Kernel Module Packages Manual
for CODE 11

August 13, 2009

1 Overview

Novell/SUSE Linux distributions use the RPM Package Manager for software management. As such, any SUSE/SLES external kernel modules (i.e., kernel modules not included in kernel packages) should be packaged in rpms. These rpms should be built in accordance with specific guidelines to ensure that the resulting Kernel Module Packages (KMPs) can be installed and updated appropriately, in sync with kernel updates.

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.

2 Scope

This version of the Kernel Module Packages Manual applies to the Novell/SUSE CODE 11 code base, which includes openSUSE 11.1 and newer, SLES 11, as well as all products based on SLES 11. Versions of this document for CODE 9 and CODE 10 are available as well [9].

This document's Appendix B highlights several CODE 10 → CODE 11 changes that may be of interest to kernel module packagers. Developers who have experience building Kernel Module Packages for CODE 10 should review Appendix B.

3 Background

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 user-space 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. The checksum 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-pae vs. kernel-xen) 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 load only 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

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

comprise the kernel Application Binary Interface (kABI). When an updated kernel includes kABI 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. Some kernel updates will change the kernel ABI (kABI) by adding or removing exported symbols, or existing symbol checksums may change in a kernel update 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 full discussions of this topic.

Novell/SUSE Linux operating systems include technology to ensure that kernel modules can be reused or updated in sync with kernel updates. To make use of this technology, kernel modules must be packaged into Kernel Module Packages (KMPs) as defined in this document.

4 Kernel Packages

All Novell/SUSE products based on 2.6.x kernels contain a set of kernel packages that 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-pae, kernel-default, kernel-xen, etc.) These are the packages that the kernel modules will be used with. Note: In CODE 11, the binary kernel is provided by two packages: kernel-*flavor*-base and kernel-*flavor*.

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.

5 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/2.6.27.8-1-pae for the kernel-pae-2.6.27.8-1 package). Different kernels have different module directories, and will not usually see each other's modules.

Update modules are modules intended to replace or augment the modules that are provided in the kernel packages. 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.

Note that while modules intended to take precedence over in-kernel modules of the same name should be stored below /lib/modules/*version-release-flavor*/updates/, other add-on modules may be stored below /lib/modules/*version-release-flavor*/extra/.

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 have 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 Module Packages must never install modules into weak-updates/ directories.

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

6 Kernel Module Packages

Novell/SUSE is working closely with the Linux Foundation Driver Backport Workgroup to establish a standard structure for building Kernel Module Packages for all RPM-based distributions. As a first step, the Workgroup is establishing a standard spec file template (http://www.linuxfoundation.org/en/Sample_KMP_spec_file) [10] which can be used regardless of distribution. The information in this section reflects a merging of this standard spec file template with the CODE 10 sample spec file [9].

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 %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:

%kernel_module_package [-f filelist] [-p preamble] [-n name] [-v version] [-r release] [-t template] [-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 are composed of the main package name, followed by a dash, the string “kmp”, followed by another dash and the flavor of the supported kernel. The first component (main package name) can be overridden with a different value by using the **-n** option of the %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 source used during the build. 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 %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 %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 %description tag will be applied to both the main package and the sub-packages.

The %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 -t 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 is 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-%1

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

%1 Flavor of the sub-package (e.g., pae).
%2 Kernel release string without flavor (e.g., 2.6.27.8-1).
%{-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 should be passed with the -x option to the %kernel_module_package macro. The sample Kernel Module Package spec file in Appendix A excludes the debug and trace flavors.

Appendix A contains an example Kernel Module Package spec file as well as the source code referenced by it. When this spec file and its accompanying source is built into an i586 rpm as described in section Building Kernel Module Packages below, the BuildRequires tag in the spec file will pull the module-init-tools, kernel-syms and kernel-source packages into the build root. (Note that the %kernel_module_package_buildreqs macro does not need to explicitly list “kernel-source” since the kernel-syms package has a dependency on the kernel-source package.) Let us assume that the required packages are available in version/release 2.6.27.8-1, and that the debug, default, pae, trace, and xen kernel flavors are available on that platform. RPM would then create the following packages:

novell-hello-kmp-default-1.0_2.6.27.8_1.1-0.i586.rpm
novell-hello-kmp-pae-1.0_2.6.27.8_1.1-0.i586.rpm
novell-hello-kmp-xen-1.0_2.6.27.8_1.1-0.i586.rpm

