Build an embedded Linux distro from scratch

Skill Level: Intermediate

Peter Seebach
Freelance author
Plethora.net

12 Aug 2008

Learn how to build a custom Linux® distribution to use in an embedded environment, in this case to drive a Technologic Systems TS-7800 single-board computer. In this tutorial, you learn about cross-compiling, the boot loader, file systems, the root file system, disk images, and the boot process, all with respect to the decisions you make as you're building the system and creating the distribution.

Section 1. Before you start

Objectives

This tutorial shows you how to install Linux on a target system. Not a prebuilt Linux distribution, but your own, built from scratch. While the details of the procedure necessarily vary from one target to another, the same general principles apply.

The result of this tutorial (if you have a suitable target) is a functional Linux system you can get a shell prompt on.

About this tutorial

The tutorial begins with a discussion of cross-compilation issues, then discusses what the components of a Linux system are and how they are put together. Both the building and the installation and configuration of the target system are covered.

The specific target discussed, a Technologic Systems TS-7800, imposes its own default boot and bring-up behaviors; other systems will have other mechanics, and this tutorial does not go into great detail about every possible boot loader.

Prerequisites and system requirements

Developers who are interested in targeting embedded systems, or who just want to learn more about what Linux systems are like under the hood, will get the most out of this tutorial.

The host environment used is Ubuntu, but other systems work as well. Users are assumed to have basic familiarity with UNIX® or Linux system administration issues. The tutorial assumes root access to a host system.

This tutorial assumes that your shell is a Bourne shell derivative; if you use a C shell derivative, the prompt will probably look different, and you will need to use different commands to set environment variables.

For cross-compiling (which is useful when targeting embedded systems), I used crosstool-ng version 1.1.0, released in May of 2008. You may download it from the distribution site (see Resources). Details follow on installing and configuring it.

Section 2. About the target and architecture

Target

The target I chose is a Technologic Systems TS-7800 (see Resources for more detail). This is a small embedded ARM system, with both built-in and removable flash storage, as well as a SATA controller. This tutorial walks you through configuring this system to boot to a login prompt, without relying on prebuilt binaries.

Architecture

I chose the ARM architecture to make it a little easier to check whether a given binary is host or target, and to make it easy to see when host pollution might be occurring. It is also nice having a machine that consumes a total of about 5W of power and runs completely silently.

Section 3. Cross-compilation

What is cross-compiling?

Cross-compiling is using a compiler on one system to develop code to run on another. Cross-compilation is relatively rare among casual UNIX users, as the default is to have a compiler installed on the system for use on that system. However, cross-compilation becomes quite common (and relevant) when targeting embedded systems. Even when host and target are the same architecture, it is necessary to distinguish between their compilers; they may have different versions of libraries, or libraries built with different compiler options, so that something compiled with the host compiler could fail to run, or could behave unexpectedly, on the target.

Obtaining cross-compilation tools

It is, in theory, quite possible to build a cross-compiler yourself, but it is rarely practical. The series of bootstrap stages needed can be difficult and time-consuming, and it is often necessary to build a very minimal compiler, which is used to partially configure and build libraries, the headers of which are then used to rebuild the compiler so it can use them, and so on. A number of commercial sources for working cross-compilers for various architecture combinations are available, as well as several free cross-compilation toolkits.

Introducing crosstool-ng

Dan Kegel's crosstool (see Resources for details) collects a variety of expertise and a few specialized patches to automatically build toolchains for a number of systems. Crosstool has not been updated in a while, but the new crosstool-ng project builds on this work. For this tutorial, I used crosstool-ng version 1.1.0, released in May of 2008. Download it from the distribution site (see Resources).

