Kernel Module Packages Manual for CODE 9

January 27, 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.

The approach described here is specific to SUSE Linux 10.0 and the SUSE Linux Enterprise Server 9.

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.

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

1 In 2.6 kernels, the /proc/kallsyms file lists all symbols currently known to the kernel.

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. Our belief is that the added flexibility pays off the disadvantage 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.

Support for kernel module packages as described in this document has been introduced with SUSE Linux 10.0, and has been back-ported to SLES 9 as well (in SLES 9, this functionality is provided as online updates). Future products will continue to offer the same features in a similar way. Due to improvements in the package manager, we will be able to further improve this process; see Section Future Products for some more details.

This document outlines the installation and update work flow as it relates to Kernel Module Packages, describes how those packages are created, specifies the rules that they must follow, and shows how installed Kernel Module Packages can be managed.

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 all 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 the patches. 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.

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/2.6.5-7.241-default for the kernel-default-2.6.5-7.241 package). Different kernels have different module directories, and will not see each other's modules.

Update modules must be stored below the /lib/modules/*version-release-flavor/*updates/ directory. Do not replace modules from the kernel package by overwriting files: this would lead to inconsistencies between the file system and the RPM database. Modules in the updates/ directory have precedence over other modules with the same name. The kernel will recognize new modules only after depmod was run, so Kernel Module Packages (see the next section) must run depmod in their %post and %postun scripts.

During kernel updates, the kernel-update-tool will make modules available to other kernels for reuse as described in section “Reuse Of Existing Kernel Module Packages”.

3 Kernel Module Packages

Kernel Module Packages contain modules for one or more kernel packages, meta information that allows to retrieve update packages during kernel updates, and scripts that control their installation and removal. Kernel modules are never installed as individual files, and always as part of one or more Kernel Module Packages.

These packages can support a single kernel flavor, or a set of kernel flavors: a single-flavor kernel module package supports a single flavor of a kernel release (e.g., kernel-default-2.6.5-7.193), while a multi-flavor kernel module package supports several flavors of a single kernel release (e.g., kernel-*flavor*-2.6.5-7.193 for the flavors default, smp, bigsmp, um, xen on the x86 architecture). Since it is almost as easy to build a multi-flavor kernel module packages as it is to build a single-flavor kernel module package, kernel module packages are expected to be reasonably small, and it is easier to deploy and manage few packages, we recommend to use multi-flavor packages.

Kernel Module Packages must follow a number of conventions:

- The package **Name** must consist of two components: a unique provider prefix, and a driver name. Hyphens are allowed in the driver name, and disallowed in the provider prefix. The provider prefix serves to create a non-overlapping name space for all providers.
- The kernel module package **Version** must contain the driver version, followed by an underscore and the kernel version as reported by ``uname -r''. The driver version must not contain any hyphens or underscores. Hyphens in the kernel version must be replaced by underscores. See the template .spec file in Appendix A for how the version number can be computed at package build time.
- The kernel module package **Release** can be assigned freely as required. It must be incremented at least once for each package release.
- The kernel module package must **Require** the kernel it supports. Single-flavor packages must require “kernel-*flavor* = *version-release*” (e.g., “kernel-default = 2.6.5-7.193”). Multi-flavor packages must require “kernel = *version-release*” tag (e.g., “kernel = 2.6.5-7.193”).
- Modules must be installed below /lib/modules/*version-release-flavor*/updates/. Modules installed below this path have precedence over modules installed below other paths in /lib/modules/*version-release-flavor*/. This allows to override modules without overwriting files. Make sure to run depmod after changing the contents of /lib/modules/*version-release-flavor*/updates/ (see the example spec file).
- The kernel update tool expects kernel module packages to 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.

- To allow automatic kernel module package downloads upon kernel updates, the module package must contain a file /var/lib/YaST2/download/*module_package_name*. This file contains the download location for module packages for future kernel releases. See Section Kernel Module Package Download below.
- Kernel Module Packages may be reused for an update kernel if they are compatible. When that happens, the %pre, %preun, %post, and %postun scripts from the previous kernel's module package are used for the new kernel. These scripts should refer to the specific kernel releases they support only by defining the KERNELRELEASES variable on an otherwise empty line. When a module package is reused, the definition of this variable will be updated.

