by Bob Japenga (USA)

Getting Started with Embedded Linux (Part 4)

Linux Software Development Tools

This is the final installment in a four-part series introducing the topic of working with embedded Linux. Here you learn about the tools required for the software development process.

This article will discuss some of the software development tools available for developing embedded Linux. Let me start by identifying the basic steps and required tools in the software development process. The first tool that you need is a means to enter the software code into the computer. We'll call this "the editor." Then you need the means to take this representation of the code and turn it into instructions that the machine can read. Called in various environments, the assembler, compiler, linker and loader, we will call this "the code generator." Finally you need a means to debug, analyze and test your code. We will call this "the debugger." I am excluding some other management style tools (configuration control, librarians, etc) because these don't usually need to be unique to the hardware and software you are developing. In addition, a nice tool to have is called an integrated development environment (IDE) which ties all of these together into a single user interface.

THE OLD DAYS

The first software program I wrote was in 1969. It was programmed in FORTRAN and was supposed to solve a differential equation. The development process involved creating a flow chart (the design), writing the code on punch cards (the editor) and submitting the punch card deck to the computer center to be run in batch mode (the code generator). Several hours later, you would get a printout of the results. Debugging consisted of reading the results of your printout and figuring out what you did wrong. Often the first results did not indicate the answer to the equation but that the instructions were not in the proper syntax. Debugging was similar to the old adage from your shampoo bottle "lather, rinse, and repeat." We found ourselves in a seemingly endless cycle of punching the program in, submitting the batch to run, and repeating the process until the program ran cleanly.

A few years later, I was doing embedded programming on PDP-8s and PDP-11s. Software development was initially done by entering the programs from a printer with a keyboard (!) into a line editor that enabled us to change and view one line of our code at a time. We had a code generator that could be run from the command line from the terminal. The program would be installed and run. We looked at the results and tried to figure out what we did wrong. Most debugging was done by inserting some form of instrumentation into the software that would turn on a light or print something to a printer or to the user interface to indicate progress, status or some result. Sometimes we would insert a halt or a software trap into the code and read the results from the switch panel on the front of the computer. Such was the state of debugging embedded software in the early 1970s.

Using these "tools" we developed (albeit slowly) "powerful" controls that revolutionized an inherently unsafe bottle making process. Development continued this way for several years.

POWER TO THE PROGRAMMER

One day in the late 1970s, I was at a trade show and saw an integrated development environment (IDE) for developing software on a Motorola 6800 microprocessor. All the tools were together in one graphical user interface (GUI). You could write your code with their editor and could see 24 lines of the text at a time! You could do the code generation and the screen would point you to the exact place where the error existed. You could run the code from the same screen. You could actually stop the code in specific locations (break points) and view the results on the screen. You could step through the code and watch your variables change by name. You could stop the code when a location in memory was changed. I was in awe of having such a powerful tool. This was revolutionary.

Over the next 30 years, the software development tools advanced significantly enabling us to create even more powerful software. Real-time profiling (showing where your code is spending its time), memory leak detectors (monitoring buffer over-runs and dynamic memory allocation), code coverage tools (telling us that we have exercised every line of code) and many other enhancements came along.

DÉJÀ-VU ALL OVER AGAIN

Then we started developing embedded Linux applications. With the exception of our graphical editors, we went back to 1970s development. We did code generation from the command line. Errors were reported by their line numbers. We debugged by looking at the results. It was awful! I was incredulous. Over the first few years, we purchased a number of integrated development tools and tried some open source tools but found they were totally inadequate. They were slow. They were buggy. They didn't work. They were nonintuitive. Where were the memory leak detectors? Where were the code coverage tools? Where were the profiling tools? We had products to create, so we went back to the 1970s development paradigm with the exception of our editors. We compiled from the command line. We debugged by adding instrumentation (printf statements to the console). Lather, rinse, and repeat.

"If you tried to develop embedded Linux systems more than four years ago and gave up in disgust because of the inadequacy of the tools, it is time to take another look. I will describe two open source IDEs we use in our company. There are many, many more. That's why we call this column 'Embedded in Thin Slices.'"

HOW DO YOU SPELL RELIEF?

Thankfully, over the past five years, this has changed for the better. I want to talk about two tools that are currently available. If you tried to develop embedded Linux systems more than four years ago and gave up in disgust because of the inadequacy of the tools, it is time to take another look. I will describe two open source IDEs we use in our company. There are many, many more. That's why we call this column "Embedded in Thin Slices."

ECLIPSE

Initially developed by IBM in 2001, Eclipse is a full featured and extensible (through plug-ins) cross platform IDE written in Java available for Windows (32 bit and 64 bit), Linux and Mac OS X (Cocoa). With it, you can design, develop and debug C/C++, Java, JavaScript, Perl, PHP, Python, Ruby (including Ruby on Rails), and many other languages for Linux (and other systems). In 2004, IBM released Eclipse to the open source community and formed a separate nonprofit foundation (The Eclipse Foundation) to support it. It is funded by membership dues in the foundation. Many of the major players in the software industry are members such as IBM, Oracle, Cisco, NEC, Intel, and Mentor Graphics.

