Name	Main architectures supported
Das U-Boot	ARC, ARM, Blackfin, Microblaze, MIPS, Nios2, OpenRiec, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, Nios2, PowerPC
GRUB 2	X86, X86_64
Little Kernel	ARM
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS

We are going to focus on U-Boot because it supports a good number of processor architectures and a large number of individual boards and devices. It has been around for a long time and has a good community for support.

It may be that you received a bootloader along with your SoC or board. As always, take a good look at what you have and ask questions about where you can get the source code from, what the update policy is, how they will support you if you want to make changes, and so on. You may want to consider abandoning the vendor-supplied loader and using the current version of an open source bootloader instead.

U-Boot

U-Boot, or to give its full name, **Das U-Boot**, began life as an open source bootloader for embedded PowerPC boards. Then, it was ported to ARM-based boards and later to other architectures, including MIPS and SH. It is hosted and maintained by Denx Software Engineering. There is plenty of information available, and a good place to start is `http://www.denx.de/wiki/U-Boot`. There is also a mailing list at `u-boot@lists.denx.de`.

Building U-Boot

Begin by getting the source code. As with most projects, the recommended way is to clone the `.git` archive and check out the tag you intend to use, which, in this case, is the version that was current at the time of writing:

```
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
$ git checkout v2017.01
```

Alternatively, you can get a tarball from `ftp://ftp.denx.de/pub/u-boot`.

There are more than 1,000 configuration files for common development boards and devices in the `configs/` directory. In most cases, you can make a good guess of which to use, based on the filename, but you can get more detailed information by looking through the per-board README files in the `board/` directory, or you can find information in an appropriate web tutorial or forum.

Taking the BeagleBone Black as an example, we find that there is a likely configuration file named `configs/am335x_boneblack_defconfig` and we find the text `The binary produced by this board supports` … Beaglebone Black in the board README files for the `am335x` chip, `board/ti/am335x/README`. With this knowledge, building U-Boot for a BeagleBone Black is simple. You need to inform U-Boot of the prefix for your cross compiler by setting the make variable `CROSS_COMPILE`, and then selecting the configuration file using a command of the type `make [board]_defconfig` . Therefore, to build U-Boot using the Crosstool-NG compiler we created in `Chapter 2`, *Learning About Toolchains*, you would type:

```
$ source MELP/chapter_02/set-path-arm-cortex_a8-linux-gnueabihf
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
am335x_boneblack_defconfig
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

The results of the compilation are:

- `u-boot`: U-Boot in ELF object format, suitable for use with a debugger
- `u-boot.map`: The symbol table
- `u-boot.bin`: U-Boot in raw binary format, suitable for running on your device
- `u-boot.img`: This is `u-boot.bin` with a U-Boot header added, suitable for uploading to a running copy of U-Boot
- `u-boot.srec`: U-Boot in Motorola S-record (**SRECORD** or **SRE**) format, suitable for transferring over a serial connection

The BeagleBone Black also requires a **secondary program loader** (**SPL**), as described earlier. This is built at the same time and is named `MLO`:

```
$ ls -l MLO u-boot*
-rw-rw-r-- 1 chris chris 78416 Mar 9 10:13 u-boot/MLO
-rwxrwxr-x 1 chris chris 2943940 Mar 9 10:13 u-boot/u-boot
-rwxrwxr-x 1 chris chris 368348 Mar 9 10:13 u-boot/u-boot.bin
-rw-rw-r-- 1 chris chris 368412 Mar 9 10:13 u-boot/u-boot.img
-rw-rw-r-- 1 chris chris 520741 Mar 9 10:13 u-boot/u-boot.map
-rwxrwxr-x 1 chris chris 1105162 Mar 9 10:13 u-boot/u-boot.srec
```

The procedure is similar for other targets.

Installing U-Boot

Installing a bootloader on a board for the first time requires some outside assistance. If the board has a hardware debug interface, such as JTAG, it is usually possible to load a copy of U-Boot directly into RAM and set it running. From that point, you can use U-Boot commands to copy itself into flash memory. The details of this are very board-specific and outside the scope of this book.

Many SoC designs have a boot ROM built in, which can be used to read boot code from various external sources, such as SD cards, serial interfaces, or USB mass storage. This is the case with the `am335x` chip in the BeagleBone Black, which makes it easy to try out new software.

You will need an SD card reader to write the images to a card. There are two types: external readers that plug into a USB port, and the internal SD readers that are present on many laptops. A device name is assigned by Linux when a card is plugged into the reader. The command `lsblk` is a useful tool to find out which device has been allocated. For example, this is what I see when I plug a nominal 8 GB microSD card into my card reader:

```
$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 477G 0 disk
├─sda1 8:1 0 500M 0 part /boot/efi
├─sda2 8:2 0 40M 0 part
├─sda3 8:3 0 3G 0 part
├─sda4 8:4 0 457.6G 0 part /
└─sda5 8:5 0 15.8G 0 part [SWAP]
sdb 8:16 1 7.2G 0 disk
└─sdb1 8:17 1 7.2G 0 part /media/chris/101F-5626
```

In this case, `sda` is my 512 GB hard drive and `sdb` is the microSD card. It has a single partition, `sdb1`, which is mounted as directory `/media/chris/101F-5626`.

Although the microSD card had 8 GB printed on the outside, it was only 7.2 GB on the inside. In part, this is because of the different units used. The advertised capacity is measured in Gigabytes, 10^9, but the sizes reported by software are in Gibibytes, 2^{30}. Gigabytes are abbreviated GB, Gibibytes as GiB. The same applies for KB and KiB, and MB and MiB. In this book, I have tried to use the right units. In the case of the SD card, it so happens that 8 Gigabytes is approximately 7.4 Gibibytes. The remaining discrepancy is because flash memory always has to reserve some space for bad block handling. This is a topic that I will return to in `Chapter 7`, *Creating a Storage Strategy*.

If I use the built-in SD card slot, I see this:

```
$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 477G 0 disk
├─sda1 8:1 0 500M 0 part /boot/efi
├─sda2 8:2 0 40M 0 part
├─sda3 8:3 0 3G 0 part
├─sda4 8:4 0 457.6G 0 part /
└─sda5 8:5 0 15.8G 0 part [SWAP]
mmcblk0 179:0 0 7.2G 0 disk
└─mmcblk0p1 179:1 0 7.2G 0 part /media/chris/101F-5626
```

In this case, the micro SD card appears as `mmcblk0` and the partition is `mmcblk0p1`. Note that the microSD card you use may have been formatted differently to this one and so you may see a different number of partitions with different mount points. When formatting an SD card, it is very important to be sure of its device name. You really don't want to mistake your hard drive for an SD card and format that instead. This has happened to me more than once. So, I have provided a shell script in the book's code archive named `MELP/format-sdcard.sh`, which has a reasonable number of checks to prevent you (and me) from using the wrong device name. The parameter is the device name of the microSD card, which would be `sdb` in the first example and `mmcblk0` in the second. Here is an example of its use:

```
$ MELP/format-sdcard.sh mmcblk0
```

The script creates two partitions: the first is 64 MiB, formatted as `FAT32`, and will contain the bootloader, and the second is 1 GiB, formatted as `ext4`, which you will use in `Chapter 5`, *Building a Root Filesystem*.

After you have formatted the microSD card, remove it from the card reader and then re-insert it so that the partitions are auto mounted. On current versions of Ubuntu, the two partitions should be mounted as `/media/[user]/boot` and `/media/[user]/rootfs`. Now you can copy the SPL and U-Boot to it like this:

```
$ cp MLO u-boot.img /media/chris/boot
```

Finally, unmount it:

```
$ sudo umount /media/chris/boot
```

Now, with no power on the BeagleBone board, insert the micro-SD card into the reader. Plug in the serial cable. A serial port should appear on your PC as `/dev/ttyUSB0`. Start a suitable terminal program, such as `gtkterm`, `minicom`, or `picocom`, and attach to the port at `115200` bps (bits per second) with no flow control. `gtkterm` is probably the easiest to setup and use:

```
$ gtkterm -p /dev/ttyUSB0 -s 115200
```

Press and hold the Boot Switch button on the Beaglebone Black, power up the board using the external 5V power connector, and release the button after about 5 seconds. You should see a U-Boot prompt on the serial console:

```
U-Boot#
```

Using U-Boot

In this section, I will describe some of the common tasks that you can use U-Boot to perform.

Usually, U-Boot offers a command-line interface over a serial port. It gives a Command Prompt which is customized for each board. In the examples, I will use `U-Boot#`. Typing `help` prints out all the commands configured in this version of U-Boot; typing `help <command>` prints out more information about a particular command.

The default command interpreter for the BeagleBone Black is quite simple. You cannot do command-line editing by pressing cursor left or right keys; there is no command completion by pressing the *Tab* key; and there is no command history by pressing the cursor up key. Pressing any of these keys will disrupt the command you are currently trying to type, and you will have to type *Ctrl + C* and start over again. The only line editing key you can safely use is the backspace. As an option, you can configure a different command shell called **Hush**, which has more sophisticated interactive support, including command-line editing.

The default number format is hexadecimal. Consider the following command as an example:

```
nand read 82000000 400000 200000
```

This will read `0x200000` bytes from offset `0x400000` from the start of the NAND flash memory into RAM address `0x82000000`.