Uncanny: Scratching a Linux Device Driver Itch

Ian Osgood
Portland State Aerospace Society
Portland, Oregon
iano@quirkster.com
http://psas.pdx.edu

Draft of 2003/12/31 15:44

Abstract

The average application developer considers device drivers to be be deep voodoo. You're stuck if you don't have the driver for your hardware, and at the mercy of the driver writer for any that are available. This paper shows, however, that open source Linux drivers are by no means out of reach of even a kernel novice. I will show the path from an open source Linux CAN bus driver through the minor customization required for our embedded environment to the final completely rewritten driver tailored specifically to the needs of our application.

1 Introduction

The Portland State Aeronautics Society builds rockets with the eventual goal of a guided trajectory into orbit. The system we have designed to do this is a distributed system of PIC controllers and a 486 running Linux. The PIC nodes operate various sensors (GPS, accelerometers, pressure and temperature sensors) and functions (power, apogee detection, parachute ejection, manual override, active guidance). The Linux node integrates the data from the various nodes, coordinates them to determine attitude and position, and corrects the trajectory by controlling the guidance. The nodes intercommunicate using a CAN bus.

CAN (Controller Area Network) bus is a 1 Mbps two wire bus developed in the automotive industry for real-time distributed sensing and control. Data on the bus are sent serially, in variable length packets. Each packet has an 11-bit identifier, 1 bit identifying a Remote Transmit Request (RTR), a 4-bit data length, and 0-8 bytes of data. The bus is designed so that contention is resolved strictly via the message identifier; lower IDs have precedence.

2 Starting Point

The Flight Computer and Launch Tower Computer MOPS/520 boards contain a built-in Intel i82527 CAN interface chip. We found an open source Linux driver [1] which handles this chip, which we modified to work on the MOPS board. The main modifications were to extend the driver with a source module abstracting the details of our specific motherboard (interrupt number, memory map), add a plethora of kernel debug messages, and replace an overly complicated auto-detection of bus speed with hardwired values.

The Intel CAN interface chip presents fifteen message objects, each of which can be used to independently send and receive messages on the CAN bus. The existing driver presents 15 different devices, one per message object. This is good for a model where you can have multiple processes each in charge of reading or writing CAN messages. However, the architecture we developed for the Flight Computer (FC) only has one process which reads and writes to the CAN bus, and distributes messages to other processes as necessary. This mismatch in goals between our application and the driver led to underuse of the potential of the chip. It was also likely that the current driver architecture could not support the expected combined telemetry and data bandwidth of the fully functional Flight Computer.

In addition there were many required functions that were simply missing from the device driver. First, it appeared that RTR messages were completely unsupported. The driver code to support these messages was only half-written and extremely convoluted to deal with the stateful nature of these messages. Also, to handle these RTR messages, the incoming buffer to the `read` system call was significant, which breaks the standard Unix device driver interface specification, and lead to many wasted hours figuring out why our application code wasn't working. Our application used the `select` system call to multiplex CAN message handling with internal interprocess message handling, but we found only recently that the CAN driver did not support 'select'.

3 Driver Modification and Rewrite

Although I had written Unix apps and taken courses in Unix internals, I had never sullied my hands with an ac-

tual driver. I originally pored over the existing driver source trying to patch it up just enough to meet our needs. Incremental changes to an existing system often don't require the depth of knowledge required to rewrite an entire system. However, driver work on an embedded system is quite tedious. And after a few months it became clear that a full driver rewrite was the option of least resistance. The new plan was to develop a Linux CAN driver as simple as possible, tailored specifically to our hardware and performance requirements.

I started by finding material on the Internet about the proper structure of a Linux device driver [4, 5]. Immediately useful was the extensive documentation collected by the PSAS and made available through their Twiki web site. Through them, I found the Intel i82527 chip architecture specifications [2] and prior history on the MOPS board and the requirements for the driver's throughput. Then I took the existing source code and pared it down to those modules specific to our application. This reduced the project from sixty source files to eight source files: the general Linux driver framework and specific support for the MOPS board, the i82527 chip specific handler, the 'fileops' (read/write, open/close), and the corresponding header files. This phase consisted of removing from the overly general driver all that was not required by our specific hardware configuration.

