The LinuxBIOS project

Putting Linux on your motherboard

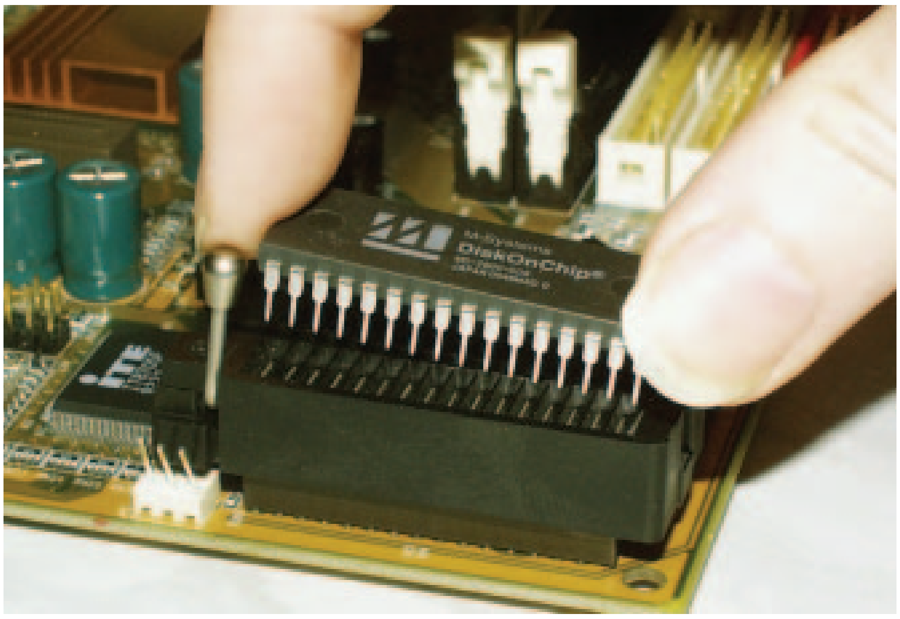
If you haven't come across the LinuxBIOS project [1] yet, you may be amazed at what it sets out to do. BY ANTONY STONE

LinuxBIOS releases yet another part of your PC to Open Source software – in this case, the BIOS chip itself. BIOS stands for Basic Input Output System, and the BIOS chip is installed on the motherboard by the manufacturer, and which most users, no matter which Operating System they run on their computer, hardly ever think about. They only ever see the BIOS screen when the machine is first booting up, and it is usually taken for granted as simply another piece of the hardware, which hardly anyone ever considers the idea of changing.

The code inside the BIOS chip (which is simply a non-volatile memory device, so that the software is available immediately the computer is powered on) is responsible for starting up the machine, checking for the presence of hardware such as memory and disk drives, and then initialising them so that the real operating system can start booting. Without the BIOS, your computer would do absolutely nothing when it was turned on, because the BIOS contains the very first instructions which the CPU executes in order to start everything working.

The LinuxBIOS project replaces the normal BIOS code on your motherboard with the Linux kernel itself, so that your machine boots instantly into Linux within seconds of turning it on.

LinuxBIOS has more advantages than simply very fast boot times, however. LinuxBIOS has been mainly developed for cluster systems, because it allows far greater remote management and configuration than a standard BIOS chip does. If you have lots of servers configured in a cluster, and you need to change a (normal) BIOS setting, then going around connecting a screen and keyboard to each machine, rebooting and making a manual change can be tedious and inconvenient, to say the least.



LinuxBIOS also provides a good amount of bootup diagnostic information on the system's serial port, and allows control of the bootup process from a serial terminal as well. This can make debugging of hardware problems, or reconfiguration of a system, much easier than the usual vendor-specific keyboard-and-screen method.

This article shows you how to swap your BIOS chip for LinuxBIOS, and explains the detailed steps necessary to compile the kernel and program the code into a LinuxBIOS chip. Note that since LinuxBIOS is still very much a work in progress, some details might have changed since this article was written.

