



How to Port a Driver from 2.4 to 2.6 Linux* Kernels

by Mark Gross

How to Port a Driver from 2.4 to 2.6 Kernels and Get it Included in the Upstream / Main line Kernel *

Mark Gross mark.gross@intel.com

July 12, 2006

Abstract

This tutorial describes how to port older legacy drivers from a 2.4 kernel to recent 2.6 kernels, and attempt to get the ported driver included in the upstream kernels.

This tutorial has two parts, and provides basic information and pointers to information on the mechanics of porting simple drivers from 2.4 to 2.6 kernels as well as udev, sysfs and the process of getting your driver into the upstream kernel. This is a minimal overview of porting drivers to the 2.6 kernel assuming a basic knowledge of 2.4 driver structure. It is based on my experience porting a simple legacy driver from 2.4 to 2.6, and getting that driver into the main line kernel. Examples and references are provided from this driver. Additional examples are taken from the port of the `riport.c` driver:

<http://marc.theaimsgroup.com/?t=112861156300005&r=1&w=2>

and

<http://marc.theaimsgroup.com/?t=115098732100003&r=1&w=2>

1 Syllabus

In part 1 of the tutorial, we look at the mechanics of porting from 2.4 to 2.6 and some of the new things you'll need to work with.

*Helpful review comments and input from Greg Kroah-Hartman, Randy Dunlap and Adrian Van De Ven gratefully acknowledged.

1. Mechanical changes you can expect between a 2.4 and a 2.6 driver.
2. Kernel build integration differences between 2.4 and 2.6.
3. An overview of using udev for your driver to get device nodes you can talk to.
4. Very brief introduction to sysfs programming and its use for legacy devices to provide an interface to user space.

Part 2 discusses working with the community. Its goal is to help you get your drivers into the mainstream kernel.

1. Getting your driver ready for LKML posting.
2. Setting up your email client for working with the LKML.
3. Knowing what documentation to read.
4. Understanding what to expect and ways to deal with it.

Part I

Mechanics of moving a 2.4 driver to 2.6

The following sections provide an overview of some of the activities to expect when moving a legacy driver from 2.4 to the 2.6 kernel. The first part of the port is very mechanical and trivial. However, if you want the driver to get into the upstream kernel trees you will not be able to stop at this point. You will need to use udev to get your device nodes created automatically, and use sysfs device attributes in place of IOCTLs. The bulk of this part of the tutorial provides information and an overview for working with these 2 aspects of driver implementation.

Your mileage will definitely vary.

2 Mechanical changes just to get your 2.4 driver to build and load under the 2.6 kernel

The task of building and running your 2.4 unit tests under the 2.6 kernel won't take a lot of effort. Some tools you will likely use in the process of executing your port will be :

- Tags. If you don't know how to use tags within VIM you HAVE to learn.

- Cscope

`http://cscope.sourceforge.net/`

For browsing a kernel tree

`cscope -Rk`

- An lxr web page

`http://lxr.linux.no/`

or

`http://lxr.free-electrons.com/`

- Bash shell, grep, find, editor, make, and the typical developer tools for doing software development under Linux.

As an example starting point, I'm using the telecom clock driver. The following URL is the posting of the 2.4.31 driver:

`http://marc.theaimsgroup.com/?t=111945397900009&r=1&w=2`

The easiest way to get started on your 2.4 to 2.6 port is to build your driver out of tree. Save off the 2.4 driver source in an out of tree directory. For this tlclk example, save off the LKML by posting to tlclk-2.4.31.patch and run

```
patch -p1 -f < tlclk-2.4.31.patch
```

to extract the starting source code for tlclk.c tlclk.h from the above posting.

