*Reprinted from the*

# Proceedings of the Linux Symposium

Volume One

July 21th–24th, 2004
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

## Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

# Dynamic Kernel Module Support: From Theory to Practice

*Matt Domsch & Gary Lerhaupt*
Dell Linux Engineering
Matt_Domsch@dell.com, Gary_Lerhaupt@dell.com

## Abstract

DKMS is a framework which allows individual kernel modules to be upgraded without changing your whole kernel. Its primary audience is fourfold: system administrators who want to update a single device driver rather than wait for a new kernel from elsewhere with it included; distribution maintainers, who want to release a single targeted bugfix in between larger scheduled updates; system manufacturers who need single modules changed to support new hardware or to fix bugs, but do not wish to test whole new kernels; and driver developers, who must provide updated device drivers for testing and general use on a wide variety of kernels, as well as submit drivers to kernel.org.

Since OLS2003, DKMS has gone from a good idea to deployed and used. Based on end user feedback, additional features have been added: precompiled module tarball support to speed factory installation; driver disks for Red Hat distributions; 2.6 kernel support; SuSE kernel support. Planned features include cross-architecture build support and additional distribution driver disk methods.

In addition to overviewing DKMS and its features, we explain how to create a dkms.conf file to DKMS-ify your kernel module source.

## 1 History

Historically, Linux distributions bundle device drivers into essentially one large kernel package, for several primary reasons:

- Completeness: The Linux kernel as distributed on kernel.org includes all the device drivers packaged neatly together in the same kernel tarball. Distro kernels follow kernel.org in this respect.
- Maintainer simplicity: With over 4000 files in the kernel `drivers/` directory, each possibly separately versioned, it would be impractical for the kernel maintainer(s) to provide a separate package for each driver.
- Quality Assurance / Support organization simplicity: It is easiest to ask a user "what kernel version are you running," and to compare this against the list of approved kernel versions released by the QA team, rather than requiring the customer to provide a long and extensive list of package versions, possibly one per module.
- End user install experience: End users don't care about which of the 4000 possible drivers they need to install, they just want it to work.

This works well as long as you are able to make the "top of the tree" contain the most current

and most stable device driver, and you are able to convince your end users to always run the "top of the tree." The `kernel.org` development processes tend to follow this model with great success.

But widely used distros cannot ask their users to always update to the top of the kernel.org tree. Instead, they start their products from the top of the kernel.org tree at some point in time, essentially freezing with that, to begin their test cycles. The duration of these test cycles can be as short as a few weeks, and as long as a few years, but 3-6 months is not unusual. During this time, the kernel.org kernels march forward, and some (but not all) of these changes are backported into the distro's kernel. They then apply the minimal patches necessary for them to declare the product finished, and move the project into the sustaining phase, where changes are very closely scrutinized before releasing them to the end users.

### 1.1 Backporting

It is this sustaining phase that DKMS targets. DKMS can be used to backport newer device driver versions from the "top of the tree" kernels where most development takes place to the now-historical kernels of released products.

The `PATCH_MATCH` mechanism was specifically designed to allow the application of patches to a "top of the tree" device driver to make it work with older kernels. This allows driver developers to continue to focus their efforts on keeping kernel.org up to date, while allowing that same effort to be used on existing products with minimal changes. See Section 6 for a further explanation of this feature.

### 1.2 Driver developers' packaging

Driver developers have recognized for a long time that they needed to provide backported versions of their drivers to match their end users' needs. Often these requirements are imposed on them by system vendors such as Dell in support of a given distro release. However, each driver developer was free to provide the backport mechanism in any way they chose. Some provided architecture-specific RPMs which contained only precompiled modules. Some provided source RPMs which could be rebuilt for the running kernel. Some provided driver disks with precompiled modules. Some provided just source code patches, and expected the end user to rebuild the kernel themselves to obtain the desired device driver version. All provided their own Makefiles rather than use the kernel-provided build system.

As a result, different problems were encountered with each developers' solution. Some developers had not included their drivers in the kernel.org tree for so long that that there were discrepancies, e.g. `CONFIG_SMP` vs `__SMP__`, `CONFIG_2G` vs. `CONFIG_3G`, and compiler option differences which went unnoticed and resulted in hard-to-debug issues.

Needless to say, with so many different mechanisms, all done differently, and all with different problems, it was a nightmare for end users.

A new mechanism was needed to cleanly handle applying updated device drivers onto an end user's system. Hence DKMS was created as the one module update mechanism to replace all previous methods.

## 2 Goals

DKMS has several design goals.

