LINUX JOURNAL

Embedded Linux on the PowerPC

By M. Jones
Created 2001-07-20 01:00

The x86 isn't always the best platform choice. Tim shows us how easy it is to work with embedded Linux on the PPC.

Although Intel provides the greatest user base for Linux, many other architectures are supported. These include ARM, MIPS, PowerPC, Alpha, SPARC and Hitachi. The availability of cheap x86 hardware makes Intel a simple choice, but in cases where performance is important, the PowerPC (and others) can't be ignored.

This article will demonstrate Linux using the MontaVista Hard Hat Linux distribution on an Embedded Planet PowerPC board. We'll also discuss how to flash Linux with a ramdisk root filesystem onto the hardware for self-boot.

Why PowerPC?

With the ubiquity of x86 and available tools, why choose anything else? A quick summary of the architecture may answer that question.

PowerPC is a highly integrated RISC architecture optimized for communication systems. The PowerPC combines a PowerPC processor core with a CPM (communications processor module) that offloads traditional communications tasks, giving the core more cycles for the actual processing requirements. The CPM also permits a variety of serial communication controllers (SCCs) to be programmed with a particular personality chosen from the set available from the core (such as Ethernet, SDLC, IDA or standard serial).

The other driving force behind PowerPC is its power consumption. For applications that require high performance with low-power consumption, PowerPC is a great choice. Tie this altogether with wide Linux and tool support, and you have a very capable solution.

Environment Setup

The MontaVista distribution is quite easy to install and configure, and their 100-page CDK document provides details on host configuration, building applications and other topics.

One of the interesting features of the MontaVista distribution is they not only provide configurable kernel source (for a variety of hardware architectures and platforms) but also a tested set of application packages for each architecture. These packages include the Apache web server, standard network tools (including **ipchains**, **ipmasqadm**), Perl, SNMP and many others.

Porting apps to the particular architecture can sometimes be nontrivial, but in the Linux Support Package (LSP), MontaVista has already integrated much of what you need.

As far as your development host goes, you must have an Ethernet port and an available serial port to work with the target. More on this later.

The RPX Net

For demonstration purposes, I've chosen the RPX Net board from Embedded Planet (see Figure 1). The RPX Net is a PC/104 form-factor PPC855-based board that includes 32MB of DRAM, 8MB of Flash, a Fast Ethernet Controller (FEC), four-port LAN hub, a standard serial port and other interfaces. This particular board can serve as an embedded gateway to provide IP-masquerading (network address translation) and port forwarding between an external WAN and the internal LAN.

Figure 1. RPX Net

While MontaVista supports a variety of Embedded Planet boards directly through their LSPs, the RPX Net is not currently one of them. (MontaVista currently supports the RPX Lite, Classic and Credit Card boards.) Therefore, a number of changes were necessary to support the new Ethernet devices. In most cases, you can pick the LSP that most closely matches your hardware and then modify for the differences.

The RPX Net also has PlanetCore utilities, which include a bootloader, diagnostics software and an external suite of tools for embedded development. PlanetCore can be used for board diagnostics (for any EP board, including I/O modules) and also for OS-neutral boot loading (in the absence of GRUB or LILO), RAM-based software

downloads and host-based Flash burning. We'll illustrate RAM-based downloading and flashing of a kernel shortly.

Setup

Developing with the RPX Net (and virtually any other embedded network board) requires two connections between the host development system and the target. First, a network connection is required for high-speed download. This can be a Cat-5 crossover cable if the host is connected directly to the board or a standard Cat-5 cable if the host and target are connected through a HUB. A serial connection is required for PlanetCore communication and is provided through a standard DB-9 connector. A terminal emulator, such as minicom, can be used for serial communications with the RPX Net.

The ramdisk Script

The purpose of the ramdisk image is to provide a root filesystem for our kernel when it boots. This implies that the filesystem will contain a number of things, including filesystem structure, binaries (such as a shell and utilities), configuration files (such as rc.sysinit), device entries (/dev/kmem, etc.) and runtime libraries.

We'll build our kernel with ramdisk using a single script. This script will perform all of the necessary functions to build a kernel, initial ramdisk and bind them together for Flash (see Listing 1 at ftp://ftp.ssc.com/pub/elj/listings/issue04/ [1]).