Installing crosstool-ng

Crosstool-ng has a `configure` script. To configure it, just run the script using `--prefix` to set a location. For instance:

```
$ ./configure --prefix=$HOME/7800/ctng
```

Once you have configured it, build it using `make` and then `make install`. The build process creates a `ctng` directory in the 7800 working directory that holds the crosstool-ng build scripts. Add the `ctng/bin` subdirectory to your path:

```
$ PATH=$PATH:$HOME/7800/ctng/bin
```

Configuring crosstool-ng

Crosstool-ng uses a `.config` file similar to those used by the Linux kernel. You need to create a configuration file matching your target to use crosstool-ng. Make a working directory for a crosstool-ng build:

```
$ mkdir toolchain-build
$ cd toolchain-build
```

Now, copy in a default configuration. It's possible to manually configure crosstool-ng, but one of the sample configurations happens to fit the target perfectly:

```
$ cp
../ctng/lib/ct-ng-1.1.0/samples/arm-unknown-linux-uclibc/* .
```

Finally, rename the `crosstool.config` file:

```
$ mv crosstool.config .config
```

This copies in a configuration file that targets an armv5te processor, the model used on the TS-7800. It builds with uClibc, a libc variant intended for embedded systems. However, the configuration file does need one modification.

Fixing the configuration path

The default target directory for a crosstool-ng build is `$HOME/x-tools/$TARGET`. For instance, on this build, it would come out as `x-tools/arm-unknown-linux-uclibc`. This is very useful if you are building for a lot of targets, but not so useful if you are building for only one. Edit the `.config` file and change `CT_PREFIX_DIR` to `${HOME}/7800/toolchain`.

Building the toolchain

To build the toolchain, run the `ct-ng` script with the `build` argument. To improve performance, especially on a multi-core system, you may want to run with multiple jobs, specified as `build.#`. For example, this command builds with four jobs:

```
$ ct-ng build.4
```

This may take quite a while, depending on your host system. When it's done, the toolchain is installed in `$HOME/7800/toolchain`. The directory and its contents are marked read-only; if you need to delete or move them, use `chmod u+w` The `ct-ng` script takes other arguments, such as `help`. Note that `ct-ng` is a script for the standard `make` utility, and as a result, the output from `--help` is just the standard `make` help; use `ct-ng help` to get the help for crosstool-ng.

If you haven't seen this trick before, it's a neat one. Modern UNIX systems interpret an executable file in which the first line starts with `#!` as a script, specifically, a script for the program named on the rest of the line. For instance, many shell scripts start with `#!/bin/sh`. The name of the file is passed to the program. For programs that treat their first argument as a script to run, this is sufficient. While `make` does not do that automatically, you can give it a file to run with using the `-f` flag. The first line of `ct-ng` is `#!/usr/bin/make -rf`. The `-r` flag suppresses the built-in default construction rules of `make`, and the `-f` flag tells it that the following word (which is the script's file name) is the name of a file to use instead of one named `Makefile`. The result is an executable script that uses `make` syntax instead of shell syntax.

Using the toolchain

For starters, add the directory containing the compiler to your path:

```
$ PATH=~/7800/toolchain/bin:$PATH
```

With that in your path, you can now compile programs:

```
$ arm-unknown-linux-uclibc-gcc -o hello hello.c $ file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for
GNU/Linux 2.4.17, dynamically linked (uses shared libs), not
stripped
```

Where are the libraries?

The libraries used by the toolchain to link binaries are stored in `arm-unknown-linux-uclibc/sys-root`, under the `toolchain` directory. This directory forms the basis of an eventual root file system, a topic we'll cover under Filesystems, once the kernel is built.

Kernel setup

The kernel distribution tree provided by the vendor is already configured for cross compilation. In the simplest case (which this is), the only thing you have to do to cross compile a Linux kernel is to set the `CROSS_COMPILE` variable in the top-level Makefile. This is a prefix that is prepended to the names of the various programs (gcc, as, ld) used during the build. For instance, if you set `CROSS_COMPILE` to `arm-`, the compile will try to find a program named `arm-gcc` in your path. For this build, then, the correct value is `arm-unknown-linux-uclibc`. Or, if you don't want to rely on path settings, you can specify the whole path, as in this example:

```
CROSS_COMPILE ?=
$(HOME)/7800/toolchain/bin/arm-unknown-linux-uclibc-
```

Section 4. Building the kernel

Downloading the source

Download Technologic's Linux source and TS-7800 configuration files and unzip them in a suitable location.

