernel modules interact with the kernel by the means of exported symbols, in a way similar to how binaries use shared libraries.[1] To ensure that the kernel and modules refer to the same symbols, a version checksum (aka modversion) is added to each symbol that is computed from the symbol's type: in the case of function symbols, the checksum is determined by the function's parameters and return type.

When any of a function's parameters or the return type changes, the checksum changes as well. This includes all the data types involved recursively: if a function takes a struct task_struct as parameter and struct task_struct includes a field of type struct dentry, then a change to struct dentry will cause the symbol's version checksum to change as well. Symbol version checksums for different kernel flavors (e.g., kernel-default vs. kernel-smp) will not match, and symbol versions of the same kernel package on different architectures (e.g., kernel-default on i386 vs. x86_64) will not match, either. This mechanism ensures that the kernel and kernel modules agree on the types of data structures that they use to communicate.

Unless symbol version checking is disabled, modules will only load if the checksums of the symbols they use match the checksums of the symbols that the kernel exports. The exported symbols and their version checksums comprise the kernel Application Binary Interface (ABI). When an update kernel includes kernel ABI changes, kernel modules that use any modified symbols must be updated as well.

1 The /proc/kallsyms file lists all symbols currently known to the kernel.

During its multi-year life cycle, products like SUSE Linux Enterprise Server (SLES) undergo continuous changes, and different kinds of updates like Service Packs (SPs), maintenance / security updates, and customer-specific updates (PTFs = Program Temporary Fixes) are released. The Application Binary Interface (ABI) between the kernel and kernel modules is volatile, and some kernel updates will change the kernel ABI by adding or removing exported symbols, or existing symbol checksums may change because of changes in data structures they reference. We strive to keep the kernel ABI stable in maintenance / security and customer-specific updates, but sometimes we cannot avoid changes. In Service Packs, we reserve the right to introduce more intrusive changes, which increases the likelihood of ABI changes. We believe that the added flexibility outweighs the disadvantages of breaking older modules. Please see kABI Stability in SLES [1] and The Linux Kernel Driver Interface [2] for discussions of this topic.

All Linux products that Novell/SuSE offers primarily use the RPM Package Manager for software management. This also applies to kernel modules, which must be packaged following a number of rules that ensure that the resulting Kernel Module Packages (which are also referred to as Kernel Driver Packages) can be installed and updated appropriately, in sync with kernel updates.

This document specifies Kernel Module Packages, describes how they fit into the build system, and how to create, sign, and deploy them.

1 Kernel Packages

All Novell/SUSE products based on 2.6.x kernels contain a set of kernel packages that all share the same version and release number; they are built from the same kernel sources. These packages are:

kernel-*flavor*

A binary kernel package. Each architecture has its own set of kernel flavors (e.g., kernel-default, kernel-smp, kernel-bigsmp, kernel-um, kernel-xen). These are the packages that the kernel modules will be used with.

kernel-source

The kernel source tree, generated by unpacking the vanilla kernel sources and applying all necessary patches. While the kernel-*flavor* packages technically are not built from the kernel-source package, they are built from the same source tree. This tree should be used for module building.

kernel-syms

Kernel symbol version information for compiling external modules. This package is *required* for building external modules. If this package is not used, the resulting modules will be missing symbol version information, which will cause them to break during kernel updates. The kernel-source and kernel-syms packages used for compiling external modules must match each other exactly.

Please refer to Working With The SUSE 2.6.x Kernel Sources [3] for more information.

2 Kernel Modules

Documentation on general kernel module building can be found in abundance on the Internet [5, 6]. Novell/SUSE specific information is found in Working With The SUSE 2.6.x Kernel Sources [3].

Once built, kernel module binaries are installed below /lib/modules/*version-release-flavor* on the file system (e.g., /lib/modules/2.6.16-97-default for the kernel-default-2.6.16-97 package). Different kernels have different module directories, and will not usually see each other's modules.

Update modules must be stored below the /lib/modules/*version-release-flavor/*updates/ directory. Modules in the updates/ directory have precedence over other modules with the same name. Never replace modules from the kernel package by overwriting files: this would lead to inconsistencies between the file system and the rpm database.

Modules usually remain compatible with a range of kernel-*flavor* packages. To make such modules visible to other kernel-*flavor* packages, symbolic links to compatible modules are put in /lib/modules/*version-release-flavor*/weak-updates/ directories. Modules in the weak-updates/ directory has lower priority than modules in the updates/ directory, but higher priority than all other modules in /lib/modules/*version-release-flavor*. If more than one compatible module is available for a kernel, the module with the highest kernel release is chosen.