- Implement only mechanism, not policy.
- Allow system administrators to easily know what modules, what versions, for

what kernels, and in what state, they have on their system.

- Keep module source as it would be found in the "top of the tree" on kernel.org. Apply patches to backport the modules to earlier kernels as necessary.

- Use the kernel-provided build mechanism. This reduces the Makefile magic that driver developers need to know, thus the likelihood of getting it wrong.

- Keep additional DKMS knowledge a driver developer must have to a minimum. Only a small per-driver dkms.conf file is needed.

- Allow multiple versions of any one module to be present on the system, with only one active at any given time.

- Allow DKMS-aware drivers to be packaged in the Linux Standard Base-conformant RPM format.

- Ease of use by multiple audiences: driver developers, system administrators, Linux distros, and system vendors.

We discuss DKMS as it applies to each of these four audiences.

## 3 Distributions

All present Linux distributions distribute device drivers bundled into essentially one large kernel package, for reasons outlined in Section 1. It makes the most sense, most of the time.

However, there are cases where it does not make sense.

- Severity 1 bugs are discovered in a single device driver between larger scheduled updates. Ideally you'd like your affected users to be able to get the single module update without having to release and Q/A a whole new kernel. Only customers who are affected by the particular bug need to update "off-cycle."

- Solutions vendors, for change control reasons, often certify their solution on a particular distribution, scheduled update release, and sometimes specific kernel version. The latter, combined with releasing device driver bug fixes as whole new kernels, puts the customer in the untenable position of either updating to the new kernel (and losing the certification of the solution vendor), or forgoing the bug fix and possibly putting their data at risk.

- Some device drivers are not (yet) included in kernel.org nor a distro kernel, however one may be required for a functional software solution. The current support models require that the add-on driver "taint" the kernel or in some way flag to the support organization that the user is running an unsupported kernel module. Tainting, while valid, only has three dimensions to it at present: Proprietary—non-GPL licensed; Forced—loaded via `insmod -f`; and Unsafe SMP—for some CPUs which are not designed to be SMP-capable. A GPL-licensed device driver which is not yet in kernel.org or provided by the distribution may trigger none of these taints, yet the support organization needs to be aware of this module's presence. To avoid this, we expect to see the distros begin to cryptographically sign kernel modules that they produce, and taint on load of an unsigned module. This would help reduce the support organization's work for calls about "unsupported"

configurations. With DKMS in use, there is less a need for such methods, as it's easy to see which modules have been changed.

*Note: this is not to suggest that driver authors should not submit their drivers to* `kernel.org`*—absolutely they should.*

- The distro QA team would like to test updates to specific drivers without waiting for the kernel maintenance team to rebuild the kernel package (which can take many hours in some cases). Likewise, individual end users may be willing (and often be required, e.g. if the distro QA team can't reproduce the users's hardware and software environment exactly) to show that a particular bug is fixed in a driver, prior to releasing the fix to *all* of that distro's users.

- New hardware support via driver disks: Hardware vendors release new hardware asynchronously to any software vendor schedule, no matter how hard companies may try to synchronize releases. OS distributions provide install methods which use driver diskettes to enable new hardware for previously-released versions of the OS. Generating driver disks has always been a difficult and error-prone procedure, different for each OS distribution, not something that the casual end-user would dare attempt.

DKMS was designed to address all of these concerns.

DKMS aims to provide a clear separation between mechanism (how one updates individual kernel modules and tracks such activity) and policy (when should one update individual kernel modules).

### 3.1 Mechanism

DKMS provides only the mechanism for updating individual kernel modules, not policy. As such, it can be used by distributions (per their policy) for updating individual device drivers for individual users affected by Severity 1 bugs, without releasing a whole new kernel.

The first mechanism critical to a system administrator or support organization is the `status` command, which reports the name, version, and state of each kernel module under DKMS control. By querying DKMS for this information, system administrators and distribution support organizations may quickly understand when an updated device driver is in use to speed resolution when issues are seen.

DKMS's ability to generate driver diskettes gives control to both novice and seasoned system administrators alike, as they can now perform work which otherwise they would have to wait for a support organization to do for them. They can get their new hardware systems up-and-running quickly by themselves, leaving the support organizations with time to do other more interesting value-added work.

### 3.2 Policy

Suggested policy items include:

- Updates must pass QA. This seems obvious, but it reduces broken updates (designed to fix other problems) from being released.

- Updates must be submitted, and ideally be included already, upstream. For this we expect kernel.org and the OS distribution to include the update in their next larger scheduled update. This ensures that when the next kernel.org kernel or distro update

comes out, the short-term fix provided via DKMS is incorporated already.