  For example, the definition in line one below will be replaced by line two when package novell-kmp for kernel 2.6.11.4-20a is reused for kernel 2.6.11.4-21.7:

```
KERNELRELEASES="2.6.11.4-20a-default 2.6.11.4-20a-smp"
KERNELRELEASES="2.6.11.4-21.7-default 2.6.11.4-21.7-smp"
```

Appendix A contains an example rpm spec file for a kernel module package that can be either single-flavor or multi-flavor, depending on the definition of *%flavor*.

4 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-update-tool 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.

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.

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

Define where build will look for packages for constructing the build environment.[1] 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.

--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 #neededforbuild line in spec files takes care of this. Without *kernel-syms* the module build will still succeed, but the resulting modules will have module symbol versions disabled. The kernel-update-tool inherently relies on symbol versions. 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.

5 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 kernel update tool will complain about unsigned packages. It will currently ask the user how to proceed when it is pointed at an unsigned package, and it will allow to install packages even when they are unsigned. This policy may change in the future.

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
```

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

1 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.

```
$ 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 on systems on which the signed packages shall be installed.

6 Deploying Kernel Module Packages

Kernel Module Packages must first be installed on target systems, either from an update medium, or by simply installing the package. Later during kernel updates, packages for successive kernel releases will be fetched from the download location specified in /var/lib/YaST2/download/*module_package_name.*

The download mechanism used during kernel updates requires a repository of packages that is laid out as described in Kernel Module Package Download below. Downloads are possible from ftp, http, or https URLs. Please keep in mind that Kernel Module Package Download is not an optional part of the process: We cannot guarantee that the kernel ABI will not change, so whenever it does, only the download mechanism ensures that users will be able to update to the new the kernel and keep the Kernel Module Package at the same time.

Please keep in mind that before packages can successfully be installed, the public key used for package signing must be installed in the rpm database of the system that perform the updates. See the Signing Kernel Module Packages section.

7 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.

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.

See Appendix C for an example script that can be used in Kernel Module Packages for recreating the initrd when necessary.

8 Kernel Updates and Kernel Module Packages

Kernel updates are ultimately performed by the kernel update tool. When doing a kernel update, the kernel update tool makes sure that all hardware supported by the old kernel will also be supported by the new kernel. It goes through the list of Kernel Module Packages for the current kernel and for each package it checks:

- Does the new kernel support these kernel modules internally?
- Is a driver package that fits the new kernel installed already?
- Can such a driver package be downloaded?
- Can the old kernel's driver package be reused?

When looking for a suitable replacement for a Kernel Module Package, the kernel update script only considers packages with the same name.

If no suitable packages for the new kernel are available and the user did not choose to skip packages for which there is no successor, the kernel update tool will abort the update. This leaves the system with the old kernel, instead of leaving the user with a broken system. Alternatively, the user can choose to skip a driver package, and thus give up support for a particular piece of functionality. This may be preferable to not being able to update to the new kernel.

The kernel update tool can be configured in /etc/sysconfig/onlineupdate to always or never download module packages, or to ask the user. It can also be configured to always or never reuse module packages, or again ask the user.

When kernel updates are carried out with YOU, the kernel update tool opens a log window where it shows its progress while it runs (Figure 1). Whenever a user interaction is required, a dialog pops up and prompts the user for his choice (Figure 2).