Kernel modules must never be installed as individual files on a production system, and always as part of a Kernel Module Package.

3 Kernel Module Packages

Kernel Module Package spec files define a main package, and a sub-package for each kernel flavor supported. The kernel-specific sub-packages are defined with the %suse_kernel_module_package rpm macro. The macro automatically determines for which kernel flavors to generate sub-packages. Several options are available to modify the macro's behavior, which are described below:

%suse_kernel_module_package [-s *subpkg*] [-f *filelist*] [-p *preamble*] [-n *name*] [-v *version*] [-r *release*] [-x] [*flavor*] ...

The main package of a Kernel Module Package can either contain no %files section, in which case rpm will not create a binary package with the main package's name, or it can be used for the user-space part associated with the kernel modules that end up in the kernel specific sub-packages. (The example Kernel Module Package in Appendix A has a main package without a %files section.)

Kernel Module Packages must adhere to the following rules:

- The package **Name** should consist of two components: a unique provider prefix, and a driver name. Hyphens are disallowed in the provider prefix, and allowed in the driver name. The provider prefix serves to create a non-overlapping name space for all providers.

  The sub-package names is composed of the main package name, followed by a dash, and the flavor of the supported kernel. The first component (main package name) can be overridden with a different value with the **-n** option of the %suse_kernel_module_package macro.

- The kernel module package **Version** can have an arbitrary value.

  The sub-package versions are composed of the main package version, followed by an underscore, and the version of the kernel supported. Since sub-packages already include the supported kernel's flavor in their name, the flavor is not again included in the sub-package's version. Dashes in the kernel release are replaced by underscores. The first component (main package version) can be overridden with the **-v** option of the %suse_kernel_module_package macro.

- The kernel module package **Release** can be assigned freely as required. It must be incremented at least once for each package release.

The sub-package release numbers equal the main package's release number. It can be overridden with the **-r** option of the %suse_kernel_module_package macro.

- The appropriate **Requires** and **Provides** tags are computed automatically by rpm as described in the RPM Provides And Requires section below. Requires and Provides tags in the spec file will only be effective for the main package.
- Kernel modules must be installed below /lib/modules/*version-release-flavor*/updates/.
- Packages must be signed with a public/private key pair, and the public key of the private/public key-pair used for signing must be made known to rpm. See the Signing Kernel Module Packages section below for details.

The spec file must define a %name-KMP sub-package as shown in the example spec file in appendix A. This sub-package must have a Summary and Group tag, and a %description section. The summary and group tags, and description, are used as the summary, group, and description of each kernel-specific sub.package that is created.[1]

The %suse_kernel_module_package macro uses a default sub-package template that should work for most Kernel Module Packages. This template can be overridden using the macro's -s option. The default template takes care of the following:

- When a Kernel Module Package package is installed, depmod is called to update module dependency information and various maps. Symlinks pointing at the new modules are created in other kernels' weak-modules/ directories for all compatible modules. Initial ramdisks used during booting are recreated if they contain some of the added modules.
- When a Kernel Module Package us removed, depmod is called to update module dependency information and various maps. The symlinks pointing to the modules being removed are removed as well. Initial ramdisks are recreated in case they did contain some of the removed modules.

By default, each kernel-specific sub-package will have the following list of files, which can separately be overridden with the -f option:

```
%defattr (-,root,root)
/lib/modules/%2
```

Additional sub-package preamble lines such as Requires, Provides, and Obsoletes tags can be specified with the -p option. Filename arguments specified in -s, -f, and -p should be given as absolute path names (e.g., %_sourcedir/file). The following substitutions are defined in those files:

%1	Flavor of the sub-package (e.g., default).
%2	Kernel release string (e.g., 2.6.16-97-default).
%3	Kernel release string without flavor (e.g., 2.6.16-97).
%{-v*}	The sub-package version.
%{-r*}	The sub-package release.

Some Kernel Module Packages may make sense only for some of the kernel flavors a given architecture supports. A list of flavors to exclude from the build can be passed to the %suse_kernel_module_package macro. The meaning of this list can be inverted with the -x option: with -x, the flavors mentioned will be included. The example Kernel Module Package spec file in Appendix A excludes the kdump and uml flavors.

1 For Novell/SUSE employees: Autobuild will replace these fields by whatever data exists in the Package Database. This mechanism ensures that the sub-package description is translated as for any other package.