- The `AUTOINSTALL` mechanism is set to `NO` for all modules which are shipped with the target distro's kernel. This prevents the DKMS autoinstaller from installing a (possibly older) kernel module onto a newer kernel without being explicitly told to do so by the system administrator. This follows from the "all DKMS updates must be in the next larger release" rule above.

- All issues for which DKMS is used are tracked in the appropriate bug tracking databases until they are included upstream, and are reviewed regularly.

- All DKMS packages are provided as DKMS-enabled RPMs for easy installation and removal, per the Linux Standard Base specification.

- All DKMS packages are posted to the distro's support web site for download by system administrators affected by the partiular issue.

## 4 System Vendors

DKMS is useful to System Vendors such as Dell for many of the same reasons it's useful to the Linux distributions. In addition, system vendors face additional issues:

- Critical bug fixes for distro-provided drivers: While we hope to never need such, and we test extensively with distro-provided drivers, occasionally we have discovered a critical bug after the distribution has cut their gold CDs. We use DKMS to update just the affected device drivers.

- Alternate drivers: Dell occasionally needs to provide an alternate driver for a piece of hardware rather than that provided by the distribution natively. For example, Dell provides the Intel iANS network channel bonding and failover driver for customers who have used iANS in the past, and wish to continue using it rather than upgrading to the native channel bonding driver resident in the distribution.

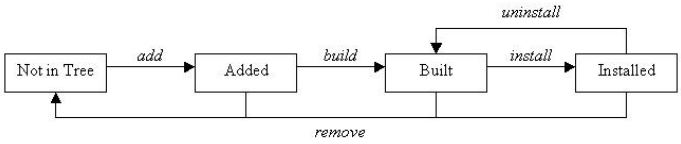
- Factory installation: Dell installs various OS distribution releases onto new hardware in its factories. We try not to update from the gold release of a distribution version to any of the scheduled updates, as customers expect to receive gold. We use DKMS to enable newer device drivers to handle newer hardware than was supported natively in the gold release, while keeping the gold kernel the same.

We briefly describe the policy Dell uses, in addition to the above rules suggested to OS distributions:

- Prebuilt DKMS tarballs are required for factory installation use, for all kernels used in the factory install process. This prevents the need for the compiler to be run, saving time through the factories. Dell rarely changes the factory install images for a given OS release, so this is not a huge burden on the DKMS packager.

- All DKMS packages are posted to support.dell.com for download by system administrators purchasing systems without Linux factory-installed.

Figure 1: DKMS state diagram.




# 5 System Administrators

## 5.1 Understanding the DKMS Life Cycle

Before diving into using DKMS to manage kernel modules, it is helpful to understand the life cycle by which DKMS maintains your kernel modules. In Figure 1, each rectangle represents a state your module can be in and each italicized word represents a DKMS action that can used to switch between the various DKMS states. In the following section we will look further into each of these DKMS actions and then continue on to discuss auxiliary DKMS functionality that extends and improves upon your ability to utilize these basic commands.

## 5.2 RPM and DKMS

DKMS was designed to work well with Red Hat Package Manger (RPM). Many times using DKMS to install a kernel module is as easy as installing a DKMS-enabled module RPM. Internally in these RPMs, DKMS is used to `add`, `build`, and `install` a module. By wrapping DKMS commands inside of an RPM, you get the benefits of RPM (package versioning, security, dependency resolution, and package distribution methodologies) while DKMS handles the work RPM does not, versioning and building of individual kernel modules. For reference, a sample DKMS-enabled RPM specfile can be found in the DKMS package.

## 5.3 Using DKMS

### 5.3.1 Add

DKMS manages kernel module versions at the source code level. The first requirement of using DKMS is that the module source be located on the build system and that it be located in the directory `/usr/src/<module>-<module-version>/`. It also requires that a dkms.conf file exists with the appropriately formatted directives within this configuration file to tell DKMS such things as where to install the module and how to build it. Once these two requirements have been met and DKMS has been installed on your system, you can begin using DKMS by adding a module/module-version to the DKMS tree. For example:

```
# dkms add -m megaraid2 -v 2.10.3
```

This example `add` command would add megaraid2/2.10.3 to the already existent `/var/dkms` tree, leaving it in the Added state.