Secondly, the driver was restructured to present only a single device. Internally, it made use of the chip's fifteen message buffers as it saw fit to handle the current communication demands.

At this stage, the driver was written without interrupts, polling the chip message buffers to read. This is easier to test and sufficient functionality for the Launch Tower Computer (LTC). The LTC simply accepts commands from the Launch Controller (LC) and retransmits them on the CAN bus, possibly returning status of the Launch Tower on demand. This is very low bandwidth, and polling is appropriate. The LTC is also a very simple system with LEDs for feedbac, and so is ideal for testing.

Later when the LTC checked out, support for interrupts and the `select` call were [will be] added to the driver. The implementation is similar to that of the original CAN driver, but much simplified and more robust.

4 Development Environment

The build environment was one of the first things to set up. The MOPS board runs the Debian Linux 'sarge' kernel version 2.4.18. Since this is not the same as our development machines, we first had to install the headers for that system and configure the build environment to use those headers.

The typical development cycle for the driver is:

1. build the driver on the development machine
2. `scp` the driver to the FC via the 802.11b wireless link
3. `ssh` to the FC and install the driver locally
4. restart the FC applications to check the functionality

Since there was no room on the Flight Computer for development tools like `gdb`, most debugging was through judiciously placed `printk` statements in the driver, which show up on the serial console provided by the MOPS board.

Virtual terminals are a nice feature of Linux, allowing a couple of terminals for editing, a terminal for building and copying, a terminal for the `ssh` connection to the FC, and a terminal running `minicom` for the FC's serial console. XTerminals could be used instead, but my development laptop could not have enough space devoted for an X installation. (This may change, seeing how small and efficient the OpenDesktop X server is becoming, thanks to the brilliance of Keith Packard [**?**].)

After each session, I check in the stable changed source to our CVS repository. I go to the PSAS Twiki and update the Linux CAN Driver page to reflect the group's newfound knowledge and current progress. The Twiki becomes my engineering notebook and communication tool in one. Hopefully, it will help the next generation of club members come up to speed on our project as quickly as I was able to.

4.1 Custom Tool: sucan

This methodology is pretty tedious, especially for the frequent changes required to explore the functionality of the i82527 CAN chip. Exploration was required because the published documentation didn't match the driver or Intel's own sample code [2]. It was desirable to have a more interactive means of exploring the chip. I discovered the `iopl` system call in Intel Linux, which allows a user-mode application (running as 'root') to gain full access to the I/O space, normally reserved only for device drivers. This allowed me to write a tool `sucan` which let me fully exercise the chip via a command-line interface. This greatly sped up the process of determining the correct I/O sequences to read and write RTR messages. Eliminating frequent driver reinstalls was crucial for getting enough time to experiment during a typical 3-4 hour weekly rocket meeting.

4.2 Custom Tool: CANtalope

Also of aid in this process was a custom CAN bus monitoring hardware, CANtalope, developed by the PSAS Avionics team. This was a generic CAN node, a small PIC circuit board which is the basis of all the CAN nodes in the Flight Computer, modified with LEDs to show the status of the CAN bus in real time, and with a serial

port interface for remotely sending and receiving arbitrary CAN messages from a development workstation. Some software was written for the workstation to conveniently display and log received CAN messages and send CAN messages with a variety of data formats. The software was used to confirm the proper operation of the Linux CAN driver revisions. CAN bus debugging was one of the last pieces of our development process that was not previously open-source, relying on a Windows application called CAN King.

5 Conclusion

The success of our rocket Avionics is predicated on the availability of open-source software for almost all aspects of the system:

- The main OS of Linux, to the drivers for the CAN bus and wireless network, the flexible development tools, and ease of installation of software and components thanks to the Debian package system.
- An open community where documentation and guidance is widely available.
- An open Twiki web system so that the PSAS's activities are immediately documented.
- An open rocket society so that new members can immediately become productive members of the team.

It is debatable whether our contributions are useful to the Linux community in the long run. I scratched a very particular itch, one which is not widely found in the Unix community. In the course of the work, there are many fixes we found which can be patched back to the original driver, but there has been no activity on that CAN driver since 2001 [1]. Does this mean we should take it over, or supplant it with our own driver? Of course, our driver is specific to our hardware, and the original one more general purpose with wider hardware support. But that generality got in our way.

