

Ingo Cyliax

# Embedded RT-Linux

## Part 2: Working with Flash Memory

*Once you decide that Linux might be the right OS for your embedded application, where do you go next? Ingo has the answers as he reduces the Linux kernel and even shows how to boot it from flash memory or floppy disk.*

When I introduced Linux last month, I covered the initial development of Linux as a conventional operating system for desktop and server systems. But, because Linux tries to satisfy the needs of many, it tends to be very modular and flexible.

Therefore, it can also be pressed into service as an embedded OS for many 32-bit processors. With such possibilities at hand, I wanted to explain how to embed Linux.

For many embedded applications, we want a small streamlined OS. Desktop and server installations typically include relatively large memory configurations since you don't usually know what applications and programs you might end up running on it.

Today's feature-laden desktop applications tend to be bloated for the amount of work they do. As we're frequently told, memory is cheap.

Well, although memory might be cheap, in the embedded-systems

world, every dollar spent on memory and other frivolous resources comes out of the profit margin. Our goal: make systems as lean and mean as possible, without investing a lot of effort. A compromise is sought between systems that are general purpose and those that are totally customized to one application.

So, rather than excessively customizing or writing something from scratch, let's look at how to embed Linux without too much fuss.

### REDUCING THE KERNEL

Most Linux distributions deliver a Linux kernel that is configured to be as general purpose as necessary and yet still support as many different devices as possible. That's fine for most desktop applications because there's memory and disk space to burn.

However, to embed Linux, the size of the kernel should be reduced as much as possible. There are two techniques for accomplishing this task—customize the kernel and compress the kernel image.

It is possible to configure only the modules and device drivers necessary for your applications. Linux lets you do this by running the kernel configuration script.

```
Volume in drive A has no label
Volume Serial Number is 2463-1AD1
Directory for A:/

command  com      54619 09-30-1993  6:20
debug    exe      15718 09-30-1993  6:20
loadlin  exe      32208 08-29-1998 16:28
vmlinuz          429371 08-29-1998 16:28
initrd          166233 08-29-1998 17:36
autoexec bat      285 08-29-1998 19:26
6 file(s)                698 434 bytes
614 400 bytes free
```

**Figure 1**—These are the contents of a DOS-based Linux boot floppy. The DOS utility *loadlin.exe* reads the Linux kernel image *vmlinuz* and the RAM disk image *initrd* into memory, and then transfers control to the kernel to boot it. This file can run out of a flash-based file system.

Photos 1a and 1b show screenshots of the graphical interface to this configuration script. You simply decide which modules you want included in a kernel build and save the configuration.

For many configuration options and device drivers, you can choose to select module support. This way, you can place the compiled code that implements the option or device driver into a loadable object module that is stored on the disk.

By putting the driver in modules, you can reduce the run-time memory requirements of the kernel. When you need a certain feature, the module is loaded into the kernel space and initialized. Once you're done with the feature, the module is unloaded and memory is reclaimed.

The kernel module loader can be used explicitly to load kernel modules via commands to load, unload, and list the modules currently loaded. As well, Linux has a dynamic kernel loader, which simply loads required modules as soon as the kernel needs them. You can use whichever method suits your application.

The downside of using modules is maintainability. Because kernel modules have to be stored on the disk, you have to make sure they're on the disk when they are needed. And, because many modules are needed, it's often more difficult to track their interdependency and version than if all the modules are statically loaded into a single kernel image.

Once you configure the kernel and the necessary modules for your application, you need to compile the kernel. Yes, full sources for the kernel and many modules and device drivers are provided in Linux.

To compile the kernel once it is configured, use the `make` utility. Usually, the command `make vmlinux` will compile and link everything that you need. But, if you have modules that are required by the kernel, you also need to compile the modules in a separate step using the command `make modules`.

Once everything is made, you end up with a kernel image file. Next, compress the kernel so it takes up less space. First use the command `make zImage`. Compressing the kernel is done with `gzip`, a compression tool and algorithms developed by the GNU project.

**Listing 1—In this LiLo configuration file, specify that there is only one image named "linux", and that the prompt should timeout after 5 s. We can use this configuration to install LiLo on a boot floppy if we use the invocation `lilo -C lilo.conf -r /mnt`, assuming the floppy is mounted on `/mnt`.**

```
boot=/dev/fd0
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinux
    label=linux
    root=/dev/fd0
```

## BOOTING FROM FLASH

Now, let's look at how to get Linux on a boot device. In normal Linux desktop installation, the kernel and application programs are stored and booted from a hard disk.

This situation is possible either in a dedicated Linux installation where Linux is the only OS on the disk or in multiboot configurations where Linux is one of the OSs that can be booted.

In the multiboot environment, a boot loader prompts for the OS to load. Choices can include booting DOS, Windows 95, Windows NT, and others besides Linux.

Installation of Linux to a hard disk is the normal procedure and since it's covered by the documentation that comes with most Linux distributions, I won't discuss it here. However, I do want to tell you how to deal with booting Linux in an embed-

ded environment, where you may boot Linux from flash memory or floppy disk.