### 5.3.2 Build

Once in the Added state, the module is ready to be built. This occurs through the DKMS `build` command and requires that the proper kernel sources are located on the system from the `/lib/module/<kernel-version>/build` symlink. The make command that is

used to compile the module is specified in the dkms.conf configuration file. Continuing with the megaraid2/2.10.3 example:

```
# dkms build -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

The `build` command compiles the module but stops short of installing it. As can be seen in the above example, `build` expects a kernel-version parameter. If this kernel name is left out, it assumes the currently running kernel. However, it functions perfectly well to build modules for kernels that are not currently running. This functionality is assured through use of a kernel preparation subroutine that runs before any module build is performed in order to ensure that the module being built is linked against the proper kernel symbols.

Successful completion of a `build` creates, for this example, the `/var/dkms/megaraid2/2.10.3/2.4.21-4.ELsmp/` directory as well as the log and module subdirectories within this directory. The log directory holds a log file of the module make and the module directory holds copies of the resultant binaries.

### 5.3.3 Install

With the completion of a `build`, the module can now be installed on the kernel for which it was built. Installation copies the compiled module binary to the correct location in the `/lib/modules/` tree as specified in the dkms.conf file. If a module by that name is already found in that location, DKMS saves it in its tree as an original module so at a later time it can be put back into place if the newer module is uninstalled. An example `install` command:

```
# dkms install -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

If a module by the same name is already installed, DKMS saves a copy in its tree and does so in the `/var/dkms/<module-name>/original_module/` directory. In this case, it would be saved to `/var/dkms/megaraid2/original_module/2.4.21-4.ELsmp/`.

### 5.3.4 Uninstall and Remove

To complete the DKMS cycle, you can also uninstall or remove your module from the tree. The `uninstall` command deletes from `/lib/modules` the module you installed and, if applicable, replaces it with its original module. In scenarios where multiple versions of a module are located within the DKMS tree, when one version is uninstalled, DKMS does not try to understand or assume which of these other versions to put in its place. Instead, if a true "original_module" was saved from the very first DKMS installation, it will be put back into the kernel and all of the other module versions for that module will be left in the Built state. An example `uninstall` would be:

```
# dkms uninstall -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

Again, if the kernel version parameter is unset, the currently running kernel is assumed, although, the same behavior does not occur with the `remove` command. The `remove` and `uninstall` are very similar in that `remove` will do all of the same steps as `uninstall`. However, when `remove` is employed, if the module-version being removed is the last instance of that module-version for all kernels on your system, after the uninstall portion of the remove completes, it will delete all traces of that module from the DKMS tree. To put it another way, when an `uninstall` command completes, your modules are left in the Built

state. However, when a `remove` completes, you would be left in the Not in Tree state. Here are two sample `remove` commands:

```
# dkms remove -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
# dkms remove -m megaraid2
   -v 2.10.3 --all
```

With the first example `remove` command, your module would be uninstalled and if this module/module-version were not installed on any other kernel, all traces of it would be removed from the DKMS tree all together. If, say, megaraid2/2.10.3 was also installed on the 2.4.21-4.ELhugemem kernel, the first `remove` command would leave it alone and it would remain intact in the DKMS tree. In the second example, that would not be the case. It would uninstall all versions of the megaraid2/2.10.3 module from all kernels and then completely expunge all references of megaraid2/2.10.3 from the DKMS tree. Thus, `remove` is what cleans your DKMS tree.

### 5.4 Miscellaneous DKMS Commands

#### 5.4.1 Status

DKMS also comes with a fully functional status command that returns information about what is currently located in your tree. If no parameters are set, it will return all information found. Logically, the specificity of information returned depends on which parameters are passed to your status command. Each status entry returned will be of the state: "added," "built," or "installed," and if an original module has been saved, this information will also be displayed. Some example status commands include:

```
# dkms status
# dkms status -m megaraid2
# dkms status -m megaraid2 -v 2.10.3
# dkms status -k 2.4.21-4.ELsmp
# dkms status -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

#### 5.4.2 Match

Another major feature of DKMS is the match command. The match command takes the configuration of DKMS installed modules for one kernel and applies this same configuration to some other kernel. When the match completes, the same module/module-versions that were installed for one kernel are also then installed on the other kernel. This is helpful when you are upgrading from one kernel to the next, but would like to keep the same DKMS modules in place for the new kernel. Here is an example:

```
# dkms match
   --templatekernel 2.4.21-4.ELsmp
   -k 2.4.21-5.ELsmp
```

As can be seen in the example, the `--templatekernel` is the "match-er" kernel from which the configuration is based, while the `-k kernel` is the "match-ee" upon which the configuration is instated.

#### 5.4.3 dkms_autoinstaller

Similar in nature to the match command is the dkms_autoinstaller service. This service gets installed as part of the DKMS RPM in the /etc/init.d directory. Depending on whether an `AUTOINSTALL` directive is set within a module's dkms.conf configuration file, the dkms_autoinstaller will automatically build and install that module as you boot your system into new kernels which do not already have this module installed.