Getting started is an easy thing to do when you use a trivial make file pointing at a known good 2.6 build tree containing a vmlinux (running this kernel is recommended but not required).

```
# point KERNELDIR to your 2.6 kernel tree.
KERNELDIR=/home/mgross/work/linux-2.6.16
obj-m := tlcclk.o
default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

At this point you need to do the compile-edit-compile cycle to clean up any compile failures. Keep in mind that kernel APIs evolve, and what builds and works for the 2.6.16 kernel may not compile with the kernels used by SuSE, RHEL, or Fedora. At the very least you can expect compile warnings when building against different kernels.

Porting the telecom clock driver from 2.4.31 to 2.6.16, I found that I needed to remove an include file that no longer existed, `tqueue.h`, and pull out the usages of `MOD_IN_USE`, `MOD_INC_USE_COUNT`, `MOD_DEC_USE_COUNT` that are now gone.

Make sure you build your driver for `i386` and `x86_64` if you can. Some problems are not exposed at compile time for one but will show for the other. I recommend you test build your driver under other architectures, the more the better.

For simple drivers there will probably not be many more changes needed. The following is a shopping list gathered from the `riport.c` driver port to 2.6.17 I recently did.

Examples of API differences you'll need to deal with include:

- `MODULE_PARM` macro is replaced with `module_param` (lower case) and its prototype now include permission flags. Note: for the 2.6.16 you'll get the error building for `i386`, where `x86_64` will build just fine, watch out.
- You'll want to use `MODULE_PARM_DESC`.
- `cli /sti` use must be replaced with proper spin locks, `spin_lock_irqsave` `spin_unlock_irqrestore`. While you are at it be sure to review your locking design. Its very likely that your driver was written for UP only and will have problems on an SMP / multi-core system.
- `init_module` exports are replaced by name space safe initialization, i.e. use `static int _init yournamespace_init()` with a `module_init(yournamespace_init);` declaration.
- `cleanup_module` exit exports with name space safe module exit definitions. i.e. Use `static void __exit yournamespace_exit(void)` with a `module_exit(yournamespace_exit)` declaration.

- Use the `MODULE_LICENSE` macro
- Use the `MODULE_DESCRIPTION` macro
- Don't use `check_region`, it is racy, and deprecated.
- Put the udev enabling code at the end of your initialization logic to avoid BUG calls if your driver fails its init function.

3 Differences in the in-tree build / makefile structures

There are no changes between merging your driver into the 2.4 and 2.6 in-tree Makefiles. However, there are some changes to the way you get your driver to show up in the make menuconfig screens.

For the telecom clock I added this to `drivers/char/Makefile`:

```
obj-$(CONFIG_TELCLOCK) += tlclk.o
```

To get the module to show up in the kernel configuration screens for the 2.4 kernels you only need to add one line to the correct `Config.in` file (e.g. `drivers/char/Config.in`) and then add the help content to another file.

For the 2.6 kernel it's a similar change only to a different file and with more options. For the 2.6 kernel you add config blocks of text to appropriate `Kconfig` files. (e.g. `drivers/char/Kconfig`):