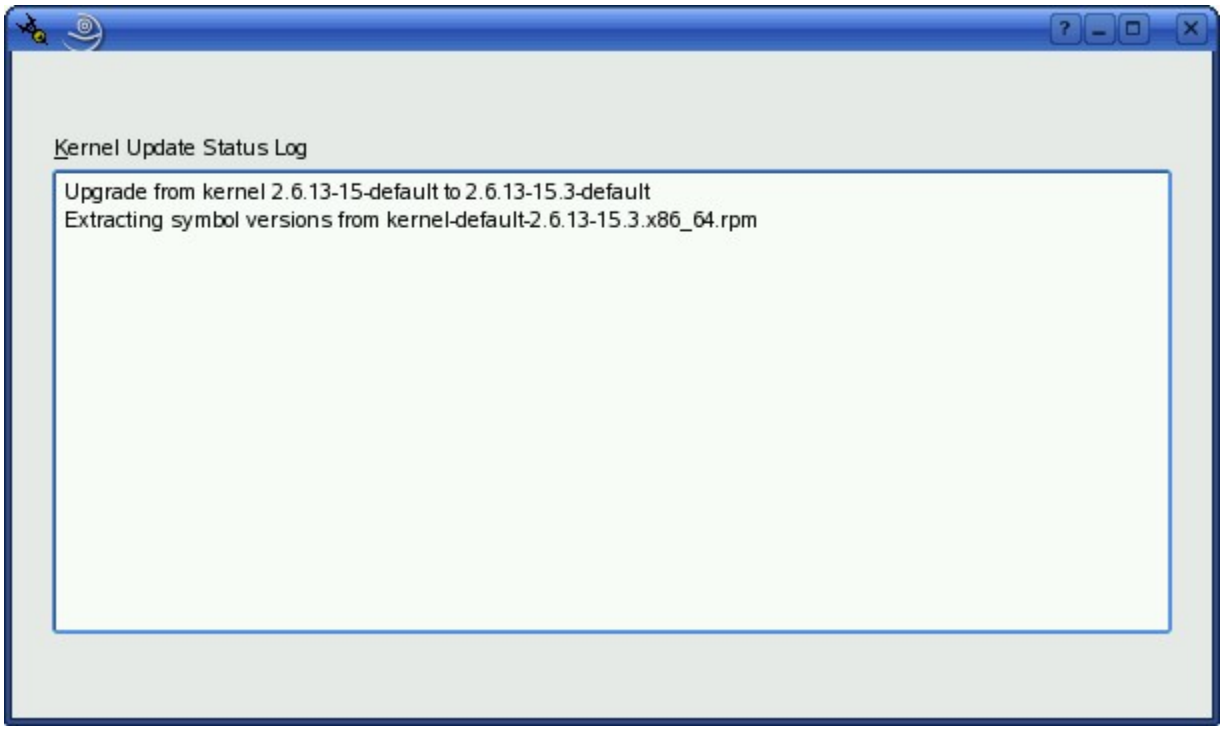

Figure 1: Log window of the kernel update tool

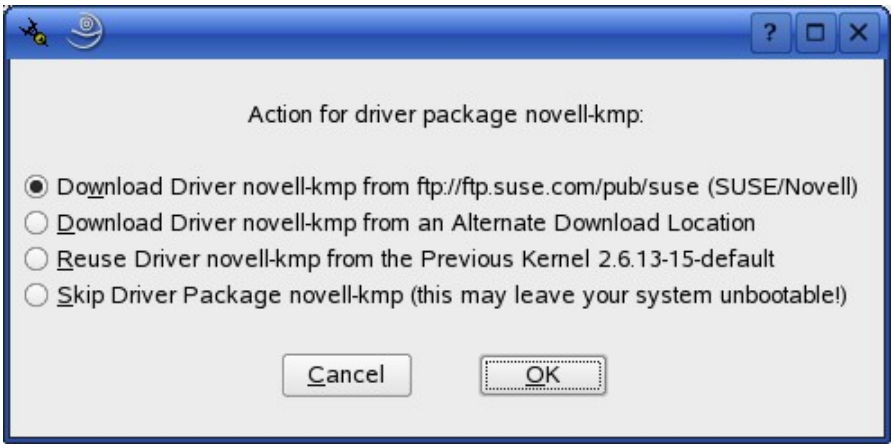

Figure 2: Pop-up prompting for user feedback during kernel updates

The windows shown in Figures 1 and 2 are also available in a text form when YOU is used in a terminal. The kernel-update-tool script itself can also be used directly from the command line. See the kernel-update-tool package for details.

9 Kernel Module Package Download

One of the choices the kernel update tool offers is to try downloading a version of a module package for the new kernel. To allow this to happen, the kernel update tool must know where to look for that package. To that end, each kernel module package contains a file /var/lib/YaST2/download/*module_package_name*. This file contains a single line specifying a URL, optionally followed by a semicolon and a description of the download location, for example:

ftp://ftp.suse.com/pub/suse;Novell/SUSE

The complete download URL for the package is computed as follows:

- Take the URL from /var/lib/YaST2/download/*module_package_name* as a prefix.
- Add the YouPath from /var/adm/YaST/ProdDB/prod_*. A product may consist of more than one sub-product, in which case all paths are tried.