#### 5.4.4 mkdriverdisk

The last miscellaneous DKMS command is `mkdriverdisk`. As can be inferred from its name, `mkdriverdisk` will take the proper

sources in your DKMS tree and create a driver disk image for use in providing updated drivers to Linux distribution installations. A sample `mkdriverdisk` might look like:

```
# dkms mkdriverdisk -d redhat
   -m megaraid2 -v 2.10.3
   -k 2.4.21-4.ELBOOT
```

Currently, the only supported distribution driver disk format is Red Hat. For more information on the extra necessary files and their formats for DKMS to create Red Hat driver disks, see `http://people.redhat.com/dledford`. These files should be placed in your module source directory.

### 5.5 Systems Management with DKMS Tarballs

As we have seen, DKMS provides a simple mechanism to build, install, and track device driver updates. So far, all these actions have related to a single machine. But what if you've got many similar machines under your administrative control? What if you have a compiler and kernel source on only one system (your master build system), but you need to deploy your newly built driver to all your other systems? DKMS provides a solution to this as well—in the `mktarball` and `ldtarball` commands.

The `mktarball` command rolls up copies of each device driver module file which you've built using DKMS into a compressed tarball. You may then copy this tarball to each of your target systems, and use the DKMS `ldtarball` command to load those into your DKMS tree, leaving each module in the Built state, ready to be installed. This avoids the need for both kernel source and compilers to be on every target system.

For example:

You have built the megaraid2 device driver, version 2.10.3, for two different kernel families (here 2.4.20-9 and 2.4.21-4.EL), on your master build system.

```
# dkms status
megaraid2, 2.10.3, 2.4.20-9: built
megaraid2, 2.10.3, 2.4.20-9bigmem: built
megaraid2, 2.10.3, 2.4.20-9BOOT: built
megaraid2, 2.10.3, 2.4.20-9smp: built
megaraid2, 2.10.3, 2.4.21-4.EL: built
megaraid2, 2.10.3, 2.4.21-4.ELBOOT: built
megaraid2, 2.10.3, 2.4.21-4.ELhugemem: built
megaraid2, 2.10.3, 2.4.21-4.ELsmp: built
```

You wish to deploy this version of the driver to several systems, without rebuilding from source each time. You can use the `mktarball` command to generate one tarball for each kernel family:

```
# dkms mktarball -m megaraid2
   -v 2.10.3
   -k 2.4.21-4.EL,2.4.21-4.ELsmp,
   2.4.21-4.ELBOOT,2.4.21-4.ELhugemem

Marking /usr/src/megaraid2-2.10.3 for archiving...
Marking kernel 2.4.21-4.EL for archiving...
Marking kernel 2.4.21-4.ELBOOT for archiving...
Marking kernel 2.4.21-4.ELhugemem for archiving...
Marking kernel 2.4.21-4.ELsmp for archiving...
Tarball location:
  /var/dkms/megaraid2/2.10.3/tarball/
megaraid2-2.10.3-manykernels.tgz
Done.
```

You can make one big tarball containing modules for both families by omitting the -k argument and kernel list; DKMS will include a module for every kernel version found.

You may then copy the tarball (renaming it if you wish) to each of your target systems using any mechanism you wish, and load the modules in. First, see that the target DKMS tree does not contain the modules you're loading:

```
# dkms status
Nothing found within the DKMS tree for
this status command. If your modules were
not installed with DKMS, they will not show
up here.
```

Then, load the tarball on your target system:

```
# dkms ldtarball
   --archive=megaraid2-2.10.3-manykernels.tgz

Loading tarball for module:
 megaraid2 / version: 2.10.3
Loading /usr/src/megaraid2-2.10.3...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.EL...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELBOOT...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELhugemem...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELsmp...
Creating /var/dkms/megaraid2/2.10.3/source symlink...
```

Finally, verify the modules are present, and in the Built state:

```
# dkms status
megaraid2, 2.10.3, 2.4.21-4.EL: built
megaraid2, 2.10.3, 2.4.21-4.ELBOOT: built
megaraid2, 2.10.3, 2.4.21-4.ELhugemem: built
megaraid2, 2.10.3, 2.4.21-4.ELsmp: built
```

DKMS `ldtarball` leaves the modules in the Built state, not the Installed state. For each kernel version you want your modules to be installed into, follow the install steps as above.