We have used Eclipse since 2007 to develop embedded Linux applications. Of all of the IDEs available for Linux development it has by far the largest number of plug-ins to extend the functionality of the tool.

EDITOR

We are using the standard editor provided with Eclipse for almost all uses. As old-time developers, we compare everything to the old DOS-based Brief. It was the most powerful and intuitive programming editor we have ever used. In our opinion, it has all been downhill from there. After Brief's demise, we switched to CodeWrite and it was almost as good a Brief. The good news is that we have found the Eclipse editor now almost as good as CodeWrite. We still cannot do macros as easily as we did with Brief. It doesn't work well for shell script programming— but we can easily configure Eclipse to seamlessly use a different editor for different file types. On the plus side, it does an adequate job displaying your errors following a compile, including knowing when the problem is a make file error. It handles conditional code (# defines in C) seamlessly—knowing from your project what code is functional and "graying" out the code not being used. It nicely points to where variables are used. As long as you are using a reasonably modern PC and are the only user, it runs acceptably. You can tell the Eclipse editor what your coding standard is and impose it upon all of your projects. We like the ability to handle refractoring (changing the source code throughout the whole project without affecting the executable code).

In summary, "Welcome to the 1990s Mr. Banks." The editor has almost gotten us back to what we had 20 years ago.

CODE GENERATION

All the IDEs we have used for embedded Linux use the GNU C/C++ compiler and linker for code generation. Depending on the processor you are developing for, you can obtain the GNU cross compiler that meets your needs. Having used C compilers for more than 30 years, we have found fewer bugs in this compiler than all our previous compilers. As with most general-purpose IDEs, Eclipse enables you to define your toolchain on a per-project basis. One of the big advantages of Eclipse in the embedded world is the fact that the Yocto Project is building plug-ins for Eclipse to better integrate with OpenEmbedded. When completed, the ease of development and managing embedded Linux systems should be greatly increased.

DEBUGGING

Support for JTAG- and Ethernet-based target debugging has been around for a long time. You can launch the code on a target using the GNU debugger over an Ethernet port or using a JTAG device and debug your application. Many of the JTAG manufacturers provide Eclipse plug-ins for their devices. In addition, Eclipse is doing a better job handling the debugging of threads. Even TurboC and Visual Studio never did that very well.

OTHER TOOLS

Eclipse provides good integration with source control tools like subversion and git. Even more important, independent of source control, it transparently manages your versions of code. For memory leak detection, we have successfully integrated with Dmalloc but have not been able to make it seamless with Eclipse. This is often the problem with any software package as large and as flexible as Eclipse. The learning curve can be steep.

WISH LIST

We would like to see a plug-in that would give us better visibility into the vast array of information Linux provides. For example, we would like to be able to monitor queues and semaphores at a higher level of abstraction. We would like it not to crash as much. We would like to see it run faster without having to liquid cool my PC. We wish that the learning curve wasn't so steep. Finally, isn't there some way they could keep the interface the same? It seems to be always changing. If you need to upgrade to use a new plug-in or to fix a bug, you feel like someone has completely rearranged all of the rooms, closets, and drawers in your home.

NetBeans

NetBeans originated as a Java IDE but has been expanded to become an IDE for PHP, C/C++, and JavaScript. MicroTools originally started using it in 2007 as an IDE for Java development on an embedded Linux system. Developed as a student project in 1996 by Sun Microsystems and released as open source in 2000, it initially was funded by Sun but is now primarily supported by Oracle.

Overall, NetBeans offers all that Eclipse offers. All of the pros and all of the cons are summarized in one word: ditto.

NetBeans manages multiple projects a little bit better than Eclipse. One single-board computer manufacturer is moving away from Eclipse because of that very reason (but they are moving to Code::Block with which we have little experience). NetBeans is a little faster on the same machine. On the down side, it is not tied into OpenEmbedded or the Yocto Project.

SUMMARY

We have developers at MicroTools who are satisfied with both. Eclipse has more big-name software companies behind it as members than NetBeans. Eclipse has many more plug-in than NetBeans. That means there will be more bugs (who can test every plug-in's interaction with every other plug-in?). There are more JTAG interfaces available for Eclipse than for NetBeans.

The bottom line is that the development tools for Linux are now almost as good as what we had in the 1990s but much better than what we had in the 1960s and 1970s. ■

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

RESOURCES

Code::Blocks, www.codeblocks.org.

Dmalloc - Debug Malloc Library, http://dmalloc.com.

The Eclipse Foundation, www.eclipse.org.

Git, http://git-scm.com.

NetBeans, "A Brief History of NetBeans," http://netbeans.org/about/history.html.

Subversion, http://subversion.apache.org.

Yocto Project, www.yoctoproject.org.