Kernel configuration

A complete discussion of kernel configuration is beyond the scope of this tutorial. In this case, the `ts7800_defconfig` target gave me a default usable configuration for the 7800, with one small hiccup: the `CONFIG_DMA_ENGINE` setting ended up off when it should have been on.

Tweaking the kernel

It is usually best to edit the kernel using `make menuconfig`, which offers a semi-graphical interface to kernel configuration. This interface is navigated using arrow keys to move the cursor, the **Tab** key to select options from the bottom of the screen, and the space or **Enter** keys to select options. For instance, to exit without changing anything, press **Tab** until the <Exit> at the bottom of the screen is highlighted, then press **Enter**. Running `make menuconfig` again reopens the editor.

Changing the default console

The TS-7800 normally boots silently, because the default kernel configuration specifies a null console device to keep the display quiet. To change this, use the arrow keys to navigate down to "Boot options," and press **Enter**. The third line shows the default kernel options, which select the ramdisk, the startup script, and the console. Use the arrow keys to navigate down to this line, press **Enter**, and change `console none` to `console ttyS0,115200`. Then, press **Tab** to move the cursor to the <Ok> at the bottom of the panel, and press **Enter**. Now press **Tab** to select <Exit> and press **Enter**, bringing you back to the main menu.

For the goal of booting as fast as possible, the console device isn't useful, and indeed, even at a high baud rate, sending kernel messages can take a noticeable fraction of the time the system takes to boot. For debugging and playing around, though, you want the console.

Enabling the DMA engine

Navigate down to "Device drivers" and press **Enter**. This list is longer than the usual display, so you will have to scroll down to the very end to reach the option for "DMA Engines." Navigate to that with the arrow keys, and press **Enter**. There are two options at the top of this page that have square brackets indicating a boolean option. The second, "Support for DMA engines," was not enabled by default in the download I started with. Navigate to it with the arrow keys, and press space to toggle its state. Now use **Tab** and **Enter** to select <Exit> from each screen to navigate out to the top level of the program, and then <Exit> one more time to leave the program. When asked whether you wish to save your new kernel configuration, tab to <Yes> and press **Enter**.

Compiling the kernel

Type `make`. Yes, it really is that simple. This builds a kernel, as well as a collection of modules. Once again, multi-core users might want multiple jobs; try `make -j 5`. For the purposes of this project, I'm going to ignore kernel modules, and favor compiling-in any needed features. The bootstrap ramdisk technique used to get needed drivers into the kernel early seems excessive, and building a root file system is already complicated enough. This, of course, brings up the question of how to get a kernel booting, the subject of the next section.

Section 5. Boot loaders

What is a boot loader?

A boot loader (or bootstrap loader) is a program that loads another program. The boot loader is a small and specialized hunk of code, specific to a target system, that has just enough complexity to find the kernel and load it without being a full-featured kernel. Different systems use a variety of different boot loaders, from the huge and complicated BIOS programs common on desktop PCs, to very small and simple programs more common on embedded systems.

A simple boot loader

The TS-7800 uses an unusually simple boot loader, which simply picks up the kernel from a predetermined partition of the SD card or on-board flash. A jumper determines whether the board looks first at the on-board flash, or at the SD cards. There are no other configuration settings. More complicated boot loaders (such as `grub`, commonly used on desktop PCs) have options for configuring kernel options and so on at boot time. On this system, the kernel's default options must be compiled in, as described previously.

The decision to compile in kernel options is a typical example of a choice that makes some sense in an embedded system, but would be uncomfortably limiting on a desktop.

Kernel formats

There are a variety of formats in which kernels are stored. The initial Linux kernel binary, named `vmlinux`, is rarely the file that a boot loader will work with. On the TS-7800, the boot loader can use two files, either `Image` (uncompressed) or `zImage` (compressed). These files are created in the `arch/arm/boot` directory within the kernel tree.

