How to Port Linux When the Hardware Turns Soft

David Lynch

Abstract

Porting Linux to run on the Pico E12 and beyond.

In software development, possibly the most mystical and prestigious effort is taking dead hardware and breathing life into it -- porting an operating system to a new platform -- the mythical land of wizards and gurus, the software side of *The Soul of a New Machine*. I had performed almost every other software development task, and I wanted a chance to conquer this one.

I had been working with Linux and open-source software for many years. I am a fairly competent software developer (with hardware experience), but prior to starting the E12 port, I had done little more than tweak a Linux driver and build custom configured kernels. I was fortunate to have a friend building a new company that was developing one of the smallest embedded systems available, the Pico E12. I practically begged for the opportunity to put Linux on the E12. “A man’s reach should exceed his grasp, or what’s a heaven for?”

The E12 used a Xilinx Virtex 4 FX20 FPGA (Field Programmable Gate Array) that included a 300MHz PowerPC 405 processor, 128MB of memory and 64MB of Flash ROM. I bought a Macintosh Lombard PowerBook Laptop on eBay, as a sort of simulator for the E12. It also provided a way to write for the E12 without a cross compiler. While waiting for the E12 to progress far enough to start working with it, I scoured the Web for information about Linux porting and developed competence in PowerPC assembly language. Linux kernel programming is primarily in C, but small parts of the Linux kernel -- parts critical to putting Linux on a new system are in assembler. I have programmed in many assemblers -- once writing the standard C library in x86 assembler, but PPC assembler was new and took a day or two to learn. Linux had been ported to PowerPCs, even a different Xilinx FPGA, long ago.
I have a reference library of software books that fills a three-car garage. With few exceptions, they gather dust. My primary research tool today is a broadband Internet connection and a search engine. There are vast resources available on the Web for Linux developers. The Linux Device Driver guide -- the Linux bible for device drivers -- and numerous mailing lists target all aspects of Linux systems development. Kernel-Newbies is a great place to start (see the on-line Resources). There are mailing lists for every Linux subsystem. And, there are several Linux PowerPC mailing lists -- one specific to embedded PowerPC Linux. At the root of this tree is LKML, the Linux Kernel Mailing List. LKML is Mount Olympus -- the home of Linus, and the other Linux gods and titans. There are Web pages documenting the experience of others porting Linux to specific boards. Finally, the ultimate reference -- the Linux kernel source -- is available on kernel.org.

Finally, the E12 was far enough along to start work, and I received one via FedEx. I had documents and specifications, but actually holding one made it real and answered questions that could not be read from the specifications.

The E12

The E12 is a Compact Flash card -- exactly like those in many digital cameras. It has only two connectors: a CF bus connection on one end and a 15-pin miniature connector on the other. There are no other external connections. The E12 is based on an FPGA. There are a few additional components, and a few fixed elements, such as the PPC405.
CPU on the FPGA. A large part of the hardware is programmable. Most external connections are through the FPGA. Almost none of the “hardware” has form or meaning until the FPGA is loaded. Changing the bit file on the fly drops in a completely new hardware design. Welcome to a new era -- even the hardware is software. The BIT image -- in essence the program for the FPGA -- is created by an FPGA developer, programmed into the Flash ROM and automatically loaded into the FPGA on power-up. Once this BIT image “boots”, hardware is created in the FPGA. Now, the pins on the connectors have meaning. The 15-pin connector provides three external connections for internal devices. It supports Ethernet, serial and JTAG connections through custom cables. The CF connector provides a bidirectional interface to a host -- in most instances using a CardBus or PCMCIA adapter. Most of the pins on either connector can be whatever the FPGA programmer chooses to make them. Fielded systems may be plugged in to a CF connector solely to get power. E12s are used in daughter cards in typical embedded applications, on bus boards in high-performance computers in clusters and for applications, such as image processing or code cracking. They also are being used in applications with no operating system or extremely minimal operating systems.

Getting a Pulse

My first objective was to write the proverbial “Hello World” program for the E12. I spent a few days and wrote two different “Hello World” programs: one for the keyhole FIFO and one for Xilinx uartlite port.

Now, I was ready to attack Linux. I decided to start with Linux 2.6. There were numerous issues -- good reasons, as well as respected and conflicting opinions favoring both 2.4 and 2.6. I elected to use Linux 2.6, because I eventually was going to have to move to 2.6 anyway. Initially, I used the PowerBook to configure and build my Linux kernel for the Pico E12. This allowed me to start without cross compilers. Eventually, I switched to building inside of a coLinux virtual machine on Windows hosting the E12. Most Pico clients are doing Windows-hosted development. It was critical that everything work in that environment. Besides, building a PowerPC Linux kernel in a Linux virtual machine running Windows and loading it into a PowerPC, means that Linux outnumbers Windows 2 to 1 inside my laptop.