# 6 Driver Developers

As the maintainer of a kernel module, the only thing you need to do to get DKMS interoperability is place a small dkms.conf file in your driver source tarball. Once this has been done, any user of DKMS can simply do:

```
dkms ldtarball --archive /path/to/foo-1.0.tgz
```

That's it. We could discuss at length (which we will not rehash in this paper) the best methods to utilizing DKMS within a dkms-enabled module RPM, but for simple DKMS usability, the buck stops here. With the dkms.conf file in place, you have now positioned your source tarball to be usable by all manner and skill level of Linux users utilizing your driver. Effectively, you have widely increased your testing base without having to wade into package management or pre-compiled binaries. DKMS will handle this all for you. Along the same line, by leveraging DKMS you can now easily allow more widespread testing of your driver. Since driver versions can now be cleanly tracked outside of the kernel tree, you no longer must wait for the next kernel release in order for the community to register the necessary debugging cycles against your code. Instead, DKMS can be counted on to manage various versions of your kernel module such that any catastrophic errors in your code can be easily mitigated by a singular dkms uninstall command.

This leaves the composition of the dkms.conf as the only interesting piece left to discuss for the driver developer audience. With that in mind, we will now explicate over two dkms.conf examples ranging from that which is minimally required (Figure 2) to that which expresses maximal configuration (Figure 3).

### 6.1 Minimal dkms.conf for 2.4 kernels

Referring to Figure 2, the first thing that is distinguishable is the definition of the version of the package and the make command to be used to compile your module. This is only necessary for 2.4-based kernels, and lets the developer specify their desired make incantation.

Reviewing the rest of the dkms.conf, `PACKAGE_NAME` and `BUILT_MODULE_NAME[0]` appear to be duplicate in nature, but this is only the case for a package which contains only one kernel module within it. Had this example been for something like ALSA, the name of the package would be "alsa," but the `BUILT_MODULE_NAME` array would instead be populated with the names of the kernel modules within the ALSA package.

The final required piece of this minimal example is the `DEST_MODULE_LOCATION` array. This simply tells DKMS where in the /lib/modules tree it should install your module.

```
PACKAGE_NAME="megaraid2"
PACKAGE_VERSION="2.10.3"

MAKE[0]="make -C ${kernel_source_dir}
  SUBDIRS=${dkms_tree}/${PACKAGE_NAME}/${PACKAGE_VERSION}/build modules"

BUILT_MODULE_NAME[0]="megaraid2"
DEST_MODULE_LOCATION[0]="/kernel/drivers/scsi/"
```

Figure 2: A minimal dkms.conf

### 6.2 Minimal dkms.conf for 2.6 kernels

In the current version of DKMS, for 2.6 kernels the MAKE command listed in the dkms.conf is wholly ignored, and instead DKMS will always use:

```
make -C /lib/modules/$kernel_version/build \
  M=$dkms_tree/$module/$module_version/build
```

This jibes with the new external module build infrastructure supported by Sam Ravnborg's kernel Makefile improvements, as DKMS will always build your module in a build subdirectory it creates for each version you have installed. Similarly, an impending future version of DKMS will also begin to ignore the `PACKAGE_VERSION` as specified in dkms.conf in favor of the new modinfo provided information as implemented by Rusty Russell.

With regard to removing the requirement for `DEST_MODULE_LOCATION` for 2.6 kernels, given that similar information should be located in the install target of the Makefile provided with your package, it is theoretically possible that DKMS could one day glean such information from the Makefile instead. In fact, in a simple scenario as this example, it is further theoretically possible that the name of the package and of the built module could also be determined from the package Makefile. In effect, this would completely remove any need for a dkms.conf whatsoever, thus enabling all simple module tarballs to be automatically DKMS enabled.

Though, as these features have not been explored and as package maintainers would likely want to use some of the other dkms.conf directive features which are about to be elaborated upon, it is likely that requiring a dkms.conf will continue for the foreseeable future.

### 6.3 Optional dkms.conf directives

In the real-world version of the Dell's DKMS-enabled megaraid2 package, we also specify the optional directives:

```
MODULES_CONF_ALIAS_TYPE[0]=
   "scsi_hostadapter"
MODULES_CONF_OBSOLETES[0]=
   "megaraid,megaraid_2002"
REMAKE_INITRD="yes"
```

These directives tell DKMS to remake the kernel's initial ramdisk after every DKMS install or uninstall of this module. They further specify that before this happens, /etc/modules.conf (or /etc/sysconfig/kernel) should be edited intelligently so that the initrd is properly assembled. In this case, if /etc/modules.conf already contains a reference to either "megaraid" or "megaraid_2002," these will be switched to "megaraid2." If no such references are found,

then a new "scsi_hostadapter" entry will be added as the last such scsi_hostadapter number.