To boot Linux, you have to load the Linux kernel that you have built into memory and start running it. Typically, you do this by using a boot loader like LiLo (Linux-Loader).

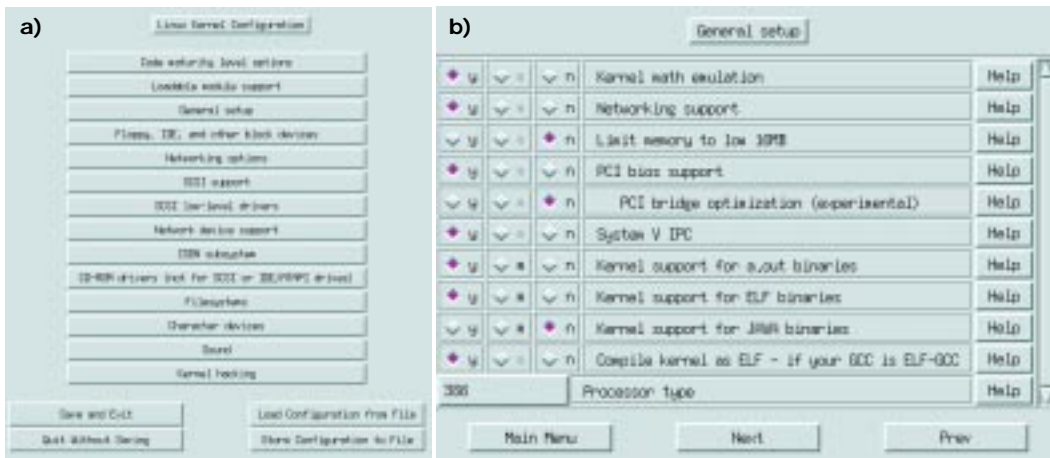
LiLo has evolved to be very flexible and configurable. It is installed in the boot block of a boot disk.

When LiLo boots, it consults a table to find out what images are available for booting. These images can be Linux kernel images, DOS or Windows boot partitions, and other Intel-based operating systems, such as OS/2 or QNX, which use the normal boot block method of booting.

Linux must be running to install and configure the LiLo boot loader. We can build a configuration LiLo file, which tells

**Listing 2—There are several steps to creating a RAM disk image suitable for use as a root partition for Linux. One handy feature in Linux is the loopback device. This device can be used to map a file to a block-oriented device using the `losetup` utility and the loopback device `/dev/loop0`.**

```
dd if=/dev/zero of=initrd.img          # create 1-MB file
    bs=2b count=1204
losetup /dev/loop0 initrd.img          # map file to loopback device
mkfs /dev/loop0                        # lay down a Linux filesystem
mount /dev/loop0 /mnt                  # mount the ramdisk image
mkdir /mnt/bin /mnt/dev /mnt/etc /mnt/lib # create some directories
cp -a /dev/console /mnt/dev           # create devices
cp -a /dev/sys/tty /mnt/dev
cp -a /dev/ram /mnt/dev
cp -a /dev/tty1 /mnt/dev
cp -a /dev/tty2 /mnt/dev
cp -a /dev/tty3 /mnt/dev
cp -a /dev/tty4 /mnt/dev
cp hello /mnt/bin                      # program to run
echo "/bin/hello" > /mnt/.profile      # create Linux startup file
cp /bin/ash.static /mnt/bin/sh         # static version of shell
umount /mnt                             # unmount the image
losetup -d /dev/loop0                  # unmap file
gzip < initrd.img > initrd             # compress it
mcopy autoexec.bat loadlin.exe        # copy everything to floppy
    vmlinux initrd a:
```



**Photo 1a—***In the kernel configuration utility, each section covers an area in the kernel, and clicking on a section brings up a detailed panel with options to select. b—In this subsection of the configuration utility, you click on each item's radio button to include the feature either statically in the kernel 'y' or 'n'. Many features can also be configured as a loadable kernel module 'm'.*

LiLo how to construct the table and how to label each boot entry. A typical configuration file is shown in Listing 1.

The configuration file contains some general information about which drive to install the boot loader onto and where to put the table on the disk. To install the loader, run `lilo`. Of course, anytime you muck around with the boot blocks of a disk drive, it's always wise to backup the contents of the drive first.

You can also use LiLo for a Linux boot floppy. To do this, use a desktop system, which serves as the development system.

Insert the floppy and install a Linux file system. Assuming the floppy is formatted, a Linux file system is installed with the `mkfs` command in Linux. Once the file is made, the floppy is mounted and a `/boot` directory that contains the Linux kernel is installed.

The `/boot` directory also contains the LiLo table image. Of course, for a boot floppy, you only have one boot option—to boot Linux. The config file featured in Listing 1 is what I'd use to configure LiLo for a floppy.

Now, if the floppy is booted in your embedded system, the system BIOS loads the LiLo boot loader, which consults its table of boot options. On the console, LiLo prompts for an image name to boot and it will timeout and boot the default image if the user does not enter anything.