Hardware

The first thing to check if you're planning to create a LinuxBIOS system of your own is whether your motherboard is compatible and supported. A very wide range of motherboards, from an impressive list of manufacturers, are supported by the LinuxBIOS project, and your first step should be to check on the LinuxBIOS website to find out which models are likely to work. The most important requirement for a motherboard to run LinuxBIOS is that it has a BIOS chip which is removable from its socket, since this is how you change the physical chip containing the old BIOS code for a larger capacity memory chip containing the LinuxBIOS code.

This article describes the PC-Chips' M810LMR motherboard, which is a fairly cheap but nicely integrated board, containing on-board VGA, ethernet and sound. However, the steps needed for installing LinuxBIOS on any other supported motherboard are very similar to those shown here.

The other main item of hardware required in order to create a working LinuxBIOS system is the Disk-on-Chip memory device, which will be plugged into the BIOS socket on the motherboard, and which has the capacity to contain the Linux kernel and the small amount of bootstrap code which LinuxBIOS generates to initialize the motherboard hardware.

Disk-on-Chip devices are memory chips which can be "formatted" to appear like a hard disk device, and which can contain a standard Linux filing system. The LinuxBIOS project uses the Disk-on-Chip (DoC) to hold the bootup code, and also optionally a root filing system (so it is in fact possible to create a completely standalone diskless machine).

The specific DoC device used in this project is the M-Systems' MD-2800-D08 (part number MD-2802-D08 is a suitable alternative as well). This device is an 8 megabyte flash-programmable device which fits into the standard 32-pin socket used by the 2 megabit BIOS chip. Note the slightly confusing contrast between the DoC devices, which are measured in bytes, and the standard Flash Rom BIOS chips, which are measured in bits. The DoC has a capacity 32 times that of the BIOS chip it is replacing; the simple reason for this being that it is not possible to fit the Linux kernel into 2 megabits.

Finally, it is highly recommended that you obtain a 32-pin Zero Insertion Force (ZIF) socket in order to make removal and insertion of the BIOS and DoC devices simple and safe. Part of the process for programming the code into the DoC device involves removing the standard BIOS chip and replacing it with the DoC device – while the power is on and the motherboard is running. Attempting this without the use of a ZIF socket is definitely not recommended.

Getting started

The first thing you should do is read the LinuxBIOS FAQ, available from the website [1], and also the LinuxBIOS documentation for your chosen motherboard, which in the case of the M810LMR being used here, is based around the SiS630 chipset. The FAQ gives you a good idea of the overall process, and the steps involved.

Note that, although it is possible to use a "development system" for creating the LinuxBIOS code, programming this into the DoC, and then placing this into a separate "target system" which will actually run the code, it is in fact just as simple, and more convenient, to use a single machine as both development and target systems at the same time. It is assumed that you are already familiar with performing a basic Linux installation on a machine, and that you are comfortable with compiling a kernel and installing it. The steps involved in creating a LinuxBIOS machine are:

- Install Linux on your target machine, including support for the flash DoC devices (which most kernels will not have as standard)
- Get the LinuxBIOS source code
- Get the correct Linux kernel source, patch it and build it
- Configure and build the LinuxBIOS boot code for your motherboard
- Get the Memory Technology Devices (MTD) utilities and build the "erase" utility
- Remove the BIOS chip from its socket (with the power on!) and put a Disk-on-Chip in its place
- Burn the LinuxBIOS image containing the boot code and the kernel into the Disk-on-Chip
- Hit reset to start the new LinuxBIOS system.

It is a good idea to plug the ZIF socket into the motherboard, and then place the original BIOS chip into the ZIF socket in order to start the system up (Figure 1).

Firstly, note the orientation of the BIOS chip in its socket (there is a notch at one end, or a dot in one corner, of the chip), remove the chip, and plug the ZIF socket into the motherboard socket. Place the lever of the ZIF socket at the same end of the socket as the notch or dot was on the BIOS chip.