People often describe the `zImage` kernel as a compressed kernel, and so expect the boot loader to need to provide decompression. In fact, it's rather more clever. The `zImage` kernel is an uncompressed executable, which contains a particularly large static data object which is a compressed kernel image. When the boot loader loads and runs the `zImage` executable, that executable then unpacks the kernel image and executes it. This way, you get most of the benefit of compression without imposing additional effort on the boot loader.

Setting up the SD card

The SD card needs an MBR table containing DOS-style partition information. A sample MBR table is available for download from Technologic's site; this is for a 512MB card, but it is easy to edit the fourth partition to a size suiting whatever size card you want to use. The first three partitions are 4MB each; the first is unused on smaller cards, and the second and third hold the kernel and initial ramdisk image respectively.

For my purposes, I used the `sfdisk` utility, which is not always recommended but has the desirable trait of trusting me when I ask it to create a partition that does not align on a partition boundary.

Installing the initial kernel

The kernel for the TS-7800 is dumped directly into the second partition of the SD card. The exact path to the card depends on how you are accessing it; typically, if you have the card on a USB card reader, it will be detected as a SCSI device. Note that there is no file system access involved; the raw kernel is just dumped into the partition. The `dd` command copies raw data from one source to another, as in this example:

```
$ dd if=zImage of=/dev/sdd2 bs=16k
93+1 records in
93+1 records out
1536688 bytes (1.5 MB) copied, 0.847047 s, 1.8 MB/s
```

This command dumps raw data from the `zImage` file to the second partition of `/dev/sdd`, using 16KB blocks.

A bit about block sizes

The output of this command is a little cryptic (as are its inputs, honestly). When `dd` runs, it copies data in "records," which are by default 512-byte blocks; this command specifies a block size of 16k (which `dd` understands to mean 16*1024).

The `dd` command reports the amount of data copied first in blocks; the number after the plus sign is the number of partial blocks copied. In this case, because 1,536,688 is not an exact multiple of the block size, the remaining bytes of the file are read (and written) separately as a partial block. This information is harmless for most modern devices but crucial to diagnosing problems with some older ones.

The ability to control block sizes (and reblock data when transferring it) was exceptionally useful for working with tape devices and other specialized media that required writes to be of particular fixed sizes, and also helps for performance and reliability reasons with flash devices.

While the kernel device representing flash media can often take writes of arbitrary sizes, it is common for the underlying device to work only in full blocks, often of somewhat larger sizes (4KB or larger). To do a partial write, the flash device must extract the current contents of the full block, modify them with the input data, and flash the whole block back. If a device uses 4KB blocks, and you write to it in 512-byte blocks, each device block gets rewritten eight times for a single copy. This is bad for the device's longevity, and also bad for performance. (A 512-byte write of the same file was half as fast on the flash card I used.)

Booting the kernel

Booting the kernel is simple enough. Set the jumper for an SD boot, put the card in the system, and power it up. If you started with a blank card, this produces a predictable result:

```
Kernel panic - not syncing: VFS: Unable to mount root fs on
unknown-block(1,0)
```

This cryptic message indicates that the kernel has been unable to find its root filesystem. That means it's about time to create one.

Section 6. Filesystems

Root filesystems

Normally, Linux has only a single root filesystem. However, during boot, it is common to use a temporary root filesystem to load and configure device drivers and the like, and then obtain the real root filesystem and switch to it. The TS-7800 uses this temporary filesystem (called an initial ramdisk, or initrd) for the SD driver and for a variety of configuration options. In fact, the ramdisk is used for many of the things (such as determining the final root filesystem to use) that might be handled by the boot loader on another system.

Manipulating filesystems

A general caveat applies: most filesystem operations require root privileges. There are some exceptions and some workarounds. The most notable is the `fakeroot` utility/library, which allows you to build applications with "fakeroot support." This allows the manipulation of a virtual root filesystem with control over ownership, permissions, and so on. However, the permissions and related material are maintained in a separate database, and no actual privileged operations are needed. This allows a non-root user to manipulate directory hierarchies with virtual root privileges, and ultimately create archives or filesystem images containing files that the user hasn't got the privileges to create. For this tutorial, however, I assume root privileges.