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

Package	**Requires**	**Provides**	**Modules**
novell-hello-kmp-default	kernel(default:kernel) = 81ee021907bb81cf	novell-hello-kmp = 1.0_2.6.27.8_1.1 ksym(default:exported_function) = e52d5bcf novell-hello-kmp-default = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-default/updates/hello.ko
novell-hello-kmp-pae	kernel(pae:kernel) = f9a733a55e8da41f	novell-hello-kmp = 1.0_2.6.27.8_1.1 ksym(pae:exported_function) = e52d5bcf novell-hello-kmp-pae = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-pae/updates/hello.ko
novell-hello-kmp-xen	kernel(xen:kernel) = 449a592a4f3b46a5	novell-hello-kmp = 1.0_2.6.27.8_1.1 ksym(xen:exported_function) = e52d5bcf novell-hello-kmp-xen = 1.0_2.6.27.8_1.1-0	/lib/modules/2.6.27.8_1.0-xen/updates/hello.ko

7 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 (about 200) 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, assume that the kernel-pae-base package provides the following symbol (among others):

kernel(pae:kernel) = f9a733a55e8da41f

and that the kernel-pae package provides the following symbol (among others):

kernel(pae:drivers_net) = 9e3d2b248b3ec097

A matching Kernel Module Package which uses symbols from these two symbol groups would then require kernel(pae:kernel) and kernel(pae:drivers_net) in the same versions. The Kernel Module Package might provide exported functions as ksym(pae:exported_function) = e52d5bcf or similar.

8 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 using the example package found in Appendix A 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 using 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 breakages.

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 using the build script from the build.rpm package.[2]

2 For Novell/SUSE employees: the build script works similarly to Autobuild's build and mbuild scripts. It is slightly

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.[3] 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 checking 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 kernel-syms packages installed in the build environment. The BuildRequires line in spec files takes care of this: the %kernel_module_package_buildreqs macro specifies the kernel-syms package, which pulls in the kernel-source package due to its dependency on it. Note that 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.

---

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.

3 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.

This can easily lead to very hard to diagnose system malfunctions.

9 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 to produce 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 are to be installed.

10 Deploying Kernel Module Packages

Kernel Module Packages may be distributed on Driver Update Disks, as Add-on Products, or simply as standalone rpms. If a Kernel Module Package's driver is required in order to boot an installation kernel, the Kernel Module Package should be provided on a Driver Update Disk (DUD). Otherwise, we recommend providing Kernel Module Packages as Add-on Products complete with URL(s) for functioning update sites.

11 System Installation and Kernel Module Packages

Initial system installation is carried out by YaST from some installation media (CDs or DVDs, network

locations, etc.) As noted above, support for additional hardware that the installation media do not provide can be added with Driver Update Disks. This is most important to enable hardware needed for booting, such as storage controllers.

Update media (aka Driver Update Disks) provide two kinds of modules: those which the kernel that runs the installation uses, and those which are installed onto the final target system. Both types of modules are provided by including Kernel Module Packages on the update media. In addition, 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 (YaST Add-On Products, YaST Software Management, YaST Online Update, the rpm command, etc.) Note that the Add-on Product format supports the ability to register the system for an update site.

Please note that any drivers 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.

12 Kernel Updates and Kernel Module Packages

After all software repositories 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.

13 Futures

Currently, SUSE Linux kernel packages provide checksums for about 200 symbol groups. Each group corresponds to some portion of exported kernel symbols. As noted in section RPM Provides and Requires above, the kernel packages provide checksums for symbol groups rather actual symbols because, so far, it has been infeasible to dependency check the 7000 or so actual kernel symbols at an rpm level. However, Novell/SUSE is looking at options for how to increase the granularity of the symbol group checksums. The subject is also under discussion at a cross-vendor level in the LF Driver Backport Workgroup [10].