Appendix A contains an example spec file for a Kernel Module Package. When this spec file is built on x86_64 as described in section Building Kernel Module Packages below, the BuildRequires tag in the spec file will pull the kernel-source and and kernel-syms packages into the build root. Let us assume that the two packages are available in version/release 2.6.16_rc1-2, and that the default, kdump, smp, and xen kernel flavors are available on that platform. RPM would then create the following packages:

novell-example-kmp-default-1.1_2.6.16_rc1_2-0.x86_64.rpm
novell-example-kmp-smp-1.1_2.6.16_rc1_2-0.x86_64.rpm
novell-example-kmp-xen-1.1_2.6.16_rc1_2-0.x86_64.rpm

The generated packages would contain the following modules, and require and provide the following symbols:

Package	**Requires**	**Provides**
novell-example-kmp-default	kernel kernel(built_in) = 368e76d2ad800f86	novell-example-kmp = 1.1_2.6.16_rc1_2-0 novell-example-kmp-xen = 1.1_2.6.16_rc1_2-0 ksym(exported_function) = 2abca7b2
Modules	/lib/modules/2.6.16-rc1-2-default/updates/moo.ko	
novell-example-kmp-smp	kernel kernel(built_in) = 6f1c8e11b913710f	novell-example-kmp = 1.1_2.6.16_rc1_2-0 novell-example-kmp-xen = 1.1_2.6.16_rc1_2-0 ksym(exported_function) = 85e1229d
Modules	/lib/modules/2.6.16-rc1-2-smp/updates/moo.ko	
novell-example-kmp-xen	kernel kernel(built_in) = e916ffe914f865fb	novell-example-kmp = 1.1_2.6.16_rc1_2-0 novell-example-kmp-xen = 1.1_2.6.16_rc1_2-0 ksym(exported_function) = e52d5bcf
Modules	/lib/modules/2.6.16-rc1-2-xen/updates/moo.ko	

4 RPM Provides And Requires

As stated in the introduction, kernels export symbols that kernel modules use. Symbols have version checksums attached, and the checksums of the exported kernel symbols must match the checksums of the used kernel symbols. These dependencies are mapped to symbols that the kernel packages provide and that Kernel Module Packages require at the rpm package level.

Typical packages only require and/or provide a handful of symbols, so package managers are not designed to handle hundreds and thousands of package dependencies. Therefore, the kernel and Kernel Module Packages do not provide/require each kernel symbol individually. Instead, symbols are grouped together into classes. The classes are computed when the kernel packages are built; the number of classes is much smaller than the number of symbols. The packages then provide and require these symbol classes instead of individual symbols.

When modules in Kernel Module Packages export additional symbols, those symbols are not in an existing class. Such symbols are mapped to per-symbol provides of those packages. Modules in other Kernel Module Packages may require those symbols; they would also do so on a per-symbol basis.

As an example, a kernel package would provide the following symbols:

kernel(vmlinux) = 0bbceb16fcab3f0e
kernel(drivers_net) = abbc3a2161979123
kernel(built_in) = f63b376dfb966bec

A matching Kernel Module Package would then require kernel(built_in) in the same version. It would provide exported functions as ksym(exported_function) = e52d5bcf or similar.

5 Building Kernel Module Packages

In addition to the C and kernel programming skills required for writing the kernel module source code in the first place, creating proper Kernel Module Packages requires some familiarity with rpm and with build environments. Those who are looking for more information on kernel module building may find the Linux Kernel Module Programming Guide [5] and the Linux Device Drivers book [6] interesting. Additional Novell/SUSE specific kernel and kernel module information can be found in Working With The SUSE 2.6.x Kernel Sources [3]. We recommend to take the example package found in the kernel-source package as a template to reduce the complexities related to rpm. A lot of additional information on rpm can be found at http://www.rpm.org/, including an online version of the excellent Maximum RPM.

We strongly recommend to use the kernel build infrastructure (kbuild) for building and installing the kernel modules, as done in the example package. Kbuild is documented in /usr/src/linux/Documentation/kbuild/ from the kernel-source package. Trying to emulate kbuild will lead to various problems including mis-compilations and missing or wrong symbol versions, and increased support load due to subtle breakage.