  For SUSE Linux 10.0, the YouPath will be *i386/update/10.0*. For SLES 9, the two product paths are *i386/update/SUSE-CORE/9* and *i386/update/SUSE-SLES/9* for the core and add-on product, respectively. There is no rigid algorithm for computing the paths for arbitrary products; please look up the correct paths in the prod_* files.
- Add “rpm/${architecture}/${module_package_name}-${kernel_release}.${architecture}.rpm”. The kernel_release field always includes the kernel version, release, and flavor. The driver version is *not* included.

Note that the module package release numbers are not included in the URLs that are tried, while rpm includes the package release numbers in the files it creates. We recommend to put module packages on the download server(s) with the filenames that rpm generates, and use symlinks to point to the packages which the kernel update tool shall use. Appendix contains a script which takes a tree of kernel module packages with their original rpm filenames, and creates symlinks so that the kernel update tool will find the packages via these symlinks.

As an example, the resulting download URL that is tried for kernel module novell-kmp version 1.0, for kernel-smp version 2.6.11.4 release 21.7 on SUSE Linux 10.0 for x86_64 with download prefix `ftp://ftp.suse.com/pub/suse` is:

```
ftp://ftp.suse.com/pub/suse/i386/update/10.0/rpm/x86_64/
      Novell-kmp-2.6.11.4_21.7_smp.x86_64.rpm
```

10 Reuse Of Existing Kernel Module Packages

When the kernel update tool decides to reuse an existing kernel module package, it repackages the files from the old package. The modules are moved to the new kernel's module load path, and the old packages' %pre, %preun, %post, and %postun scripts are reused as described in the Kernel Module Packages section. It then installs the resulting package. Reused packages will have both the original and the new kernel's version in their package name.

For example, after upgrading from kernel-smp-2.6.11.4_20a to kernel-smp-2.6.11.4-21.7 with module package novell-kmp-1.0_2.6.11.4_20a-0, when the user chooses to reuse the old novell-kmp package, the rpm database will contain a novell-kmp-1.0_2.6.11.4_20a_for_2.6.11.4_21.7-0 package.

11 Querying Installed Kernel Module Packages

The kernel update tool can be invoked from the command line to collect kernel module status information. The output shows the kernel version, module packages, and modules of each module package, for example:

```
$ /usr/sbin/kernel-update-tool --status
Kernel version: 2.6.11.4-20a-smp
Driver package: novell-kmp-1.0_2.6.11.4_20a-0
Module: updates2/moo.ko (novell-kmp-1.0_2.6.11.4_20a-0)

Kernel version: 2.6.11.4-21.7-smp
Driver package: novell-kmp-1.0_2.6.11.4_20a_for_2.6.11.4_21.7-0
Module: updates2/moo.ko (novell-kmp-1.0_2.6.11.4_20a_for_2.6.11.4_21.7-0,
2.6.11.4-20a-smp)
```

This shows that two kernels are installed on this system in parallel. For the first kernel, the novell-kmp kernel module package is installed. The second kernel is reusing the first kernel's module package.

The rpm utility can also be used to get a list of installed kernel module packages:

```
$ rpm -q --whatrequires kernel
novell-kmp-1.0_2.6.11.4_20a-0
novell-kmp-1.0_2.6.11.4_20a_for_2.6.11.4_21.7-0
```

12 Future Products

This document specifies kernel module packages for SLES 9 and SUSE Linux 10.0. This specification is designed with the limitations of the current package manager and YaST Online Update (YOU) in mind. A future version of the package manager that integrates package management and updates is in the works. This version will finally allow to register multiple sources of packages / updates, and will be better at dealing with dependencies between updates. A future version of kernel module packages will take advantage of these improvements.

While the details are not yet finalized, we may replace the kernel module package download mechanism. Download locations and update selection may be integrated into the new package manager. This will require some meta-information in the download repositories that currently is not required, and the layout could change. We have not yet fully defined how package reuse will be implemented.

