Writing Linux Device Drivers in Assembly Language

Written by Randall Hyde

0 Preface and Introduction

This document will attempt to describe how to write Linux device drivers (modules) in assembly language. This document is not self-contained; that is, you cannot learn everything you need to know about Linux device drivers (assembly or otherwise) from this document. Instead, this document is based on the text "Linux Device Drivers, Second Edition" by Rubini & Corbet (published by O'Reilly & Associates, ISBN 0-596-00008-1). That text explains how to write device drivers using C, this document parallels that text, converting the examples to assembly language. However, to keep this document relatively short, this document does not copy the information that is language-independent from the text. Therefore, you'll need a copy of that text to read along with this document so the whole thing makes sense.

Rubini & Corbet have graciously chosen to make their text freely available under the GNU Free Documentation License 1.1. Therefore, this text inherits that license. You can learn more about the GNU Free Documentation License from

`http://www.oreilly.com/catalog/linuxdrive2/chapter/licenseinfo.html`

The programming examples in this text are generally translations of the C code appearing in "Linux Device Drivers" so they also inherit Rubini & Corbet's licensing terms. Please see the section on licensing in "Linux Device Drivers, Second Edition," orthe text file LICENSE with the distributed software for more details on the licensing terms.

0.1 Randy's Introduction

As an embedded engineer, I've had the opportunity to deal with Linux device drivers in the past (back around Linux 1.x when the device driver world was considerably different). Most of the device drivers for Linux I'd dealt with were quite simple and generally involved tweaking other device drivers to get the functionality I was interested in. At one point I needed to modernize my Linux device driver skills (Linux v2.x does things way differently). As I mastered the material, I wrote about it in this document so I could share my knowledge with the world at large.

0.2 Why Assembly Language?

Rather than take the "just hack an existing driver" approach, I wanted to learn Linux device drivers inside and out. Reading (and doing the examples in) a book like *Linux Device Drivers* is a good start, but as I get tired reading I tend to gloss over important details. I'm not the kind of person who can read through a text once or twice and completely master the material; I need to actually *write code* before I feel I've mastered the material. Furthermore, I've never been convinced that I could learn the material well by simply typing code out of a textbook; to truly learn the material I've always had to write my own code implementing the concepts from the text. The problem with this approach when using a text like *Linux Device Drivers, Second Edition* is that it covers a lot of material dealing with real-world devices. Taking the time to master the particular peripherals on my development system (specific hard drives, network interface cards, etc.) doesn't seem like a good use of my time. Dreaming up new pseudo devices (like the ones Rubini & Corbet use in their examples) didn't seem particularly productive, either. What to do? It occurred to me that if I were to translate all the examples in C to a different programming language, I would have to really understand material. Of all the languages besides C, assembly is probably the most practical language with which one can write device drivers (practical from a capability point of view, as opposed to a software engineering point of view). The only reasonable language choice for Linux device drivers other than C is assembly language (that is, I *know* that I'd be able to write my drivers in assembly since GCC emits assembly code; I'm not sure at all that it would be possible to do this in some other language I have access to).

Rewriting Rubini & Corbet's examples in a different language certainly helps me understand what they're doing; rewriting their examples in assembly language really forces me to understand the concepts because C hides a lot of gory details from you (which Rubini & Corbet generally don't bother to explain). So that was my second reason for using assembly; by using assembly to write these drivers I *really* have to know what's going on.

Note that all the examples in this text are pure assembly language. I don't write a major portion of the driver in C and then call some small assembly language function to handle some operation. That would defeat the purpose for (my) using assembly language in the first place, that is, forcing me to really learn this stuff.

Of course, many people really want to know how to write Linux device drivers in assembly language. Either they prefer assembly over C (and many people do, believe it or not), or they need the efficiency or device control capabilities that only assembly language provides. Such individuals will probably find this document enlightening. While those wanting more efficient code or more capbility could probably use the C+assembly approach, they should still find this document interesting.