In order to achieve consistent and reproducible builds in a defined environment independent of the software installed on the system used for building, we recommend to use the build script from the build.rpm package.[1] This script sets up a build environment from the rpm packages the script is pointed at. The packages are then built in this environment using chroot (see the chroot(1) manual page). All Novell/SUSE packages are built using the same mechanism. When building Kernel Module Packages with build.rpm, the following options of the build script are particularly relevant:

--root *directory*

Define the directory in which to set up the build environment. Defaults to the BUILD_ROOT environment variable, and to /var/tmp/build-root if unset.

--rpms *path1*[:*path2*:...]

Define where build will look for packages for constructing the build environment.[2] The directories are searched recursively. Packages found earlier in the path have precedence over packages found later, similar to how the PATH environment variable works. Defaults to the BUILD_RPMS environment variable, and to /media/dvd/suse if unset. The --rpms option must only be specified once.

1 For Novell/SUSE employees: the build script works similarly to Autobuild's build and mbuild scripts. It is slightly easier to use Autobuild instead of build.rpm's build script where you can; this automatically gives you the most up-to-date packages in the build environments.

2 The SUSE/Novell internal build script fetches the packages over the network based on the distribution specified (--dist). Mbuild also accepts a set of distributions (--distset). The package selection can be influenced using the --prefer-rpms option.

--clean, --no-init

Reconstruct the build environment entirely from scratch (--clean), or start the build without initializing the build environment (--no-init) , which skips the check whether all packages in the build environment are up to date.

Build stores the created packages below usr/src/packages/ in the build environment.

On dual-architecture machines, packages for the other supported architecture can be built by running the build script inside an architecture selector. On x86_64, the selector is called *linux32,* on ppc64 this is *ppc32,* and on s390x the selector is called *s390.* The same build environment cannot be reused for different architectures unless it is reinitialized with build's --clean option.

See the build(1) manual page and Novell articles on build [7, 8] for further information.

Please Note

For building external modules, you need to have both the *kernel-source* and the matching *kernel-syms* package installed in the build environment; the BuildRequires line in spec files takes care of this. Without *kernel-syms* the module build may still succeed depending on how you do the build, but the resulting modules will have module symbol versions disabled. Kernel Module Packages without module symbol versions will appear to match any kernel while in fact they do not. This can easily lead to very hard to diagnose system malfunctions.

6 Signing Kernel Module Packages

Before packages are deployed, they should be signed using GNU Privacy Guard (GPG). Signing packages allows customers to define which parties they trust for producing packages for them. The package manager will refuse to install unsigned packages.

In order to sign packages, a private/public key pair must be installed on the GPG keyring of the signing user (see the --gen-key option in the gpg(1) manual page). Then, the following command can be used to sign a package (replace *build@novell.com* with the identity that identifies your signing key):

```
$ rpm --eval "%define _signature gpg" \
      --eval "%define _gpg_name build@novell.com" \
      --addsign package.rpm
```

Note that a package can only be signed once. Another --addsign operation will replace an existing old signature, and will add the new one.

The public key used for signing must then be exported into a file with:

```
$ gpg --armor --export build > build-pubkey.txt
```

Then, import the key into the rpm database with:

```
$ rpm --import build-pubkey.txt
```

You can verify that both package signing and key import have succeeded with rpm's --checksig option (note the "gpg" in the output):

```
$ rpm --checksig package.rpm
package.rpm: (sha1) dsa sha1 md5 gpg OK
```

The public key exported to build-pubkey.txt must be delivered to customers in a way that they will trust. It must be imported into the rpm database on systems on which the signed packages shall be installed.

7 Deploying Kernel Module Packages

Kernel Module Packages must first be installed on target systems, either from an update medium, or by installing the package.

8 System Installation and Kernel Module Packages

Initial system installation is carried out by YaST from some installation media (CDs or DVDs, network locations, etc.). Support for additional hardware that the installation media do not provide can be added with update media. This is most important to enable hardware needed for booting, such as storage controllers.

The update media (aka Driver Update Disks) contain two kinds of modules: those which the kernel that runs the installation uses, and those which are installed on the target system. The latter should be put on update media as rpm packages. In addition to modules and Kernel Module Packages, update media may contain scripts which are run at specific times during the installation. The Update Media HOWTO [4] describes in more detail what Novell/SUSE Update Media must contain in order to work.

We need to fix the way in which update media install rpms located on them: right now, they are all installed, disregarding dependencies. We must make sure to only install rpms that have no unresolved dependencies.

We do not yet know how to register an installation source, either from an update medium, or by hand.