Appendix A: novell-kmp.spec

This spec file can be used as a multi-flavor or single-flavor kernel module package template. When reusing, please choose one of the two variants, and remove the code for the other variant. This template is part of the kernel-update-tool package, and can be found in /usr/share/doc/packages/kernel-update-tool/novell-kmp/.

```
#neededforbuild kernel-source kernel-syms

# Change either of these definitions to define flavor. If flavor is
# non-nil, this defines a single-flavor driver package, otherwise
# this driver package will be multi-flavor. (Note that commenting out
# one of these definitions will not work due to some very strange
# RPM behavior!)
%define flavor %{nil}
%define XXflavor default

%define driver_version 1.1
%define kver %(rpm -q --qf '%{VERSION}-%{RELEASE}' kernel-source)
%define arch %(echo %_target_cpu | sed -e 's/i.86/i386/')

Name:         novell-kmp
License:      GPL
Group:        System/Kernel
Autoreqprov:  on
Summary:      An example module package
%if "%flavor" == ""
Version:      %(echo %driver_version-%kver | tr - _)
Requires:     kernel = %kver
%else
Version:      %(echo %driver_version-%kver-%flavor | tr - _)
Requires:     kernel-%flavor = %kver
%endif
Release:      0
Source0:      novell-kmp-%driver_version.tar.bz2
Source1:      depmod.sh
Source2:      mkinitrd.sh
BuildRoot:    %{_tmppath}/%{name}-%{version}-build

%description
Driver test

%prep
# Make sure to include a %setup statement in the %prep section:
# without, the ``%post -f ...'' and ``%postun -f ...'' statements
# will silently fail and produce empty scripts.
%setup -n novell-kmp-%driver_version
mkdir source
mv * source/ || :
mkdir obj

%build
export EXTRA_CFLAGS='-DVERSION=\"%driver_version\"'
%if "%flavor" == ""
flavors=$(ls /usr/src/linux-obj/%arch)
%else
flavors=%flavor
%endif
```

```
for flavor in $flavors; do
    if [ $flavor = um ]; then
      # User Mode Linux is an exception for many external kernel modules;
      # we may choose to skip it here.
      continue
    fi
    rm -rf obj-$flavor
    cp -r source obj/$flavor
    make -C /usr/src/linux-obj/%arch/$flavor modules M=$PWD/obj/$flavor
done

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

set -- $(ls $RPM_BUILD_ROOT/lib/modules)
KERNELRELEASES=$*

set -- $(find $RPM_BUILD_ROOT/lib/modules -type f -name '*.ko' \
       | sed -e 's:.*/::' -e 's:\.ko$::' | sort -u)
MODULES=$*

(   cat <<-EOF
      # IMPORTANT: Do not change the KERNELRELEASES definition; it will be
      # replaced during driver reuse!
      KERNELRELEASES="$KERNELRELEASES"
      MODULES="$MODULES"
      EOF
    cat %_sourcedir/depmod.sh
    cat %_sourcedir/mkinitrd.sh
) > post_postun.sh

mkdir -p $RPM_BUILD_ROOT/var/lib/YaST2/download
# Insert your download location here:
echo "ftp://ftp.suse.com/pub/suse;SUSE/Novell" \
    > $RPM_BUILD_ROOT/var/lib/YaST2/download/%name

%post -f post_postun.sh

%postun -f post_postun.sh

%files
%defattr(-, root, root)
/lib/modules/*
%dir /var/lib/YaST2
%dir /var/lib/YaST2/download
%config(noreplace) /var/lib/YaST2/download/%name

%changelog
* Thu Dec 01 2005 - agruen@suse.de
- Initial package.
```

Appendix B: depmod.sh

After installing or removing a Kernel Module Package, depmod must be called to update the affected kernel's module dependency cache and map files. Use this script as part of the %post and %postun scripts of kernel module packages. The script is part of the kernel-update-tool package, and can be found in /usr/share/doc/packages/kernel-update-tool/novell-kmp/.

```
# Need to call depmod when the list of modules changes
for kernelrelease in $KERNELRELEASES; do
    if [ -e /boot/System.map-$kernelrelease ]; then
      depmod -a -F /boot/System.map-$kernelrelease $kernelrelease
    fi
done
```