```
config TELCLOCK
    tristate "Telecom clock driver for ATCA"
    depends on EXPERIMENTAL
    default n
    help
        The telecom clock device allows direct userspace
        access to the configuration of the telecom clock
        configuration settings. This device is used for
        hardware synchronization across the ATCA backplane
        fabric.
```

This is a more consolidated way to define the dependencies and help content within one file. See `Documentation/kbuild/kconfig-language.txt` for more detailed information.

At this point, you have a driver integrated into a kernel tree and it should configure and build. You should also be able to run your unit tests against this driver just as you did for the 2.4 kernel.

4 Introduction to udev and how it affects your driver and its use

For most well-behaved legacy device drivers under the 2.6 kernel, you'll need to avoid defining major and/or minor device node numbers and the manual setup of device nodes.

To support usability, udev exists for automatically setting up the device nodes based on events that come up from the driver model (kobject) infrastructure. Udev is implemented through a message-based protocol over a netlink socket. It is initiated from the `kobject_uevent()` API. This API tends to get called when a driver is associated with a device and it shows up within the `sysfs/class` directory hierarchy. For legacy devices, this call to `kobject_uevent` tends to happen when the driver is loaded. Other devices have this occur within the device probe operation to support hot plug.

udev gets a `sysfs` path message and knows to look within that path for a dev text file that it then parses for the major and minor device node values. Once the dev file is parsed, udev creates the dev-node within the `/dev` directory.

Using udev is pretty painless, but understanding all the plumbing built to make it work can be complicated. For most legacy devices there are two ways to use udev. One is by using the misc device and the other is to create your own class device.

To use misc device, you need to define a miscellaneous device structure and initialize the minor, name, and fops members. Then, within your module init code call `misc_register`. It handles creating your device node for you.

When the driver is unloaded, be sure to call `misc_deregister` to have your device unregistered and removed from the `sysfs/class/misc/..` directory. Udev is automatically invoked again to remove the device node.

If your device needs to create a number of device nodes, or you actually care about the values for your device node major / minor numbers, then you can also use the `class_device_create()` api. A nice and simple example of its use is in `arch/i386/kernel/msr.c`. To use `class_device_create` you need a class object. You can create one in your driver's initialization code with the `class_create` api,

then call `class_device_create` to get udev to do its magic.

`tlclk.c` example using the miscellaneous device:

```
static struct miscdevice tlclk_miscdev = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "telco_clock",
    .fops = &tlclk_fops,
};

...

ret = misc_register(&tlclk_miscdev);
if (ret < 0) {
    printk(KERN_ERR "tlclk: misc_register returns %d.\n", ret);
    ret = -EBUSY;
    goto out3;
}

....

/* Clean up code: */
misc_deregister(&tlclk_miscdev);
```

A sample from the `riport.c` driver

<http://marc.theaimsgroup.com/?l=linux-kernel&m=115135603811650&w=2>

on how you could set up a class node and udev support without using the misc device:

```
static struct class *riport_class;
...
static int __init riport_init(void)
{
    struct class_device *class_err;
    ...
    if ((result = register_chrdev(major, "riport", &drvriport_fops)) < 0)
        goto fail_register_chrdev;
    ...
    riport_class = class_create(THIS_MODULE, "riport");
```



```

        if (IS_ERR(riport_class)) {
            result = PTR_ERR(riport_class);
            goto init_fail_dev;
        }

        class_err = class_device_create(riport_class, NULL,
            MKDEV(riport.major, 0), NULL, "riport0");

        if (IS_ERR(class_err)) {
            result = PTR_ERR(class_err);
            class_destroy(riport_class);
            goto init_fail_dev;
        }
    ...
}