After the initial YaST installation, additional driver packages can be installed using any of the mechanisms for installing rpm packages (the YaST Package Manager, YOU Online Updates, the rpm command, etc.).

Please note that drivers that are required for getting to and accessing the root file system must be part of the initial ramdisk (initrd). YaST will automatically include necessary kernel modules in the initrd created during installation, but when Kernel Module Packages are installed by hand or updated, this needs to be taken care of. Such drivers may also need to be added to the INITRD_MODULES variable in /etc/sysconfig/kernel.

9 Kernel Updates and Kernel Module Packages

After all installation sources that should be checked for updates have been added, the package manager will automatically detect when new kernel packages as well as new Kernel Module Packages become available. The dependencies between those packages will ensure that the installed kernel packages match the installed kernel module packages.

Note that the package manager described above, and zypp, the underlying dependency resolver library, is not yet finished. We have yet to see if they will work correctly, offer packages for update as they become available, and handle conflicts properly.

10 Querying Installed Kernel Module Packages

Each kernel module package requires the “kernel” symbol. RPM can be used to get a list of installed kernel module packages with:

```
$ rpm -q --whatrequires kernel
novell-example-kmp-smp-1.1_2.6.16_rc1_2-0
```

Appendix A: novell-example.spec

This example spec file is described in the Kernel Module Packages section. It can be found in the kernel-source package.

```
# norootforbuild

Name:         novell-example
BuildRequires: kernel-source kernel-syms
License:      GPL
Group:        System/Kernel
Summary:      Example Kernel Module Package
Version:      1.0
Release:      0
Source0:      %name-%version.tar.bz2
BuildRoot:    %{_tmppath}/%{name}-%{version}-build

%suse_kernel_module_package kdump um

%description
This is an example Kernel Module Package.

%package KMP
Summary: Example Kernel Module
Group: System/Kernel

%description KMP
This is one of the sub-packages for a specific kernel. All the
sub-packages will share the same summary, group, and description.

%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 /usr/src/linux-obj/%_target_cpu/$flavor modules \
      M=$PWD/obj/$flavor
done

%install
export INSTALL_MOD_PATH=$RPM_BUILD_ROOT
export INSTALL_MOD_DIR=updates
for flavor in %flavors_to_build; do
    make -C /usr/src/linux-obj/%_target_cpu/$flavor modules_install \
       M=$PWD/obj/$flavor
done

%changelog
* Sat Jan 28 2006 - agruen@suse.de
- Initial package.
```

Changes

January 27, 2006

- Take the CODE 9 Kernel Module Packages Manual, and adapt it to the CODE 10 process. A lot has changed.
- Incorporate some feedback from discussions to the rpm signing section.

February 2, 2006

- Document the new %name-KMP sub-package used to define the sub-packages' summary, group, and description.
- Document %suse_kernel_module_package's new -f option for defining an alternate list of files.
- Update the spec file example.

February 3, 2006

- Change the naming convention of the kernel-specific sub-packages from name-*flavor* to name-kmp-*flavor*. The spec files do not need to be adapted for that; all the logic is in the rpm package.

March 6, 2006

- Document additional %suse_kernel_module_package options.

References

[1] Kurt Garloff et al.: kABI Stability in SLES, http://www.suse.de/~agruen/kabi (Temporary location; please contact Kurt Garloff <garloff@suse.de> in case this URL has become unavailable.)

[2] Greg Kroah-Hartman: The Linux Kernel Driver Interface, http://www.kroah.com/log/linux/stable_api_nonsense.html

[3] Andreas Grünbacher: Working With The SUSE 2.6.x Kernel Sources, http://www.suse.de/~agruen/kernel-doc/

[4] Update Media HOWTO, ftp://ftp.suse.com/pub/people/hvogel/Update-Media-HOWTO

[5] Peter Jay Salzman, Michael Burian, Ori Pomerantz: The Linux Kernel Module Programming Guide, http://www.tldp.org/LDP/lkmpg/2.6/html/index.html.

[6] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman: Linux Device Drivers, Third Edition, February 2005, http://www.oreilly.com/catalog/linuxdrive3/. Also available online at http://lwn.net/Kernel/LDD3/.

[7] Cory Aitchison: Building Packages for Novell's Linux Products, http://developer.novell.com/ndk/whitepapers/buildrpm.htm.

[8] build.rpm: Reproducible Build and Test Cleanrooms for SUSE LINUX (White Paper), April 2005, http://www.novell.com/collateral/4621440/4621440.pdf.