Changing root

The `chroot` command, while certainly part of the story, is not sufficient to change the root filesystem. The tool for this is `pivot_root`, which makes one named directory the new root directory, and "moves" (really, changes the mount point of) the old root directory to another name.

A not-so-initial root

For the purposes of this tutorial, I'm assuming that the default initrd image provided with the board is adequate. Linux kernel distributions have some support for generating suitable initrd images, and the details of this one are not crucial to an understanding of building an actual distribution. I'll just introduce the key things needed to get the system moving, then focus on what goes on the real root filesystem.

Filesystem images

Most Linux users are familiar with files containing images of ISO CD-ROM or DVD filesystems. Linux supports the creation and use of images of other types of filesystems. In fact, any file can be treated as though it were a disk partition, using the `-o loop` option to `mount`. For instance, while researching this, I made a copy of the initrd image used by the TS-7800:

```
$ dd if=/dev/sdd3 of=initrd.dist bs=16k
$ cp initrd.dist initrd.local
```

With this file available, it's possible to mount the filesystem to have a look at it. The copy named `initrd.local` is the local version to edit; the copy named `initrd.dist` is a safe pristine copy in case something gets broken later.

```
$ mount -o loop initrd.local /mnt
```

What does the initrd do?

The initrd provides a very basic initial filesystem used to bootstrap the device by finding and mounting the real root filesystem. The default kernel configuration is hard-wired to use an initrd (`/dev/ram0`) as root, and run the `linuxrc` script on startup. There are several provided variants of this script for different root file systems. The obvious choice is to look at the `linuxrc-sdroot` variant.

Preparing initrd

The `linuxrc-sdroot` program tries to configure the system to use an SD card as root. Assuming a correctly configured SD card with an ext3 filesystem on the fourth fdisk partition, the script mounts that filesystem and runs `/sbin/init` on it. This looks like a good strategy. So, a couple of small changes are needed to the initrd image. The first is to change the `linuxrc` symlink to point to `linuxrc-sdroot`. The second is to update the script to work with the uClibc build instead of the Debian build it was configured for.

The SD driver

The SD driver for the TS-7800 includes a proprietary module, so the `tssdcard.ko` file simply has to be manually copied in, and loaded at runtime. The file on the distribution initrd works fine with the new kernel, so no change is needed.

Switching bootstrap routines

The kernel simply runs the program called `linuxrc`. By default, this is a symbolic link to the program `linuxrc-fastboot`, which comes up quickly to the initrd root filesystem. Removing the link, and linking `linuxrc` instead to `linuxrc-sdroot`, causes the system to try to boot from the last partition on the SD once the drivers are loaded.

Changing the bootstrap routine

The script checks for errors, and if it finds any, mounts from the internal flash. If it does not, it mounts the filesystem from the SD card and then tries to change to it. The bootstrap routine actually has a small quirk that affects this build; it tries to use its own `mount` utility, rather than the rootfs one, for the last cleanup work after calling `pivot_root`. This works well with the provided Debian install, but fails with the minimalist uClibc install. The solution is to change the lines following the `pivot_root` command at the end of the file (after the `else` clause) as follows:

```
pivot_root . ./initrd
/bin/mount -n --move ./initrd/dev ./dev
```

```
/bin/mount -n --move ./initrd/sys ./sys
/bin/mount -n --move ./initrd/proc ./proc
exec /sbin/init < $CONSOLE > $CONSOLE 2>&1
```

This causes the script to run the new (uClibc, statically linked) executables that you are about to create, instead of trying to run dynamically linked executables from another system. (If you were doing this without a tutorial, you probably wouldn't know this until after you'd set everything else up and started getting cryptic error messages).

Unmount the disk image

Unmount the initrd disk image like so:

```
$ umount /mnt
```

Section 7. Populating the root file system

What goes into a root filesystem?

The root filesystem needs an `/sbin/init` that will do whatever it is we want done. This can be a custom program rather than a conventional UNIX-style init, but it is easier to work with the system if you provide a working environment complete with console shell and everything.