Appendix A: Sample Source for novell-hello Kernel Module Package

The following sample spec file is described in the Kernel Module Packages section above. This spec file updates the CODE 10 sample spec file [9] to use the distro-independent macros being refined on http://www.linuxfoundation.org/en/Sample_KMP_spec_file.

novell-hello.spec

```
# norootforbuild

Name:             novell-hello
BuildRequires:    %kernel_module_package_buildreqs
License:          GPL
Group:            System/Kernel
Summary:          Sample Kernel Module Package
Version:          1.0
Release:          0
Source0:          %name-%version.tar.bz2
BuildRoot:        %{_tmppath}/%{name}-%{version}-build

%kernel_module_package -x debug -x trace

%description
This package contains the hello.ko module.

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

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

%changelog
* Tue Dec 22 2008 - andavis@novell.com
- Updated to reflect CODE 11 changes and LF standard spec file work.
* Sat Jan 28 2006 - agruen@suse.de
- Initial package.
```

The following two files should be compressed to form the novell-hello-1.0.tar.bz2 tarball referenced as Source0 in the novell-hello.spec file above.

novell-hello-1.0/Kbuild

```
obj-m      := hello.o
hello-y    += main.o
```

novell-hello-1.0/main.c

```
/*
 * main.c - A demo kernel module.
 *
 * Copyright (C) 2003, 2004, 2005, 2006
 * Andreas Gruenbacher <agruen@suse.de>, SUSE Labs
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation.
 *
 * A copy of the GNU General Public License can be obtained from
 * http://www.gnu.org/.
 */

#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("Andreas Gruenbacher <agruen@suse.de>");
MODULE_DESCRIPTION("Hello world module");
MODULE_LICENSE("GPL");

int param;

module_param(param, int, 0);
MODULE_PARM_DESC(param, "Example parameter");

void exported_function(void)
{
        printk(KERN_INFO "Exported function called.\n");
}
EXPORT_SYMBOL_GPL(exported_function);

int __init init_hello(void)
{
        printk(KERN_INFO "Hello world.\n");
        return 0;
}

void __exit exit_hello(void)
{
        printk(KERN_INFO "Goodbye world.\n");
}

module_init(init_hello);
module_exit(exit_hello);
```

Appendix B – Code 10 → Code 11 Notes for Kernel Module Packagers

Following is a brief list of CODE 10->CODE 11 changes which may affect kernel module packagers:

Kernel packages:

The CODE 11 kernel is provided via two packages: kernel-*flavor*-base and kernel-*flavor*. Kernel-*flavor* is dependent on kernel-*flavor*-base. Both packages are required to provide checksums for all the kernel symbol groups.

CODE 11 automatically installs the -pae kernel flavor 32-bit pae-enabled systems.

The CODE 11 kernel-source package does not include the sample Kernel Module Package source provided in Appendix A.

Package and file locations:

In CODE 11, a number of packages have been moved from the base SLES product to the SLES SDK. Developers building Kernel Module Packages on CODE 11 will need to include the SDK in their build environment in order to ensure that all build-time package dependencies are resolved.

In CODE 11, the rpm macros used to build Kernel Module Packages are provided in the kernel-source package (instead of the rpm package) and installed under /etc/rpm as well as /usr/lib/rpm.

rpm macro changes:

CODE 11 includes rpm macros to facilitate creating Kernel Module Package build structures that support multiple RPM-based distributions. The spec file template in Appendix A includes these macros:

%kernel_module_package_buildreqs - used for “BuildRequires”.
%kernel_module_package - used instead of %suse_kernel_module_package (note that the options are slightly different: with %kernel_module_package, “-x” is used to *exclude* flavors from the build, and “-t” replaces “-s” as the option to override the default sub-package template. See section Kernel Module Packages above for a complete description of all %kernel_module_package options.)
%kernel_source – used with the “$flavor” argument to specify the location of the top-level kernel Makefile.

Changes

December 22, 2008

Initial Version: Update the CODE 10 Kernel Module Packages Manual [9], and adapt it to the CODE 11 process.

August 13, 2009

Update to correctly reflect behavior of -f option to kernel-module-package macro.

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 Gruenbacher: 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.

[9] Andreas Gruenbacher: Kernel Module Packages Manual for CODE 10, http://www.suse.de/~agruen/KMPM/KernelModulePackagesManual-CODE10.pdf.

[10] Documentation of The Linux Foundation Driver Backport Workgroup, http://www.linuxfoundation.org/en/Driver_Backport.