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 [Related topics](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html" \l "artrelatedtopics)). Details follow on [installing and configuring it](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html).

About the target and architecture

Target

The target I chose is a Technologic Systems TS-7800 (see [Related topics](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html" \l "artrelatedtopics) 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.

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 [Related topics](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html" \l "artrelatedtopics) 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 [Related topics](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html" \l "artrelatedtopics)).

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](https://www.ibm.com/developerworks/linux/tutorials/l-embedded-distro/index.html), 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-

Building the kernel

Downloading the source

Download Technologic's [Linux source and TS-7800 configuration files](ftp://ftp.embeddedarm.com/ts-arm-sbc/ts-7800-linux/sources/) 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.