The CANtalope diagnostic system could very well be of use to those doing development on CAN systems in any environment. The barrier here is the need to build your own CAN node, even though schematics, firmware, and software are open source and freely available. Just how much open source or shoestring-budget CAN development is there in the world?

I suspect it was easier to rewrite the driver from something known to work previously than it would have been to write from scratch. But who knows? We wasted a *lot* of time working around the previous driver.

There are several directions available due to open source. One is to make your product as general purpose as possible. The goal is for the product to spread wide, and hopefully the wide audience will generate more fixes, more features, and even broader support. But one is also allowed to go the other direction; to simplify and remove all that is not needed to meet one's specific needs. This work is likely to be a dead end as far as the evolution of the source is concerned, but results in a highly tuned application. Perhaps the simplified product can be the seed for future growth in new directions the original source could not have gone.

Tried and true methods were shown to work well. Take small steps. Test frequently. Commit frequently. Have others review your work, both to get the expertise of the masters, and to educate the newbies. Learn from the old code but don't be afraid to take it in new directions. Make good use of your tools (vim).

6 Acknowledgments

The PSAS team has been very fun and rewarding work with. During the last half year, I have learned enough to install Linux, and work on drivers in an embedded environment. I had no idea I would be writing and improving device drivers half a year ago when I first joined the PSAS. I'm glad to be working with such a great group, getting in touch with my inner geek. I would specifically like to thank Andrew Greenberg, the spiritual leader of our little group. Without his energy and dedication, this group may have fallen apart many years ago. His vast electronics and firmware experience led me out of many dead ends. I also thank James Perkins and Jamey Sharp for getting the CAN driver to work for the first time in our avionics system, for their general Linux expertise, and for guiding me through the Linux avionics code.

Thanks to Andrew, Peter Welte, and Tyler Nguyen for coming up with CANtalope, finishing it just in time to confirm the first successful operation of `sucan`.

Thanks also to Linux and the open source community. Without Arnaud Westenberg's original open source Linux driver [1], none of this would have been possible. Without the publicly available books about linux drivers and modules [4, 5] I wouldn't have been able to gain the knowledge I'd need to confidently rewrite the driver.

Thanks to Bart Massey and Keith Packard for guidance rewriting the CAN driver, and thanks to Bart for encouraging us to submit a paper to the Freenix conference and providing excellent TeX templates. A big thanks to TeXShop, a GPL Max OS X TeX editor and previewer, which allowed me to go from zero TeX knowledge to a full abstract in one day! It can be found at `http://www.uoregon.edu/~koch/texshop`.

7 Availability

All documentation and working notes for the projects of the PSAS are on the PSAS Twiki web pages at `http://psas.pdx.edu/`. The hub page for the CAN driver is at `http://twiki.psas.pdx.edu/bin/view/PSAS/CanBusLinuxDriver` [3].

All source code is publicly available through the PSAS WebCVS pages at `http://cvs.psas.pdx.edu/cgi-bin/cvsweb/c/can-linux/`. The old CAN driver code specifically is at `http://cvs.psas.pdx.edu/cgi-bin/cvsweb/c/can-linux/` and the new driver is at `http://cvs.psas.pdx.edu/cgi-bin/cvsweb/c/can-linux/uncanny`.

References

[1] Arnaud Westenberg's Linux CAN-bus Driver, `http://home.wanadoo.nl/arnaud/`

[2] *82527 Serial Communications Controller Architectural Overview*, Intel Corporation (1996), `http://twiki.psas.pdx.edu/pub/PSAS/FlightComputer/Intel_82527_Architecture.pdf`

[3] Ian Osgood et al, emphPSAS CAN Bus Linux Driver, `http://twiki.psas.pdx.edu/bin/view/PSAS/CanBusLinuxDriver`

[4] Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers, 2nd Edition*, O'Reilly & Associates, Inc. (2001), `http://www.xml.com/ldd/chapter/book/index.html`

[5] Peter Jay Salzman and Ori Pomerantz, *The Linux Kernel Module Programming Guide*, `http://tldp.org/LDP/lkmpg/lkmpg.pdf`, Open Software License (2003)