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.*

e100 Weight Reduction Program

Writing for Maintainability

Scott Feldman
Intel Corporation
scott.feldman@intel.com

Abstract

Corporate-authored device drivers are bloated/buggy with dead code, HW and OS abstraction layers, non-standard user controls, and support for complicated HW features that provide little or no value. e100 in 2.6.4 has been rewritten to address these issues and in the process lost 75% of the lines of code, with no loss of functionality. This paper gives guidelines to other corporate driver authors.

Introduction

This paper gives some basic guidelines to corporate device driver maintainers based on experiences I had while re-writing the e100 network device driver for Intel's PRO/100+ Ethernet controllers. By corporate maintainer, I mean someone employed by a corporation to provide Linux driver support for that corporation's device. Of course, these guidelines may apply to non-corporate individuals as well, but the intended audience is the corporate driver author.

The assumption behind these guidelines is that the device driver is intended for inclusion in the Linux kernel. For a driver to be accepted into the Linux kernel, it must meet both technical and non-technical requirements. This paper focuses on the non-technical requirements, specifically maintainability.

Guideline #1: Maintainability over Everything Else

Corporate marketing requirements documents specify priority order to features and performance and schedule (time-to-market), but rarely specify maintainability. However, maintainability is the *most* important requirement for Linux kernel drivers.

Why?

- You will not be the long-term driver maintainer.
- Your company will not be the long-term driver maintainer.
- Your driver will out-live your interest in it.

Driver code should be written so a like-skilled kernel maintainer can fix a problem in a reasonable amount of time without you or your resources. Here are a few items to keep in mind to improve maintainability.

- Use kernel coding style over corporate coding style
- Document how the driver/device works, at a high level, in a "Theory of Operation" comment section

old driver v2	new driver v3
VLANs tagging/ stripping	use SW VLAN support in kernel
Tx/Rx checksum of loading	use SW checksum support in kernel
interrupt moderation	use NAPI support in kernel

Table 1: Feature migration in e100

- Document hardware workarounds

Guideline #2: Don't Add Features for Feature's Sake

Consider the code complexity to support the feature versus the user's benefit. Is the device still usable without the feature? Is the device performing reasonably for the 80% use-case without the feature? Is the hardware offload feature working against ever increasing CPU/memory/IO speeds? Is there a software equivalent to the feature already provided in the OS?

If the answer is yes to any of these questions, it is better to not implement the feature, keeping the complexity in the driver low and maintainability high.

Table 1 shows features removed from the driver during the re-write of e100 because the OS already provides software equivalents.

Guideline #3: Limit User-Controls—Use What's Built into the OS

Most users will use the default settings, so before adding a user-control, consider:

1. If the driver model for your device class already provides a mechanism for the user-control, enable that support in the driver rather than adding a custom user-control.
2. If the driver model doesn't provide a user-control, but the user-control is potentially useful to other drivers, extend the driver model to include user-control.
3. If the user-control is to enable/disable a workaround, enable the workaround without the use of a user-control. (Solve the problem without requiring a decision from the user).
4. If the user-control is to tune performance, tune the driver for the 80% use-case and remove the user-control.

old driver v2	new driver v3
BundleMax	not needed – NAPI
BundleSmallFr	not needed – NAPI
IntDelay	not needed – NAPI
ucode	not needed – NAPI
RxDescriptors	ethtool -G
TxDescriptors	ethtool -G
XsumRX	not needed – checksum in OS
IFS	always enabled
e100_speed_duplex	ethtool -s

Table 2: User-control migration in e100

Table 2 shows user-controls (implemented as module parameters) removed from the driver during the re-write of e100 because the OS already provides built-in user-controls, or the user-control was no longer needed.

Guideline #4: Don't Write Code that's Already in the Kernel

Look for library code that's already used by other drivers and adapt that to your driver. Common hardware is often used between vendors' devices, so shared code will work for all (and be debugged by all).

For example, e100 has a highly MDI-compliant PHY interface, so use `mii.c` for standard PHY access and remove custom code from the driver.

For another example, e100 v2 used `/proc/net/IntelPROAdapter` to report driver information. This functionality was replaced with `ethtool`, `sysfs`, `lspci`, etc.

Look for opportunities to move code out of the driver into generic code.

Guideline #5: Don't Use OS-abstraction Layers

A common corporate design goal is to reuse driver code as much as possible between OSes. This allows a driver to be brought up on one OS and "ported" to another OS with little work. After all, the hardware interface to the device didn't change from one OS to the next, so all that is required is an OS-abstraction layer that wraps the OS's native driver model with a generic driver model. The driver is then written to the generic driver model and it's just a matter of porting the OS-abstraction layer to each target OS.

There are problems when doing this with Linux:

1. The OS-abstraction wrapper code means nothing to an outside Linux maintainer and just obfuscates the real meaning behind the code. This makes your code harder to follow and therefore harder to maintain.

2. The generic driver model may not map 1:1 with the native driver model leaving gaps in compatibility that you'll need to fix up with OS-specific code.

3. Limits your ability to back-port contributions given under GPL to non-GPL OSes.

Guideline #6: Use kcompat Techniques to Move Legacy Kernel Support out of the Driver (and Kernel)

Users may not be able to move to the latest `kernel.org` kernel, so there is a need to provide updated device drivers that can be installed against legacy kernels. The need is driven by 1) bug fixes, 2) new hardware support that wasn't included in the driver when the driver was included in the legacy kernel.

The best strategy is to:

1. Maintain your driver code to work against the latest `kernel.org` development kernel API. This will make it easier to keep the driver in the `kernel.org` kernel synchronized with your code base as changes (patches) are almost always in reference to the latest `kernel.org` kernel.

2. Provide a kernel-compat-layer (kcompat) to translate the latest API to the supported legacy kernel API. The driver code is void of any `ifdef` code for legacy kernel support. All of the `ifdef` logic moves to the kcompat layer. The kcompat layer is not included in the latest `kernel.org` kernel (by definition).

Here is an example with e100.

In driver code, use the latest API:

```
s = pci_name(pdev);
...
free_netdev(netdev);
```

In kcompat code, translate to legacy kernel API:

```
#if ( LINUX_VERSION_CODE < \
      KERNEL_VERSION(2,4,22) )
#define pci_name(x)  ((x)->slot_name)
#endif

#ifndef HAVE_FREE_NETDEV
#define free_netdev(x) kfree(x)
#endif
```

Guideline #7: Plan to Re-write the Driver at Least Once

You will not get it right the first time. Plan on rewriting the driver from scratch at least once. This will cleanse the code, removing dead code and organizing/consolidating functionality.

For example, the last e100 re-write reduced the driver size by 75% without loss of functionality.

Conclusion

Following these guidelines will result in more maintainable device drivers with better acceptance into the Linux kernel tree. The basic idea is to remove as much as possible from the driver without loss of functionality.

References

- The latest e100 driver code is available at `linux/driver/net/e100.c` (2.6.4 kernel or higher).
- An example of kcompat is here: `http://sf.net/projects/gkernel`