/* Clean up code: */

static void __exit riport_exit(void)
{
    ...
    class_device_destroy(riport_class, MKDEV(riport.major, 0));
    class_destroy(riport_class);
    ...
}

```

5 Introduction to sysfs and using it to replace your IOCTL function

To use sysfs, you need to set up some connections with the driver-model. The Linux driver-model was developed and documented back in 2003 as part of the 2.5 development kernel. It is also documented in the Linux Device Driver book, 3rd edition. There are also a number of locations for documentation you can find on the net. The problem with these resources is that they tend to be 3 years old and somewhat out of date.

Specifically; the Linux Device Driver book by Corbett, Rubini, and Kroah-Hartman has a number of sections that are out of sync with what is in the 2.6.17.1 kernel tree, `kset_hotplug_ops`, `class_simple_*`, that are now gone from the kernel tree.

The good news is that you don't need to understand all the driver model / sysfs

/ uevent implementation details to use it. For your driver you are looking to implement sysfs interfaces to replace your IOCTLs. For this you just need a device instance.

You could use the class device you created using `class_device_create`, but this would put your devices and attributes under the `/sys/class/YOUR_CLASS/your_device/` path. This would not be acceptable as it is expected that you put your hardware device attributes (register access) under the `/sys/device/...` tree, and the logical (protocol, logical interface) attributes exported by the driver under the class tree.

The rule of thumb is if your attribute changes HW state it probably is a device attribute, otherwise it is a class attribute. It's not always cut and dry, but it's also not a big deal to change things around if there is disagreement with your initial implementation.

There are a number of root level sysfs directory hierarchies under the driver model. You will only need to work with class and device objects for most legacy devices.

Class Entries under the `/sys/class` directory hierarchy are intended to provide a logical interface and view of the devices represented underneath. This is why udev is tied to class events.

Devices Entries under the `/sys/devices/` directory hierarchy are intended to provide interfaces to the physical hardware. This is where you should put attributes that effect actual hardware registers.

The API used for creating a platform device is `platform_device_register_simple`. It returns the device pointer with initialized `kobj` member so that you can attach your attributes as needed.

The telecom clock driver had a lot of IOCTLs. The fastest way to register attributes to your device is to use the `sysfs_create_group` API. Using this API consists of building an `attribute_group` structure that includes a null terminated attribute array of attribute pointers. You'll want to use the declaration macros for devices, `DEVICE_ATTR`. Declare your device attribute using `DEVICE_ATTR`, and fill in your store and show function names.

A sample on how to create the device object needed to attach attributes too:

```
static ssize_t show_current_ref(struct device *d,
                               struct device_attribute *attr, char *buf)
{
```

```

        unsigned long ret_val;
....
        return sprintf(buf, "0x%lX\n", ret_val);
}
static DEVICE_ATTR(current_ref, S_IRUGO, show_current_ref, NULL);
....
static ssize_t store_hardware_switching(struct device *d,
        struct device_attribute *attr, const char *buf, size_t count)
{
....
        return strlen(buf, count);
}
static DEVICE_ATTR(hardware_switching, (S_IWUSR|S_IWGRP), NULL,
        store_hardware_switching);

static struct platform_device *tlclk_device;
static struct attribute *tlclk_sysfs_entries[] = {
        &dev_attr_current_ref.attr,
        &dev_attr_telclock_version.attr,
        &dev_attr_alarms.attr,
....
        &dev_attr_hardware_switching.attr,
        &dev_attr_refalign.attr,
        &dev_attr_mode_select.attr,
        &dev_attr_reset.attr,
        NULL
};

static struct attribute_group tlclk_attribute_group = {
        .name = NULL, /* put in device directory */
        .attrs = tlclk_sysfs_entries,
};
....
        tlclk_device = platform_device_register_simple("telco_clock",
                -1, NULL, 0);

        ret = sysfs_create_group(&tlclk_device->dev.kobj,
                &tlclk_attribute_group);

....
/* Clean up code: */
        sysfs_remove_group(&tlclk_device->dev.kobj,
                &tlclk_attribute_group);
        platform_device_unregister(tlclk_device);

```

6 Driver model

You now know enough to get by to implement your driver for 2.6 kernel leveraging the sysfs interface and the udev mechanisms. You may be wondering what all this driver model business has to do with these new driver idioms. The driver model is the collection of lower level container classes and infrastructure that implements the handful of APIs and file system features needed to make the idiom work. The design and theory of operation of the driver model is beyond the scope of this tutorial.

Part II

Working with the community and getting your driver into the upstream kernels

This part of this tutorial is intended to prepare you for submitting your work for inclusion in the upstream kernel trees to the Linux Kernel Mailing List (LKML). There is a lot of information on these topics available. I'm not going to attempt to replace any of it. It is my hope that the following helps make it easier for new community members to participate in Linux.

- Getting your driver ready for LKML posting
- Setting up your email client for working with the LKML
- Knowing what documentation to read
- Knowing what to expect and ways to deal with it

7 Getting your driver ready for LKML posting

Here's a quick list of the types of comments I tend to get over and over again so you can avoid them yourself. First, re-read the Documentation/CodingStyle.

- Multi-line comments not formatted correctly