Basic libraries and headers

The `crosstool-ng` build created a default system root providing library code and headers. Make a copy of it:

```
$ cp -r toolchain/arm-unknown-linux-uclibc/sys-root rootfs
```

Note that no effort is being made to fix permissions right now; for a real distribution, you'd probably want to correct this, but it's harmless enough for demonstration purposes. This gets you the first basic parts of a root filesystem, containing only the libraries (and headers) future builds will use.

Busybox

When it comes to embedded root filesystems, busybox is probably the best starting point. Busybox offers a reasonably complete set of core system utilities all built into a single binary, which uses the name it gets invoked with to determine what function to perform. This, coupled with a lot of links, allows dozens of the main system programs to be bundled into small amount of space. This isn't always necessary, but it is convenient for our purposes.

Downloading busybox

Busybox can be downloaded from its downloads page (see Resources). The archive is just a standard tarball and can be unpacked in a build directory. I used version 1.10.2.

Configuring busybox

Busybox is built around similar configuration tools to those used for `crosstool-ng` and the Linux kernel. To build a default configuration, run `make defconfig`. This creates the default configuration. A following `make menuconfig` allows you to change settings; under "Busybox Settings," the "Build Options" menu allows you to specify a static build. Do that, because the default `crosstool-ng` build of uClibc requires it. You might sometimes prefer a dynamically linked binary, to reduce the size of the executable, but the size problem isn't such a big deal on a system where nearly every binary is just a symbolic link. As with the kernel, the top-level `Makefile` has a `CROSS_COMPILE` variable which holds the prefix to use on the names of compiler tools. Set it to the same value you used before.

Building busybox

To build busybox, just run `make`. Surprisingly, there was a compilation problem during the build, which required me to add `<linux/types.h>` to the main `libbb.h` header. With that out of the way, the compile completed quite quickly.

Installing busybox

To install busybox, you need to run `make install`, but specify the root filesystem path:

```
$ make CONFIG_PREFIX=$HOME/7800/rootfs install
```

This copies in the `busybox` binary and creates all of the necessary links. This root filesystem now has libraries and all the common UNIX utilities. What's missing?

Device nodes

You need device nodes. There are two ways to provide device nodes for a root file system. One is to statically create all the device nodes you plan to have, the other is to use something like `udev`, which provides an automatic current device node tree for the current kernel. While `udev` is extremely elegant, it can also be fairly slow, and embedded systems have the luxury of being able to predict their hardware components.

Creating the needed device nodes

Many programs depend on access to device nodes, such as `/dev/console` or `/dev/null`. Device nodes are typically made at runtime on ARM systems, using the mdev utility, part of busybox. In fact, this has already been done by the initrd filesystem; one of the changes above was to move the already populated `/dev` mount point onto the new root filesystem. You can manually create key device nodes, but it is usually not worth the trouble; `mdev` is smarter.

Mount points

Several mount points are required. The `/sys` and `/proc` mount points are used to move the sysfs and procfs virtual filesystems. The `/dev` directory is used as a mount point for a tmpfs filesystem, populated by `mdev`. These directories do not need any other contents:

```
$ cd $HOME/7800/rootfs
$ mkdir sys proc dev
```

Etcetera

The /etc directory, while not strictly necessary to get to a shell prompt, is very useful to get to a useful shell prompt. The most important part during initial booting is the `init.d` subdirectory, which `init` searches for system startup files. Create a trivial `rcS` script and give it execute permissions for initial system startup:

```
$ mkdir rootfs/etc/init.d
```

Create the `rootfs/etc/init.d/rcS` file with the following contents:

```
#!/bin/sh
echo "Hello, world!"
```

Now mark it executable:

```
$ chmod 755 rootfs/etc/init.d/rcS
```

Section 8. Creating a disk image

Make a large empty file

You can create a file system image by copying blocks from `/dev/zero` into a file, or just by copying an existing image. Whatever size space you have available (remembering that 12MB are already reserved for the first three partitions), you can create a file of that size with `dd`. This creates a nice roomy 64MB root filesystem:

```
$ dd if=/dev/zero of=rootfs.local bs=1M count=64
```