Of course, any die-hard Unix/Linux fan is probably screaming "don't, don't, don't" at this point. "Why on Earth would anyone be crazy enough to write a document about assembly language drivers for Linux? " they're probably saying. "Doesn't this fool (me) know that Linux runs on different platforms and assembly drivers won't be portable?" Of course I realize this. I'm also assuming that anyone bright enough to write a Linux device driver *also* realizes this. Nevertheless, there are many reasons for going ahead and writing a device driver in assembly, portability be damned. Of course, portability isn't that much of an issue to most people since the vast majority of Linux systems use the x86 processor anyway (and, most likely, the devices that such people are writing drivers for may only work on x86 systems anyway).

0.3 Assembly Language Isn't *That* Bad

Linux & Unix programmers have a pathological fear of assembly language. Part of the reason for this fear is the aforementioned portability issue. *NIX programmers tend to write programs that run on different systems and assembly is an absolute no-no for those attempting to write portable applications. In fact, however, most *NIX programmers only write code that runs on Intel-based systems, so portability isn't *that* much of an issue.

A second issue facing Linux programmers who want to use assembly is the toolset. Traditionally, assemblers available for Linux have been very bad. There's Gas (as), the tool that's really intended only for processing GCC's output. Most people attempting to learn Gas give up in short order and go back to their C code. Gas has many problems, not the least of which it's *way* underdocumented and it isn't entirely robust. Another assembler that has become available for Linux is NASM. While much better than Gas in terms of usability, NASM is still a bit of work to learn and use. Certainly, your average C programmer isn't going to pick up NASM programming overnight. There are some other (x86) assemblers available for Linux, but I'm going to use HLA in this document.

HLA (the High Level Assembler) is an assembler I originally designed for teaching assembly language programming at UC Riverside. This assembler is public domain and runs under Windows and Linux. It contains comprehensive documentation and there's even a University-quality textbook ("The Art of Assembly Language Programming") that newcomers to assembly can use to learn assembly and HLA.

I'm using HLA in the examples appearing herein for several reasons:

- I designed and wrote HLA; so allow me to toot my own horn,
- HLA source code is quite a bit more readable than other assembly code,
- HLA includes lots of useful libary routines,
- HLA is easy to learn by those who know C or Pascal,.
- HLA is far more powerful than the other assemblers available for Linux.

For more information on HLA, to download HLA, or to read "The Art of Assembly Language Programming" go to the Webster website at

`http://webster.cs.ucr.edu`

1 An Introduction to Device Drivers

Linux Device Drivers, Second Edition (LDD2), Chapter One, spends a fair amount of time discussing the role and position of device drivers in the system. Much of the material in this chapter is independent of implementation language, so I'll refer you to that text for more details.

There are, however, a couple of points that are made in LDD2 that are worth repeating here. First, this document, like LDD2, deals with device drivers implemented as *modules* (rather than device drivers that are compiled in with the kernel). Modules have several advantages over traditional device drivers including: (1) they are easier to write, test, and debug, (2) a (super) user can dynamically load and unload them at run-time, (3) modules are not subject to the GPL as are device drivers compiled into the system; hence device driver writers are not compelled to give away their source code which might leave them at a commercial disadvantage.

Rubini & Corbet graciously grant permission to use the code in their book on the condition that any derived code maintain some comments describing the origin of that code. I usually put my stuff directly in the public domain, but since my code is (roughly) based on their code, I will keep the copyright on this stuff and grant the same license. That is, you many use any of the code accompanying this document as long as you maintain comments that describe its source (specifically, LDD2 and Rubini & Corbet, plus mentioning the fact that it's an assembly translation by R. Hyde).

In chapter one, Rubini & Corbet (R&C here on out) suggest that you join the Kernel Development Community and share your work. While this is a good idea, I'd strongly recommend a flame-resistant suit if you're going to submit drivers written in assembly language to the Linux community at large. I anticipate that assembly drivers will not be well-received by the Linux/Unix community. Don't say you weren't warned.