The first task of the script is to define a few symbols so that the script knows where to get the necessary elements, like binaries and libraries. These are configurable, as their location may differ in your setup.

After some general cleanup (removing old compressed ramdisk images), we'll create our blank ramdisk image with **dd** and then make it an ext2 filesystem with **mke2fs**. It's important that the ramdisk image is zeroed to aid in efficient compression of the filesystem later on. Next, we mount the ramdisk image on /mnt using the loop device. The loop device allows us to treat a raw file as a device and then mount a filesystem on it. Your development host's Linux kernel needs support for the loop device; if it's not available, a quick host kernel rebuild will be required (see ``loopback device support'' in the block devices configuration section of kernel config).

At this point, we have a 4MB ext2 filesystem mounted on /mnt. We can now populate this filesystem with our files so it can act as a root filesystem on our embedded device. We'll create the necessary subdirectories (bin, lib, etc, dev) and copy a variety of files from the MontaVista LSP. These include PowerPC binaries, libraries and various other files from the LSP target subdirectory. We'll also create our necessary device entries using the standard **mknod** utility.

One item to note here is that all binaries and libraries are stripped as they are copied to our mounted filesystem. This reduces their size by removing excess symbols and

debugger information and maximizes our utilization of the available Flash memory.

The script makes liberal use of the **pushd/popd** utility to ensure that all symbolic links are created correctly. I've been bitten by that one way too many times.

After the filesystem has been populated, we create a statistics file called image.map that defines how much space was used for each of the major subdirectories.

Finally, with our filesystem complete, we **umount** it and then compress it. I've used a symbolic link of the gzipped image to my kernel's mbxboot subdirectory to avoid copying it directly.

The next step in the script is the building of the Linux kernel using the **zImage.initrd** rule. This builds a kernel with the bound compressed ramdisk image and sets the expectations for the kernel that the ramdisk will be present for the root filesystem.

The final section of the script binds the Embedded Planet burner program (**burner.srec**) with the new kernel/ramdisk. The burner program is a parasitic utility that, when downloaded to the RPX Net and run, will take the Linux kernel and filesystem and write them into Flash.

Next, we'll talk about getting this image into Flash and the variety of ways to make it happen.

Flashing a Kernel

Downloading the kernel from an external host via TFTP is great in the debugging environment, but few embedded applications exist in an environment where they can bootstrap themselves from another device. The ability to boot from onboard Flash is therefore required.

Of the two flashing methods I've used (absent a BDM programmer), the one presented below is the simplest. The other method utilizes an application called **linuxflash** to burn the image from an NFS mounted filesystem. The problem with this method is a Linux kernel must already be booted on the board for this to work. The former method requires only the onboard EP PlanetCore utilities to perform the flashing and can therefore be done much faster.

The development host must be configured for TFTP (as discussed in the MontaVista CDK), and an IP address must be known for transfer. Listings 2 and 3 are extracted from the minicom session.

Listing 2. Burner Program [2]

Listing 3. Starting the Flashed Image [3]

As shown in Listing 2, once power is applied the PlanetCore bootloader will begin and provide a command prompt. Typing **help** here provides a useful list of available

commands. It's interesting to note that all RPX boards use the same PlanetCore bootloader for operation. Therefore, the same software is used whether executing on an RPX Net or a newer RPX Super (8260).

I use the **t** command to start a TFTP session and provide the parameters for the load. I've stored some of these parameters in the onboard serial EEPROM to simplify debugging (only typing the items that have changed). In this case, I tell the bootloader my image to download is fImage.mot. My Linux development hosts know this file will be located in the /tftpboot directory, as it's identified in the TFTP service line in my inetd.conf.

Pressing Enter for the unchanged parameters (S--record format and 0--offset) starts the download, as shown by the binary number progression. Once the download completes, I tell the bootloader **go**, which executes the fImage.mot application just downloaded. Listing 2 shows the work of the burner program.

So, we've now flashed the RPX Net with our image. We can now start this image by telling the bootloader where to find it (see Listing 3); in this case it's at 0xff840000, as defined within our script. From here, we see a typical Linux boot, with a quick test to show network connectivity.

Extending the Image

So now we can boot a Linux kernel with a ramdisk root filesystem. What's needed is customization for our particular application. Let's have a look at what's required to build a simple application and then update our script so it becomes part of our embedded system.