```
/*
 * bla bla...
 */
```

- Don't use C++ comment lines
- Name a maintainer for the driver, with yourself as the most likely victim. You do this by changing the MAINTAINERS file. Unmaintained drivers tend to not get included in the kernel tree.
- Be sure to address your email to the correct folks and CC the LKML.
- Be sure to include the signed off by line in your email before the patch.
- Not using static enough. Protect the kernel name space from pollution.
- Don't use mixed case symbols like iWasACppProgrammerNowIDoLinux-Drivers or variations on this theme.
- Remove white spaces at the end of lines. In vim, search for the regular expression.

```
␣$
```

- Remove code that's not needed, like assignments to zero for a structure that's been allocated with kzmalloc.
- Install 'sparse' and run make C=1 on your driver build to get have the space tool do a pass on your driver code. Clean up any warnings. See Documentation/sparse.txt for where to find the program.
- Don't miss __init declarations for linker optimizations.
- In if blocks the style police like to place constants to the right of the l-value

```
if (x == constant)
```

is better than

```
if (constant == x).
```

- Double check the indenting as it's easy to miss some indentation. If all else fails, you can run the scripts/Lindent tool to reformat your code. Be sure to carefully go through the Lindent output if you use it. It is famous for making code look bad in some places.
- Don't use compound statement blocks with only 1 statement in them.

```

If (ret) {
    bla;
} /* bad */

if (ret)
    bla; /* good */

```

- Keep in mind that `copy_to_user` doesn't return the same thing as user mode copy or write operations. It returns the number of bytes NOT copied.
- C99-style initialization of structure elements. This is the `.name = value,` style.
- Make sure you can handle an interrupt instantly after requesting the IRQ, i.e. before the code returns from the function block containing the `request_irq` call.
- For debug prints that are compiled in / out use `pr_debug` declared in `linux/kernel.h`
- Build the driver for multiple architectures and make sure to clean up any issues found.
You should build for multiple architectures and configurations to make sure things are clean on as many as you can handle. Try to include UP, SMP, as well as architectures that have differences in word sizes and if possible, little and big endian integers.
- Be sure to include a test pass with your driver built with a kernel that has all the debug options and `PREEMPT + SMP` enabled.
- Be sure to fix all the warnings, and don't just cast them away, really fix them.

8 Setting up your email client for dealing with the LKML

Email is the transport of information in the Linux community and there are standards for email behavior that are difficult to maintain without support of your email client. For instance, HTML and rich text (MS word) formatted email are universally rejected by the mailing list. Top posting will get yourself unfriendly feedback. There are multiple places to get documentation on mailing list behavior (it pays to know these expected norms). A good reference is from the ARM-Linux list, where a periodic posting of a set of such expectations happens once a week.

<http://www.arm.linux.org.uk/maillinglists/etiquette.php>
<http://www.arm.linux.org.uk/maillinglists/faq.php>

Another good FAQ you should read is the one from kernel.org:

<http://www.kernel.org/pub/linux/docs/lkml/>

The two most important points are no top posting and text only email. Your email client can help a lot with your compliance with the behavioral norms. I have tried a number of email clients with the community mailing lists, and though you can get by with Windows Outlook, and one of the GUI clients under Linux, I have had bad things happen to me with all of them. Mutt has been the only email client I've used, so far, that has not helped me to look dumber than I already do. Others may recommend pine or other clients; they might work well too, so give them a try if you like. I can only recommend mutt today.

9 Getting Mail

I'm using fetchmail to get my email from my account. You need your own version of my .fetchmail file:

```
poll your_pop3_mail_server_URL protocol pop3 user "your_UID" password "your_PW"
```

Fetchmail will complain about the access settings on your .fetchmail file, but just do what it asks.

To get my mail, I use one of the following command lines depending on where I'm getting my mail from:

```
fetchmail <-- from home
fetchmail ssl <-- at work where our sysadmin doesn't
                  allow the sendmail ports to be open
                  or in the clear.
```

10 Sending Mail

I'm using Open SuSE10, and I only needed to change the MTA setting in the YaST/Network Services/Mail Transfer Agent. I needed to set outgoing mail

server to "your_smtp_server_URL".

With the SuSE YaST configuring this isn't hard to muddle through.

11 Setting up mutt

The biggest trick to getting things working for me was setting the envelope_from item in my .muttrc file

```
set envelope_from = yes
```

You may also want to fiddle with the color body line in the muttrc so you can read strings that begin and end with _ like _this_.

```
color body brightyellow default "(^| )_[-a-z0-9_]+_[.,.?]?[ \n]"
```

Mutt doesn't have a very sophisticated address book. It uses aliases for email addresses that you can define using the following .mutt configuration file additions.

```
set alias_file = ~/.mutt/aliases
source ~/.mutt/aliases
```

When reading an email from someone you'd like to have in your alias list, just use the command 'a' in mutt to add the mail sender into your aliases. Then when addressing your email, use the tab key to auto complete / select from your list of aliases. The format of the alias records is simple, and you can hand edit the alias file to add entries and even distribution lists.

12 Spell Checking

There are not as many spelling police on the LKML as you would expect, but they are there. It's always a good idea to run your work through a spell checker. I like to use Vim and, for versions of Vim < 7.0 there is a must have plugin, VimSpell, that is very nice that works well with my Fedora and RHEL installations. I have had problems with using VimSpell on my SuSE installations

where Vim 6.x with VimSpell didn't show spelling errors in comment blocks, but luckily the Vim 7.0 spelling works great on both. I recommend Vim 7.

Spell check your work.

13 What documentation to read

Here is a list of documentation that is useful for getting your work included in upstream kernels. Most of this documentation is available on Greg KH's ddk CD.

<http://www.kernel.org/pub/linux/kernel/people/gregkh/ddk/>

Read the following documentation in more or less the following order of importance:

1. Documentation/CodingStyle : you need to read and follow as best as you can.
2. Documentation/SubmittingPatches
3. Documentation/SubmittingDrivers
4. Documentation/Submitchecklist <- a new file that is currently only in 2.6.17-mm tree
5. Unreliable Guide To Hacking The Linux Kernel : not a bad document.
6. Unreliable Guild to Locking : ditto
7. Documentation/HOWTO, this document is idealistic, but more or less it's true. However, there is a social hierarchy and it helps to know who the more influential folks are and what they say.
8. Documentation/driver-model/* : has the sysfs stuff, but it's only an overview and not too helpful in practice.
9. Documentation/filesystems/sysfs.txt
10. Documentation/kobject.txt
11. LinuxHelp_UDEVPrimer.html
12. Linux kernel Development (the newer edition, 2nd at this time.)
13. Understanding the Linux Kernel (the newer edition, 2nd at this time.)

14. The Linux Device Drivers book (the newer edition, 3rd at this time.)
15. <http://lwn.net/>
16. <http://lwn.net/Articles/driver-porting/> <- a bit dated but contains some helpful articles including the following:
17. <http://lwn.net/Articles/31185/> <- Device model overview
18. <http://lwn.net/Articles/51437/> <- The zen of kobjects
19. <http://lwn.net/Articles/54651/> <- kobjects and sysfs
20. <http://lwn.net/Articles/52621/> <- kobjects and hotplug events
21. <http://lwn.net/Articles/55847/> <- Examining a kobject hierarchy
22. <http://lwn.net/Articles/31370/> <- Driver porting: Device classes

14 Keeping a thick skin and not giving up

Working with the LKML and the Linux kernel community has its challenges. I'd like to give a heads up on what to expect, and how you can avoid digging holes you can't get out of or burning bridges.

The Linux community has had a reputation for being hard to work with. It is, but it is getting better and less insulting for the professional engineer. Things have been changing a lot over the past 7 years I've been watching. It is much easier to make contributions without getting flamed in a public forum by rude persons with obviously limited life experience. This isn't to say that you will not be given a hard time—it's just not nearly as bad as it used to be. There has been a noticeable effort by key LKML personalities to squelch the more childish behaviors.