I used the Xilinx ML300 as a template to create a new Linux BSP (Broad Support Package). I grepped the kernel source for all references to the Xilinx ML300. I copied and renamed all ML300-related files to new files for the Pico E12. There were four completely unique files for the E12:

- arch/ppc/platforms/4xx/pic0-e1x.c: board-specific setup code.
- arch/ppc/platforms/4xx/pic0-e1x.h: headers and data structures for the board-specific setup code.
- arch/ppc/platforms/4xx/parameters/parameters_pic0-e1x.h: a set of hardware definitions created by the Xilinx software that created the bit image for the FPGA.
- arch/ppc/configs/defconfig_pic0-e1x: a default Linux configuration file for the E12.

There were major similarities between the Xilinx ML300, but there were a few specific differences. The E12 deliberately implements a lot less hardware. The E12's purpose is to provide a very minimal base platform, with the largest percentage of FPGA left for the client. The minimal useful Linux configuration must have either Ethernet, a serial port or the keyhole port. The default E12 does not have an interrupt controller -- the PPC405 provides a timer interrupt that does not require a PIC. The E12 also uses the Xilinx uartlite uart, not the much larger and more common 16550 uart. There were no Linux drivers for the uartlite. Two other ML300 files, generic support for Virtex FPGAs required minor modifications.

The next major issue was learning the Linux configuration system. I was not able to find much documentation. With Linux kernel programming, the two primary resources are the Linux source itself and the mailing lists. Sometimes, there is excellent documentation for a system; sometimes there is nothing. Sometimes I found documentation in some obscure corner of the Web -- after I had figured things out on my own. I had to develop enough understanding of the kernel build system to add a new board, some new configuration options and a few new drivers to the build system.

The first element is the Kconfig files in most of the Linux source directories. Kconfig is a cross between a very, very simple scripting language and a menu construction language. The entries in Kconfig files determine the menu structure and choices that you get when you execute any of the Linux menu configuration build options -- make oldconfig, make menuconfig, make xconfig.

I had to create a new menu item under the ppc 4xx menu for the Pico E12, menu items in the drivers/serial/Kconfig file for the uartlite and keyhole serial ports, and a small collection of menu items for other options. The syntax for the Kconfig items I needed to create could be easily worked out by inspection and a small amount of trial and error. I copied blocks for similar objects, made name changes as needed, and without too much effort, it worked. Inside the .config file, source code and Makefiles, the configuration items defined in Kconfig files are prefixed with CONFIG_. After the Kconfig entries were created, entries needed to be added to the matching Makefiles. This mostly involved copying similar objects and making name changes, and except for a few very complex choices, was pretty easy.
How to Port Linux When the Hardware Turns Soft

http://0-delivery.acm.org.innopac.lib.ryerson.ca/10.1145/1200000/11...
My confidence in my serial drivers was not high at that point. Further, the Pico minimalist mantra does not include networking as part of the base Linux, and many E12/Linux applications do not need it.

The root filesystem can be on a hard disk (none readily available in the E12) or in Flash. The E12 uses a very simple Pico File System, but one that is not suited for a root filesystem. Another alternative was to put the root filesystem on a RAM disk. Linux provided the ability to use and populate a RAM filesystem as an intermediate step in the boot process. One objective was to migrate as much of the Linux boot code out of the kernel to user space as possible. Linux systems going back many years boot through initrd, then execute a pivot_root to switch the root filesystem from the initrd RAM disk to the disk-based root filesystem. Using initrd requires the loader to copy the compressed Linux image and the separate image of the contents of the initial RAM disk into memory, and provides Linux with a pointer to the initial RAM disk data.

Linux 2.6 introduced a new variation -- initramfs. One difference between initramfs and initrd was that with initramfs, the contents of the initial root RAM disk filesystem were compressed into the Linux image during build, so there was only one file -- in my case an ELF file -- to load. This meant that the Pico monitor would not need changes. This initramfs approach proved to be extremely clean, simple and easy to use. Getting it working was complex and time consuming, because initramfs is fairly new. The primary documentation is a collection of posts to LKML. To create an initramfs for the Pico E12, I determined that I needed to create a directory on my build system and populate it with the files for the root filesystem. I enabled the initramfs option using menuconfig and told menuconfig where to find the directory that represented my root filesystem. There are a few other ways to do this, but that was the simplest. Initially, I decompressed the initramfs from the Gentoo Linux install on my PowerBook, and eventually switched to a cross-compiled BusyBox when I erroneously thought I might be having problems with my boot image, because the binaries were built for the PowerBook, not a PPC405.

First Words

After this, I hit my next problem. Linux was booting all the way through to executing /init where it just stopped. I wrote a trivial version of ls and included it in the kernel, calling it prior to exec'ing /init. Everything was fine. But on exec'ing /init, Linux became deaf and dumb. Debugging can be particularly difficult when the horses look like zebras. I spent a lot of time tracing through the Linux exec process, which was remarkably ingenious in many instances, doing minimal work and loading a process through page faults. Unfortunately, this made tracing what was happening very difficult and led me once again to the (almost) erroneous conclusion that I had a virtual memory problem. I wrote a Linux version of “Hello World” in PPC assembler with no external libraries and was able to execute it as /init. But, I could not exec anything more complex. I eventually found and enabled system call tracing and was able to watch as /init executed. The system always died while in the middle of virtual memory operations. I ended up with failure cases when Linux would go dumb right in the middle of outputting some debug string -- again, always during a VM operation. I could actually change the point of failure by inserting additional debugging. I was a victim of the Heisenberg uncertainty principle -- observation changed the observed behavior.