You may need to bend the pins of connectors nearby to get the ZIF socket to fit – on the M810LMR there is an unused 3-pin fan connector in the way. Make sure you plug the ZIF socket cleanly into all 32 holes on the socket on the motherboard – it's easy to miss a couple of pins at one end and get the whole thing moved along one place. You will probably want to do this with the motherboard not installed in a case, so you can look underneath the ZIF socket as you are inserting it.

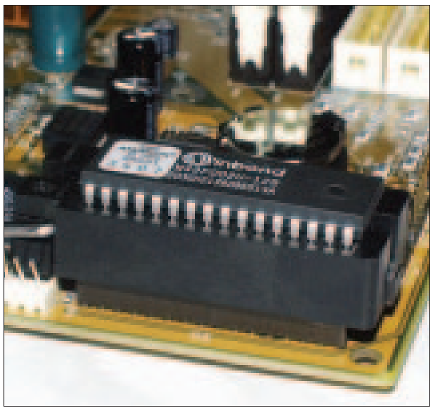


Figure 1: The ZIF socket plugged into the motherboard, with the original BIOS chip inserted

Once the ZIF socket is in place, lift the lever, insert the original BIOS chip (placing the notch or dot at the lever end of the socket) and lower the lever to secure the chip in place. Then reassemble the motherboard into the case and power up the system to make sure you get the usual BIOS startup screen, confirming that the ZIF socket and BIOS chip are correctly installed.

If you don't already have Linux installed on the machine, install a basic Linux system; note that you will require the usual development tools (compilers etc.) for building your own kernel, and you will also need to install Python, as this is used to create the configuration files used for LinuxBIOS.

The first thing you should do after installing the basic system is compile the kernel which will be used to create the LinuxBIOS system, so that it contains support for MTD (Memory Technology Devices), which is unlikely to be included in a standard kernel. It is important that you have support for loadable modules on the development machine, since for programming the DoC device in the BIOS socket of the motherboard, it is necessary to run a command before loading the DoC support modules, and therefore you cannot compile this support directly into the kernel.

If you use *make menuconfig* to configure your kernel, the additional options you need to select (accurate for a 2.4.19 kernel) in order to build LinuxBIOS into a DoC device are given in Listing 1.

There is an important change needed in one of the kernel source files in order to get MTD support working properly. If you do not make this change, you will get errors later on when you try to erase or program the device, such as:

```
/dev/mtd0: No such device
/dev/mtd0: Bad file descriptor
```

The change required is in the kernel source file */usr/src/linux/drivers/mtd/devices/docprobe.c*. Change the line which reads:

```
#define DOC_SINGLE_DRIVER
```

so that it becomes:

```
#undef DOC_SINGLE_DRIVER
```

Next, get the LinuxBIOS source by CVS from sourceforge. Press [Return] at the password prompt and ignore errors about *failed to open ./cvspass for reading*, and even *login aborted: fatal error: exiting*. Carry on with:

```
export CVS_RSH=ssh
cvs -d:pserver:anonymous@cvs.➘
freebios.sourceforge.net:➘
/cvsroot/freebios login
cvs -z3 -d:pserver:anonymous@➘
cvs.freebios.sourceforge.net:➘
/cvsroot/freebios co freebios
```

Note that the LinuxBIOS project has grown from an earlier project named FreeBIOS, and therefore this directory name will appear throughout the files used in compiling the LinuxBIOS system. Before unpacking a fresh kernel source to patch with LinuxBIOS, check the LinuxBIOS kernel patches to see which kernel version is supported for your motherboard / chipset.