In general, when you are given a hard time, take your time and think very carefully about how you respond to any flame bait or attacks. 99.99 % of the time, you are a lot better off to overlook any inflammatory comments and only address the technical comments. Don't take the bait. You may want to even snip the flame bait out of your replies when possible.

If you've ever participated in a peer review of your own code, then interacting with the LKML for a simple driver patch will feel familiar to you. Expect a lot of comments on formatting, coding style and spelling, followed by more technical comments on alternative APIs you could / should be using, dumb things your code is doing that it could do better, or things your code could do differently. Overall, you can expect a lot of good input to your code that is important to

incorporate into an updated version of the driver, that you will post for more feedback.

You may disagree with some of the input provided. After considering the input, if you still disagree, then don't feel afraid to respond with technical justifications why that input should not be applied to your code. This is the source of many interesting threads on the LKML. Just keep your responses technical and don't get personal or take things too personally. Chances are you'll learn something new, and start building your own credibility for working well with the community. Being right on some minor point is sometimes not as important as being able to work well with the community.

When you get into the space of dealing with code larger than your source module, things change a lot, as now you are in a design review situation. Talking design over email can be hard even when you have code implementing your design as a reference.

When you get into design discussions things can get much harder for you. Keep your senses, keep trying, and realize that sometimes you will not win, because a longer view of working with the kernel community is needed. Look for opportunities to up-level the discussion, try to get face to face time with your antagonist, and assume they are working for the good of Linux just like yourself.

Look for help from others to get across the importance of the problem you are trying to solve. For instance, look at the pain and suffering that has gone into getting low latency, RT changes or HRT, and the new time subsystems into the kernel.

Submitting drivers for review and consideration is relatively easy compared to getting subsystem designs agreed upon, so count your blessings that you are only trying to get a legacy driver ported and accepted into the upstream kernel.

15 What to expect and ways to deal with it

Now that you have gone through the LKML code review process and the motions of getting your driver patch accepted into the mm tree, and then the main line kernel, you will undoubtedly start getting emails about issues others have with your code. Most of these issues will be raised by Operating System Vendor (OSV) engineers stumbling over your code as they integrate it with their releases. This is a really good sign for you and your driver. Gratefully take their work and promptly evaluate and incorporate it into your driver.

Congratulations, you have now reaped your first Free and Open Source Software

(FOSS) process benefits. You have other engineers actually contributing work back to you. Depending on how the other engineer communicates the issue to you, after testing the changes, you may only have to do a group reply (reply all) with the following letters, ACK. Otherwise you may need to merge their work into yours and re-post the driver patch to the LKML.

After a few releases the emails drop off, and you need only smoke test your driver from time to time and respond to issues as they come. How many issues you'll have to deal with depends on how good / simple / well-used your driver is. For legacy devices like the telecom clock, this will be low maintenance. You'll have activity when your customers are using your hardware and even then it will likely be coming from the OSV engineer.

Finally, when you do have updates to your code that need to get into the next versions of the kernel, send the update patch to the maintainer. For odd-ball drivers like the telecom driver and your legacy driver this maintainer is Linus Torvalds. Sometimes you'll need to re-post your update a few times. If you find that your update isn't in the next Release Candidate (RC), then soon after that RC release you should re-post your patch against that RC to Linus, and CC the LKML. After a few times, the patch will make it into the mainline kernel—you sometimes just have to be persistent.

You now have a good idea on what to do to get your driver ready for inclusion in the 2.6 kernels and how to work with the LKML community.



Copyright © 2006 Intel Corporation. All rights reserved. BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo,

Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.