I was sure something was wrong with my serial drivers, despite the fact that this did not make sense, but how else could output stop in the middle of a string? All the critical clues were present to solve this problem, though one of them was buried as an artifact of the machine-check problem. This was a VM problem, in a twisted sense, and it was a serial driver problem. I will not confess to how long it took for the answer to dawn on me. Let's just say I rewrote the serial drivers several times before I saw that although the serial drivers requested and saved a virtual address for the memory mapped hardware -- partly as an error induced by using the 8250 serial driver as a starting point -- the virtual address for the serial port was subsequently getting overwritten by the physical address of the port. Because in my efforts to debug the machine-check problem I put a 1-1 physical-virtual mapping directly into the MMU Translation Lookaside Buffer, I/O continued to work until the Linux VM system overwrote my temporary TLB entry. After recognizing this, it took less than 30 minutes to correct, and I was able to boot up Linux to a bash prompt.

The End -- the Beginning

Little matches the thrill of seeing a new machine reach a shell prompt and knowing I made it happen. I had completed my base Pico E12 Linux port. Well, that is not quite true -- no port is ever finished. When I completed my Pico E12 port, I was unaware of any other port of Linux to a Xilinx V4 chipset. Subsequently, a Linux port for the Xilinx ML403 by Grant Likely started working its way through Linux embedded PPC development trees and has been accepted into the distribution kernel. The Pico E12 is distinct from the ML403, but they are more similar to each other than the older ML300 from which I started. Grant's ML403 port reflected changes that were impacting the whole Linux PPC development tree, so I made my Pico E12 port track those developments.
I have depended on the keyhole port for hosted development work, and as a result, the keyhole serial driver gradually has grown smaller and more consistent with the direction in which Linux serial drivers seem to be headed. I will have to update the uartlite driver to catch up.

I am currently on my third iteration of a Linux network driver for the E12, and Pico is on its second iteration of the underlying network hardware. The new network hardware is interrupt-driven requiring a PIC.

A second Pico board has matured, and with minimal changes, the Pico E12 port has evolved into the Pico E1X port.
I am working to get the Linux Memory Technology Device (MTD) system to work with the Pico Flash. This is complicated by the fact that the Flash in Pico hardware can be read and written to by both Linux and the host, and Pico is eventually planning on Flash device sizes that should be windowed into Linux memory instead of mapped in their entirety.
Once the Linux MTD work is completed, Pico wants a Linux (and Windows) filesystem driver for its simple filesystem -- PicoFS.

Pico is considering changing the keyhole port so that on the host side (and possibly the target side), it sufficiently and closely resembles an 8250-compatible UART to use only the OS's native serial drivers.

Later, Pico developed a daughterboard for the E12 called the Little Brother Board that allows using the E12 in a non-hosted environment and includes three USB ports, an LED and several other hardware components. In one application, the E12/LBB combination is being used as a very high-performance Webcam.

Figure 5. E12 in Little Brother Board Daughter Card

The E12 also can be hosted as a grid on a bus board called the supercluster. Currently, that configuration is used for blazingly fast code cracking, using FPGA hardware without OS support, but Linux HPC support is on the wish list. Higher performance can no longer be achieved simply by doubling the clock every 18 months. Clusters are a significant alternative; 16 E12s provide enormous horsepower while occupying little space and consuming little power.

There have been several iterative releases to Linux 2.6 since the E12 port, and occasionally, these require changes to the port.

I would like to get the Pico E1X port and drivers I wrote for it included into the Linux distribution kernel. Within the Linux embedded PPC mailing list, there has been some interest in seeing that happen. The code has been moved into git to make it easier to merge with new Linux iterations and to produce patch files for submission to LKML.

“The bigger game was ‘pinball’. You win one game, you get to play another. You win with this machine, you get to build the next. Pinball was what counted.” Tracy Kidder -- *Soul of a New Machine*. I won at pinball.

I got Linux up and running on new hardware, and other opportunities with other hardware and with other embedded OSes have occurred. Board bring-up for the E12 was hard. Somewhere on Kernel-Newbies I read advice to newbie kernel hackers to lurk on the mailing lists for a few years before attempting anything serious -- advice I am glad I did not take. I did not start this as a complete novice. I had a lot of experience that made this much easier. It was thrilling, mythical and magical. I can call myself a Linux Kernel Developer -- though maybe not too loudly around Linus Torvalds, Andrew Morton or Alan Cox. But, it was not more difficult than many other software tasks -- just more rewarding.

**Acknowledgement**

I would like to thank Dr Trout at Pico for paying me for projects I would do for free.

**Resources for this article:** [http://www.linuxjournal.com/article/9462](http://www.linuxjournal.com/article/9462)