Listing 1: Kernel options required

```
Loadable module support
[*] Enable loadable module support
[ ]   Set version information on all module symbols
[*]   Kernel module loader

Memory Technology Devices (MTD)
<M> Memory Technology Device (MTD) support
[ ] Debugging
< >   MTD partitioning support
< >   MTD concatenating support
--- User Modules and Translation Layers
<M>   Direct char device access to MTD devices
< >   Caching block device access to MTD devices
< >   Readonly block device access to MTD devices
< >   FTL (Flash Translation Layer) support
< >   NFTL (NAND Flash Translation Layer) support
RAM/ROM/Flash chip drivers  --->
Mapping drivers for chip access  --->
Self-contained MTD device drivers  --->
< >   Ramix PMC551 PCI Mezzanine RAM card support
< >   Uncached system RAM
< >   Test driver using RAM
< >   MTD emulation using block device
--- Disk-On-Chip Device Drivers
< >   M-Systems Disk-On-Chip 1000
< >   M-Systems Disk-On-Chip 2000 and Millennium
<M>   M-Systems Disk-On-Chip Millennium-only alternative driver
[*]     Advanced detection options for DiskOnChip
(0)     Physical address of DiskOnChip
[*]     Probe high addresses
[ ]     Probe for 0x55 0xAA BIOS Extension Signature

NAND Flash Device Drivers  --->
```

You may be able to apply the patches to a different kernel, but at this stage in the game it's probably better to build an old kernel strictly by the instructions, and make sure you can get LinuxBIOS working at all. Then afterwards you can try to bring the kernel up to the version you'd like it to be.

This article discusses kernel version 2.4.19, because this was the most recent kernel patch file available for the M810LMR motherboard. In this case the patch file is called *linux-2.4.19-sis.patch* and is found in the FreeBIOS source tree under *freebios/src/kernel-patches*. This directory contains both the patches for the kernels, and also sample config files for building the new kernel (note that not all of these are guaranteed to work in all situations – you may need to look at other config files and make some manual adjustments to get your particular setup working).

It is important to recognize that the kernel patches and config files are for the kernel you will eventually program into the DoC device and boot your LinuxBIOS machine from. They may not be the best choice for the kernel which you use to build LinuxBIOS and burn the DoC before rebooting it. When you build the kernel, simply use *make bzImage* and then leave the compiled kernel where it is. LinuxBIOS will later look for the file */usr/src/linux/vmlinux* as the image to be included in the DoC device.

Building LinuxBIOS

It is recommended that you create your own config file based on one of the examples, and make the build images for programming into the DoC device, in a different directory outside the FreeBIOS source tree. This will ensure that they are not deleted when you update your copy of the source code from the CVS repository. Because of the way the directory names are arranged, it is recommended that you create a new directory called *linuxbios* side by side with *freebios*, and build the DoC images in there:

```
mkdir linuxbios
cd linuxbios
cp ../freebios/util/config/➘
NLBConfig.py .
cp ../freebios/util/config/➘
pcchips.config .
```

The first *cp* command copies the Python program which is used to process the configuration file, so that it is in a convenient place for use later on, and the second copies the standard *pcchips.config* file (which is the one appropriate to the motherboard used in this article) into the newly-created linuxbios directory, where we shall be carrying out the work. Having copied the *pcchips.config* file into the working directory, edit the new file and make the following changes:

- Remove *single* from the end of the kernel commandline, so that the LinuxBIOS machine boots into standard multiuser mode
- Add *cpu k7* if you are using an Athlon processor
- Add option *ENABLE_MII = 1* to get the onboard ethernet working
- Change *option HAVE_FRAMEBUFFER* to *option HAVE_FRAMEBUFFER = 1* (this is simply to eliminate a warning message later on).

There may also be some editing of files needed in the LinuxBIOS source tree – for example, in the version of LinuxBIOS being used here, a change is needed in order to get the keyboard working on this particular motherboard. In the file *freebios/src/arch/i386/lib/hardwaremain.c*, uncomment the function call *keyboard_on()* around line 344. If you don't do this, then when you finally boot your LinuxBIOS machine, you will get several hundred error messages *pc_keyb: controller jammed (0xFF)*, and your keyboard will not work. It will not stop your LinuxBIOS system from working, however – you will still be able to log in on the serial port, or ssh across the network.