Format the file

The various `mkfs` utilities can be run on files, not just on disks. To build an ext3 root filesystem (the kind the initrd looks for), run `mkfs.ext3` on your disk image:

```
$ mkfs.ext3 rootfs.local
```

The `mkfs` utility prompts you as to whether or not to proceed even though the file is not a block special device:

```
rootfs.tmp is not a block special device.
Proceed anyway? (y,n) y
```

Mount the new disk image

To mount the disk image, use the `-o loop` option to the `mount` command again to mount the image somewhere.

```
$ mount -o loop rootfs.local /mnt
```

Copy the rootfs files to the mounted disk

The `pax` utility is particularly good at this.

```
$ cd 7800/rootfs
$ pax -r -w -p e . /mnt
```

This copies the directory tree you've just created to `/mnt`, preserving symlinks and special files (except for sockets, but you can live without the log socket).

Unmount the image

Once again, having finished work on the image, unmount it:

```
$ umount /mnt
```

Section 9. Getting the images back on the card

Copying files back

You might wonder why the above procedure didn't just use the card as a filesystem. The answer is that, in general, it is better not to use flash filesystems very heavily, because flash devices have much shorter life expectancies than hard drive media under heavy load. Thus, the complicated edits and copies are done to disk images, then the disk images are copied back as a single write operation.

Copy in the initrd

Copy the initrd image in to the third partition of the SD card:

```
$ dd if=initrd.local of=/dev/sdd3 bs=16k
```

Copy in the root filesystem

Likewise, copy the filesystem image in to the fourth partition of the SD card.

```
$ dd if=rootfs.local of=/dev/sdd4 bs=16k
```

Wait for device access

Do not yank the SD card out of the reader right away. Sometimes it can take a few extra seconds for the system to finish writing. Wait until any activity lights have stabilized, then wait a few seconds longer just to be sure.

Section 10. The boot process reviewed

What does init do?

The `init` program manages system startup, and then keeps the system running. For instance, when processes die, `init` is the program that collects their exit status values so the kernel can finish unloading them. The exact startup procedure varies from one version of Linux to another. In this case, it is the busybox version of `init` that will be booting our test system.

You may have wondered why the previous startup script uses `exec /sbin/init` rather than just invoking `init`. The reason is that the startup script was the previous `init`, and had the special process ID 1. The `init` program does not become the system's startup daemon unless its process ID is 1; `exec` causes it to take over the calling shell's process ID, rather than acquiring a new one.

Initial startup

The init program starts by running the first system startup script, `/etc/init.d/rcS`. (If there is no such file, it prints a warning message and continues without it.) After this, it runs according to the instructions in `/etc/inittab`, if it exists. In the absence of an `/etc/inittab`, busybox `init` runs a shell on the console, and deals gracefully with reboot and halt requests.

Bringing it together

With your own kernel, a slightly customized initrd, and your own root filesystem, you should now be able to boot the system to a shell prompt. If you created the `rcS` file, the system should greet you on boot. Now you have a working, if rather minimal, Linux system, built entirely from source, and assembled by hand.

What you do next is up to you. I suggest installing some video games, but you could always configure the system for a database server or a Web server. If you plan to do anything that will produce a lot of disk activity, consider using an external hard drive attached via USB.

Resources

Learn

- Check out this introduction to the Linux boot process (developerWorks, May 2006).
- Learn more about Dan Kegel's crosstool on its original site.
- Read about Linux initial RAM disks and how to use them (developerWorks, July 2006).
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Get the complete Linux source and configuration files for the TS-7800 from Technologic's FTP server.
- Download the crosstool-ng sources and documentation to build your own cross compiler.
- Download the Busybox source code to get a complete Linux system in a tiny little package.
- Learn more about the Technologic Systems TS-7800 single-board computer.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and spaces.

About the author

Peter Seebach
Peter Seebach has been collecting video game consoles for years, but has only been running Linux on them recently. He is still not sure whether this is a Linux machine that plays video games, or a game machine that runs Linux.

Trademarks