We'll start with a very simple app that will test our ability to create an application:

```
#include <stdio.h>
main()
{
  printf("Our app works.\n");
}
```

I run an x86 Red Hat development host, but MontaVista's tools allow you to create PowerPC systems on x86 systems (known as cross development). The resulting binary cannot be executed on an x86 host but, when bound with our kernel, can be run on the RPX Net target.

We first set our path to make the cross-development tools visible (opt12 is where mine are):

```
export PATH=/opt12/hardhdat/devkit/ppc/8xx/bin:$PATH
```

Then we use the PowerPC C compiler to build our binary:

```
ppc_8xx-gcc -o testapp testapp.c
```

This binary can now be included in our script by adding the following line in the /bin

section. This assumes we've built the binary at the same level that our **makerd** scripts are run from.

```
cp testapp /mnt/bin/testapp
```

Once we rebuild with our script, burn it into Flash and restart the board, we can test our new app:

```
sh-2.03# testapp
Our app works.
sh-2.03#
```

If we wanted to start our app automatically, it could be easily added to our rc.sysinit script:

```
#!/bin/sh
#
# rc.sysinit
#
>/etc/mtab
mount -a -n
#
ifconfig eth0 192.168.1.15 up
ifconfig eth1 10.0.0.1 up
inetd &
```

If our new binary had required any libraries not present in our current root filesystem, we would also need to migrate these as well. To find out which libraries are used by a particular binary, the following command can be used:

```
ppc_8xx-objdump -p testapp | grep NEEDED
```

In the case of testapp, that would provide:

```
NEEDED      libc.so.6
```

Since our script already collects this library, nothing else is required. Otherwise, the /lib population section of the script would need to be updated for new library stripping and symbolic link setting.

Final Details

The final issue is how to get the board to boot our Linux kernel when power is applied. We again go back to PlanetCore to configure the bootloader on how to behave on boot.

Three EEPROM parameters are used to define our boot behavior. These are defined in the PlanetCore bootloader (see Listing 4).

Listing 4. Setting the autoboot Parameters [4]

First, we set the autoboot key to Confirm, telling the bootloader to wait two seconds before execution. In this way, we can still stop the boot from our terminal session in

the event something goes wrong. Next, we set the format key to Flash to specify where to get our boot image. Finally, we set the start key to the address in Flash where our kernel has been stored (in this case ff840000). Finally, by using the **store** command, we store these in EEPROM so they're present at the next boot.

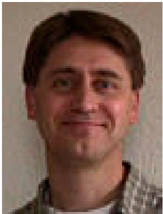
On the next boot, we'll see the following from PlanetCore:

```
        Autoboot in 2 seconds.
        ESC to abort, SPACE or ENTER to go.
loaded at:     FF840000 FF84B1FC
relocated to:  00180000 0018B1FC
board data at: 001801C8 001801E4
relocated to:  00200100 0020011C
zimage at:     FF846000 FF8A64EC
initrd at:     FF8A64EC FF9C23B2
relocated to:  01EE4000 01FFFEC6
avail ram:     00201000 01EE4000
Linux/PC load: root=/dev/ram
```

Summary

Hopefully this article has provided some insight into developing embedded applications with Linux on a PowerPC target. It should also be apparent that working with Linux, whether on an Intel desktop or an embedded PowerPC, is very similar. To me, this is one of the greatest strengths of Linux and why traditional embedded operating system vendors are concerned about the future.

Resources [5]



M. Tim Jones has been developing embedded software since 1986 (1979, if you count T-BUG on the TRS-80 Level I with 4KB). He has designed and developed software ranging from OS kernels for geostationary spacecraft to a variety of terrestrial embedded network applications. Mr. Jones is currently employed as an OS developer with Avaya Communication in Westminster, Colorado. He can be reached at mtj@mtjones.com.

Links

[1] http://www.linuxjournal.com/

[2] http://www.linuxjournal.com//articles/web/2001-07/4624/4624l2.html
[3] http://www.linuxjournal.com//articles/web/2001-07/4624/4624l3.html
[4] http://www.linuxjournal.com//articles/web/2001-07/4624/4624l4.html
[5] http://www.linuxjournal.com//articles/web/2001-07/4624/4624s1.html

Source URL: http://www.linuxjournal.com/article/4624