After making these changes, run the Python program to create the build files:

```
python NLBConfig.py ➫
pcchips.config ~/freebios
```

This creates a subdirectory within the *linuxbios* directory called *pcchips*, and creates the following files in it:

```
LinuxBIOSDoc.config
Makefile
Makefile.settings
crt0_includes.h
nsuperio.c
```

Once you have these files, and you have compiled your target kernel (which is left sitting in */usr/src/linux/vmlinux*), you can run the makefile to build your LinuxBIOS image:

```
cd pcchips
make clean
make
```

Next copy the *burn_mtd* utility into the newly-created *pcchips* directory, because by default *burn_mtd* looks in the current directory for the source files to burn into the DoC device, so there's a lot less typing involved if the utility is in the same place.

```
cp ../../freebios/util/mtd/➫
burn_mtd .
```

The *burn_mtd* utility doesn't quite match the filenames generated by the Makefile, so it is useful to edit *burn_mtd* (which is simply a shell script), in order to use the correct names: Change the first two occurrences of *vmlinux* (one in the comment on line 3, the other in *linux = vmlinux.bin.gz* on line 16) to *linux* (so that line 16 now reads *linux = linux.bin.gz*).

The next step is to get the MTD utilities from [2] and build the "erase" utility – simply download the current version of the kernel tools under the *ChangeLog* section, and then *make erase* in the *util* subdirectory of the download.

The final utility needed for programming the DoC devices is *flash_on* from the *freebios/util/sis* directory. This utility allows you to use the BIOS socket on your motherboard as a flash programmer (thus saving the need for an expensive separate piece of equipment specially for this job):

```
cd ~/freebios/util/sis
make flash_on
```

Copy the "erase" and "flash_on" utilities which you just built into your search path (for example */usr/local/sbin*). Now comes the interesting part of programming LinuxBIOS – removing the BIOS chip from a live, running motherboard, and replacing it with the Disk-on-Chip.

This is the point where you are grateful you got yourself a 32-pin ZIF socket and plugged it into your motherboard.

Programming the chip

With the power on and your system running, release the lever on the ZIF socket, remove the original BIOS chip and replace it with a Disk-on-Chip. Be very careful to get the orientation correct (the notch in the end of the chip goes at the lever end of the socket) and make sure the pins are lined up properly – remember that the socket has power on it. Secure the DoC in place with the lever on the ZIF socket. Run the command:

```
./burn_mtd
```

and it should program a LinuxBIOS chip, ready to run on your motherboard. The output of *burn_mtd* should look something like:

```
# ./burn_mtd
rmmod: module docprobe is not
loaded
rmmod: module doc2001 is not
loaded
rmmod: module docecc is not
loaded
11+1 records in
12+0 records out
0+1 records in
1+0 records out
Erase Total 1024 Units
Performing Flash Erase of length
 8192 at offset 0x7fe000 done
1+0 records in
1+0 records out
1+0 records in
1+0 records out
126+0 records in
126+0 records out
1536+0 records in
1536+0 records out
#
```

If at this stage, you get the following instead:

```
# ./burn_mtd
rmmod: module docprobe is not
loaded
rmmod: module doc2001 is not
loaded
rmmod: module docecc is not
loaded
11+1 records in
```

```
12+0 records out
0+1 records in
1+0 records out
File open error
dd: opening '/dev/mtd0':
No such device
dd: opening '/dev/mtd0':
No such device
dd: opening '/dev/mtd0':
No such device
dd: opening '/dev/mtd0':
No such device
#
```

then you should check the kernel running on your machine: ensure that you edited the file */usr/src/linux/drivers/mtd/devices/docprobe.c* to undefine *DOC_SINGLE_DRIVER* before building the kernel, that you selected the MTD options listed earlier, and that you rebooted the machine after building the kernel so that it is now actually running.

If the *burn_mtd* output looks good, reboot your machine to test the code programmed into the DoC. If your system reboots and you see a penguin in the top corner of your screen instead of an AMI or Award BIOS startup message, then you have succeeded in creating a LinuxBIOS system, booting the Linux kernel directly from the DoC instead of the hard disk boot sector as usual.