On the other hand, if it had also included:

```
MODULES_CONF_OBSOLETES_ONLY="yes"
```

then had no obsolete references been found, a new "scsi_hostadapter" line would not have been added. This would be useful in scenarios where you instead want to rely on something like Red Hat's kudzu program for adding references for your kernel modules.

As well one could hypothetically also specify within the dkms.conf:

```
DEST_MODULE_NAME[0]="megaraid"
```

This would cause the resultant megaraid2 kernel module to be renamed to "megaraid" before being installed. Rather than having to propagate various one-off naming mechanisms which include the version as part of the module name in /lib/modules as has been previous common practice, DKMS could instead be relied upon to manage all module versioning to avoid such clutter. Was megaraid_2002 a version or just a special year in the hearts of the megaraid developers? While you and I might know the answer to this, it certainly confused Dell's customers.

Continuing with hypothetical additions to the dkms.conf in Figure 2, one could also include:

```
BUILD_EXCLUSIVE_KERNEL="^2\.4.*"
BUILD_EXCLUSIVE_ARCH="i.86"
```

In the event that you know the code you produced is not portable, this is how you can tell DKMS to keep people from trying to build it elsewhere. The above restrictions would only allow the kernel module to be built on 2.4 kernels on x86 architectures.

Continuting with optional dkms.conf directives, the ALSA example in Figure 3 is taken directly from a DKMS-enabled package that Dell released to address sound issues on the Precision 360 workstation. It is slightly abridged as the alsa-driver as delivered actually installs 13 separate kernel modules, but for the sake of this example, only 9 are shown.

In this example, we have:

```
AUTOINSTALL="yes"
```

This tells the boot-time service dkms_autoinstaller that this package should be built and installed as you boot into a new kernel that DKMS has not already installed this package upon. By general policy, Dell only allows `AUTOINSTALL` to be set if the kernel modules are not already natively included with the kernel. This is to avoid the scenario where DKMS might automatically install over a newer version of the kernel module as provided by some newer version of the kernel. However, given the 2.6 modinfo changes, DKMS can now be modified to intelligently check the version of a native kernel module before clobbering it with some older version. This will likely result in a future policy change within Dell with regard to this feature.

In this example, we also have:

```
PATCH[0]="adriver.h.patch"
PATCH_MATCH[0]="2.4.[2-9][2-9]"
```

These two directives indicate to DKMS that if the kernel that the kernel module is being built for is >=2.4.22 (but still of the 2.4 family), the included adriver.h.patch should first be

```
PACKAGE_NAME="alsa-driver"
PACKAGE_VERSION="0.9.0rc6"

MAKE="sh configure --with-cards=intel8x0 --with-sequencer=yes \
  --with-kernel=/lib/modules/$kernelver/build \
  --with-moddir=/lib/modules/$kernelver/kernel/sound > /dev/null; make"
AUTOINSTALL="yes"

PATCH[0]="adriver.h.patch"
PATCH_MATCH[0]="2.4.[2-9][2-9]"

POST_INSTALL="alsa-driver-dkms-post.sh"
MODULES_CONF[0]="alias char-major-116 snd"
MODULES_CONF[1]="alias snd-card-0 snd-intel8x0"
MODULES_CONF[2]="alias char-major-14 soundcore"
MODULES_CONF[3]="alias sound-slot-0 snd-card-0"
MODULES_CONF[4]="alias sound-service-0-0 snd-mixer-oss"
MODULES_CONF[5]="alias sound-service-0-1 snd-seq-oss"
MODULES_CONF[6]="alias sound-service-0-3 snd-pcm-oss"
MODULES_CONF[7]="alias sound-service-0-8 snd-seq-oss"
MODULES_CONF[8]="alias sound-service-0-12 snd-pcm-oss"
MODULES_CONF[9]="post-install snd-card-0 /usr/sbin/alsactl restore >/dev/null 2>&1 || :"
MODULES_CONF[10]="pre-remove snd-card-0 /usr/sbin/alsactl store >/dev/null 2>&1 || :"

BUILT_MODULE_NAME[0]="snd-pcm"
BUILT_MODULE_LOCATION[0]="acore"
DEST_MODULE_LOCATION[0]="/kernel/sound/acore"

BUILT_MODULE_NAME[1]="snd-rawmidi"
BUILT_MODULE_LOCATION[1]="acore"
DEST_MODULE_LOCATION[1]="/kernel/sound/acore"

BUILT_MODULE_NAME[2]="snd-timer"
BUILT_MODULE_LOCATION[2]="acore"
DEST_MODULE_LOCATION[2]="/kernel/sound/acore"

BUILT_MODULE_NAME[3]="snd"
BUILT_MODULE_LOCATION[3]="acore"
DEST_MODULE_LOCATION[3]="/kernel/sound/acore"

BUILT_MODULE_NAME[4]="snd-mixer-oss"
BUILT_MODULE_LOCATION[4]="acore/oss"
DEST_MODULE_LOCATION[4]="/kernel/sound/acore/oss"

BUILT_MODULE_NAME[5]="snd-pcm-oss"
BUILT_MODULE_LOCATION[5]="acore/oss"
DEST_MODULE_LOCATION[5]="/kernel/sound/acore/oss"

BUILT_MODULE_NAME[6]="snd-seq-device"
BUILT_MODULE_LOCATION[6]="acore/seq"
DEST_MODULE_LOCATION[6]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[7]="snd-seq-midi-event"
BUILT_MODULE_LOCATION[7]="acore/seq"
DEST_MODULE_LOCATION[7]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[8]="snd-seq-midi"
BUILT_MODULE_LOCATION[8]="acore/seq"
DEST_MODULE_LOCATION[8]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[9]="snd-seq"
BUILT_MODULE_LOCATION[9]="acore/seq"
DEST_MODULE_LOCATION[9]="/kernel/sound/acore/seq"
```