Appendix C: mkinitrd.sh

If a kernel module package installs or removes modules that are part of the initial ramdisk (initrd), the initrd must be recreated. Use this script as part of the %post and %postun scripts of kernel module packages. The script is part of the kernel-update-tool package, and can be found in /usr/share/doc/packages/kernel-update-tool/novell-kmp/.

```
# If one of the modules in this package is in the initrd,
# we need to recreate the initrd.

if [ -e /etc/sysconfig/kernel -a -f /etc/fstab ]; then
    source /etc/sysconfig/kernel
    run_mkinitrd=
    for module in $INITRD_MODULES; do
      case " $MODULES " in
      *" $module "*)
          run_mkinitrd=1
          break ;;
      esac
    done
    if [ -n "$run_mkinitrd" ]; then
      for kernelrelease in $KERNELRELEASES; do
          for image in vmlinuz image vmlinux linux bzImage; do
            if [ -f /boot/$image-$kernelrelease ]; then
                /sbin/mkinitrd -k /boot/$image-$kernelrelease \
                               -i /boot/initrd-$kernelrelease \
                || exit 1
            fi
          done
      done
    fi
fi
```

13 Appendix D: link-kmps

When looking for kernel module packages to download, the kernel update tool looks for package files that do not include a package release number: the release number can be freely assigned, so the kernel update tool has no way to guess it. We recommend putting kernel module packages on the download server(s) with the filenames generated by rpm (which include the release numbers), and using symlinks for the names the kernel-update-tool will use. It is not always trivial to create the correct symlinks for a non-trivial set of kernel module packages. This script is part of the

kernel-update-tool package, and can be can be used to create these symlinks automatically. It can be found in /usr/share/doc/packages/kernel-update-tool/.

```
#! /bin/sh

for package in $(find "$@" -type f -name '*.rpm'); do
    if rpm -qp --provides $package | grep -q -e kernel -e kernel-nongpl; then
      # Looks like a kernel-$flavor package
      continue
    fi
    set -- $(rpm -qp --qf '%{NAME} %{ARCH} %{SOURCERPM}' $package 2> /dev/null)
    [ $? -eq 0 ] || continue
    declare name=$1 arch=$2 sourcerpm=$3

    # don't create links for source rpms: we don't need them.
    [ -z "$sourcerpm" ] && continue

    flavors=0
    for krel in $(rpm -qlp $package \
              | sed -ne 's:^/lib/modules/\([^/]*\)/.*:\1:p' \
              | sort -u); do
      echo "$name-${krel//-/_}.$arch.rpm ${package##*/} $package"
      (( flavors++ ))
    done

    # Check if the package requires kernel = $version-$release or
    # kernel-$flavor = $version-$release as appropriate.
    if [ $flavors != 0 ]; then
      flavor=${krel#*-*-}
      if [ $flavors = 1 ]; then
          req="kernel-$flavor = ${krel%-$flavor}"
      else
          req="kernel = ${krel%-$flavor}"
      fi
      if ! rpm -qp --requires $package | grep -qF "$req"; then
          echo "Package $package does not require \"$req\"" >&2
      fi
    fi
done \
| ${0%/*}/rpm-version-cmp \
| tac \
| awk '{print $3,$1}' \
| uniq -f1 \
| while read package link; do
    echo ln -s ${package##*/} ${package%/*}/$link
    ln -sf ${package##*/} ${package%/*}/$link
done
```

Changes

December 5, 2005

- It is no longer necessary to use the *get_version_number.sh* Autobuild mechanism to insert the correct kernel version into the Version tag in kernel module package spec files. Instead, rpm macros should be used; see Appendix A.
- Kernel Module Packages previously had the provider prefix, driver name, and driver version in the package name. The driver version is now part of the package version instead, which is more in line with how rpm usually is used.

December 6, 2005

- Change the kernel-update-tools script to look for downloadable driver packages with the new kernel's version + release + flavor as the package version. Change link-kmps to generate the appropriate symlinks: one symlink is generated for each kernel version + release + flavor that a Kernel Module Package supports. Adapt this document accordingly.

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.