Do not worry if bits of the system do not seem to get started properly (e.g. hard disk, ethernet, keyboard, root filing system etc.). They can easily get sorted out later. The important thing at the moment is to have a running kernel at all. If you do not get a penguin on your screen followed by the normal kernel startup messages, and in fact get nothing at all, then the best way to discover what is happening with LinuxBIOS is to plug a serial cable into the first RS232 port, connect another system running a serial terminal emulator such as minicom (set to 115200 baud, 8 bits, no parity), and press reset. You should get some debugging information and startup messages displayed on the terminal, which will help to indicate how far through the startup process the system is getting. If absolutely nothing happens, then it's possible that you haven't got a suitable image burned into the DoC, so power off the motherboard, remove the DoC and put the original BIOS back in again, power the system back up, and see what you have missed from the above instructions.

Silicon disk

The last thing you may want to do once your system successfully boots the Linux kernel directly from the Disk-on-Chip, is to create a root file system in the remainder of the 8 megabyte capacity of the DoC, so that you can dispense with the hard disk drive inside your machine altogether. You will need a few more MTD kernel options turned on in your development system (to format and write the root fs) and this time also in the kernel running in the target (so that it can read from the MTD-based file system). The options you should enable during *make menuconfig* for the kernels are shown in listing 2.

Listing 2: Kernel options for MTD support

```
Memory Technology Devices (MTD)
<M> Memory Technology Device (MTD) support
[ ] Debugging
< >   MTD partitioning support
< >   MTD concatenating support
--- User Modules and Translation Layers
<M>   Direct char device access to MTD devices
< >   Caching block device access to MTD devices
< >   Readonly block device access to MTD devices
<M>   FTL (Flash Translation Layer) support
<M>   NFTL (NAND Flash Translation Layer) support
[*]     Write support for NFTL (BETA)
RAM/ROM/Flash chip drivers  --->
Mapping drivers for chip access  --->
Self-contained MTD device drivers  --->
NAND Flash Device Drivers  --->
<M>   NAND Device Support
[*]     Enable ECC correction algorithm
[ ]     Verify NAND page writes
```

Once you have recompiled the kernels (remember to install the kernel to the development system, and reboot, then leave the kernel image generated by *make bzImage* in */usr/src/linux/vmlinux*), you should be able to create and format a partition in the remaining capacity of the Disk-on-Chip device:

```
nftl_format /dev/mtd0 0x100000
```

nftl_format is in the *linuxbios/mtd/util* directory. You can then use *fdisk /dev/nftla* to create a single primary partition, occupying the entire available device (about 7 megabytes), and then format this with *mke2fs /dev/nftla1* in the usual way. You can change the device where the LinuxBios kernel expects to mount the root file system by modifying the compiled kernel image (before burning it into the DoC):

```
rdev /usr/src/linux/vmlinux ↩
/dev/nftla1
```

For suggestions on what to place into such a small root fs and still have a working Linux system, consult one of the tiny distributions such as Tom's Root Boot [3].

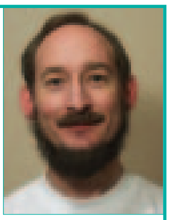
Conclusion

I found the LinuxBios project absolutely fascinating, and it is an incredible way to boot your machine directly into Linux quickly, easily, and at very little expense. I hope that you have as much fun with the system as I have. ■

INFO

[1]	LinuxBIOS website: *http://www.linuxbios.org/*
[2]	Memory Technology Device (MTD) Subsystem for Linux: *http://www.linux-mtd.infradead.org/*
[3]	Tom's Root Boot: *http://www.toms.net/rb/*

THE AUTHOR



Antony Stone has a degree in Medical Electronics, and has been working with Linux since 1994. He is Technical Director of Rockstone Ltd, a UK company producing Linux-based Firewalls, and is a contractor to Hewlett-Packard Laboratories, working on secure operating systems design. He is a part-time lecturer on the Information Security MSc at the University of London.