Figure 3: An elaborate dkms.conf

applied to the module source before a module build occurs. In this way, by including various patches needed for various kernel versions, you can distribute one source tarball and ensure it will always properly build regardless of the end user target kernel. If no corresponding `PATCH_MATCH[0]` entry were specified for `PATCH[0]`, then the adriver.h.patch would always get applied before a module build. As DKMS always starts off each module build with pristine module source, you can always ensure the right patches are being applied.

Also seen in this example is:

```
MODULES_CONF[0]=
  "alias char-major-116 snd"
MODULES_CONF[1]=
  "alias snd-card-0 snd-intel8x0"
```

Unlike the previous discussion of /etc/modules.conf changes, any entries placed into the `MODULES_CONF` array are automatically added into /etc/modules.conf during a module install. These are later only removed during the final module uninstall.

Lastly, we have:

```
POST_INSTALL="alsa-driver-dkms-post.sh"
```

In the event that you have other scripts that must be run during various DKMS events, DKMS includes `POST_ADD`, `POST_BUILD`, `POST_INSTALL` and `POST_REMOVE` functionality.

# 7 Future

As you can tell from the above, DKMS is very much ready for deployment now. However, as with all software projects, there's room for improvement.

### 7.1 Cross-Architecture Builds

DKMS today has no concept of a platform architecture such as i386, x86_64, ia64, sparc, and the like. It expects that it is building kernel modules with a native compiler, not a cross compiler, and that the target architecture is the native architecture. While this works in practice, it would be convenient if DKMS were able to be used to build kernel modules for non-native architectures.

Today DKMS handles the cross-architecture build process by having separate /var/dkms directory trees for each architecture, and using the `dkmstree` option to specify a using a different tree, and the `config` option to specify to use a different kernel configuration file.

Going forward, we plan to add an `--arch` option to DKMS, or have it glean it from the kernel config file and act accordingly.

### 7.2 Additional distribution driver disks

DKMS today supports generating driver disks in the Red Hat format only. We recognize that other distributions accomplish the same goal using other driver disk formats. This should be relatively simple to add once we understand what the additional formats are.

# 8 Conclusion

DKMS provides a simple and unified mechanism for driver authors, Linux distributions, system vendors, and system administrators to update the device drivers on a target system without updating the whole kernel. It allows driver developers to keep their work aimed at the "top of the tree," and to backport that work to older kernels painlessly. It allows Linux distributions to provide updates to single device drivers asynchronous to the release of a larger

scheduled update, and to know what drivers have been updated. It lets system vendors ship newer hardware than was supported in a distribution's "gold" release without invalidating any test or certification work done on the "gold" release. It lets system administrators update individual drivers to match their environment and their needs, regardless of whose kernel they are running. It lets end users track which module versions have been added to their system.

We believe DKMS is a project whose time has come, and encourage everyone to use it.

## 9 References

DKMS is licensed under the GNU General Public License. It is available at

`http://linux.dell.com/dkms/`,

and has a mailing list `dkms-devel@lists.us.dell.com` to which you may subscribe at `http://lists.us.dell.com/`.