That's configurable, of course, and LiLo can boot the image directly without a prompt if you want. However, providing the prompt lets a user enter options and flags for the kernel and can be used to debug things or change the location of Linux's root partition.

Once LiLo determines which image to boot, it loads the image into memory and

transfers control to it. If no options are specified, the Linux kernel uses whatever file system it was booted from as the root file system. That's important because the kernel has to find the device entries for the console and any other device to be used.

The kernel then executes `/linuxrc` or `/sbin/init`, depending on whether the file is running from RAM disk. I'll get back to that later on. If the system can't find `/sbin/init`, it starts up the shell, the command-line interpreter for Linux.

Although LiLo is flexible and complex, it requires a Linux file system to find its tables and access the Linux file system. This can be a problem.

Because Linux bypasses the BIOS to access devices like the floppy disk and hard disk, it can't access devices like a flash-memory based disk unless they look like one of the more traditional devices. So, installation of LiLo on a flash-memory-based file system is almost impossible.

Of course, if you use an ATA-compatible flash disk like the one from SanDisk shown in Photo 2, this is no problem. To the system, the disk will look like a fast IDE drive, and any OS (including Linux) that knows how to access an IDE drive will be able to deal with it.

Use the same technique here as when you generated the boot floppy. SanDisk flash disks are popular in high-end digital cameras, and PCMCIA adapters are available at many places carrying such cameras. A Linux boot disk can be built with a SanDisk on any notebook that runs Linux and has a PCMCIA adapter.

My embedded system also needs a SanDisk interface either via a PCMCIA or PC/104 adapter or through a Motorola NLX 55 Pentium-based SBC with an embedded SanDisk interface.

If you have flash-based memory already on the embedded-system board but don't want to use an ATA flash card or SanDisk, you can try a Linux boot method that doesn't require a Linux-based file system.

Here, use another Linux boot loader—the DOS utility `loadlin.exe`. This boot loader is simply a DOS program that loads a Linux kernel from a DOS file system.

With `loadlin`, you simply copy `loadlin.exe` and the kernel image `zImage`, or whatever you want to call it, onto a boot disk similar to flash memory. You can then place `loadlin zImage` into your `autoexec.bat` file, and you're all set.

With this method, you need DOS installed in your system, which requires a DOS license. Or, you can use FreeDOS (see the excellent series on FreeDOS by Pat Villani [*INK* 95–96]). Also, many flash-based embedded-system controllers come with a version of DOS installed.

There's one catch when booting Linux via the DOS boot method. To use devices, the Linux kernel still needs some sort of Linux-based file system to load the initial program from as well as to map device entries.

Well, the Linux developers considered this problem right from the early days of Linux, and that's why Linux supports using RAM disks as the root disks. You can use RAM disks with either LiLo or `loadlin`. Linux even supports using compressed RAM disk images, which take up the least amount of space on the boot media.

So, how do you build a RAM disk image for Linux to use as a root file system? You have to go back to your development system and use the sequence

of steps in Listing 2.

Just create an empty file, initialize it as a Linux file system, and populate it with device entries, the shell `/bin/sh`, and a test program. In this case, I used `hello`, which is just a standard "Hello World" program featured in most C books. Also, I placed a startup file `.profile`, which instructs the shell to execute my program.

Once everything is loaded, unmount and unmap the file and compress it into a compressed RAM disk image using the program `gzip`. This program is also used to compress the Linux kernel image. Once you have a compressed RAM disk image, simply copy this to the boot device along with the kernel image, the `autoexec.bat` file, and the `loadlin.exe` utility.

Voilà! Figure 1 shows what you need on a simple DOS-based boot disk that will work from flash memory. Listing 3 shows the `autoexec.bat` file for this setup. The only difference is that I decided to use `vmlinuz` instead of `zImage` for the kernel image. The name change is purely cosmetic.

Earlier, I mentioned that when the Linux kernel boots up, it looks for a program in either `/linuxrc` or `/sbin/init`. If the root device specified as a boot option is equal to the RAM disk, as



Photo 2—These flash-based disk modules are used in many digital cameras. Adapters for PCMCIA and PC/104 also exist for SanDisk modules.

**Listing 3—This sample `autoexec.bat` file can be used to boot Linux from a DOS file system. Often, commands to initialize and configure hardware in the system can be used before `loadlin.exe`.**

```
rem DOS Autoexec boot file for launching Linux with ramdisk
rem Author: Ingo Cyliax, Derivation Systems, Inc.
rem Date: Aug 29, 1998

rem insert DOS command here needed in order to bring machine
rem into sane state

rem start Linux
loadlin.exe vmlinuz root=/dev/ram rw initrd=initrd
```

in my example, the kernel uses `/sbin/init`.

But, if the root device is different than `/dev/ram`, and you are specifying a RAM disk with `initrd`, the Linux kernel looks for and executes `/linuxrc`. If this program exits, the kernel unmounts the RAM disk and mounts whatever root device has been specified. If Linux can't find `/sbin/init` or `/linuxrc`, it executes `/bin/sh` in the hopes that some intelligent operator will tell it what to do next.

You might wonder about this strange behavior. The reason: so root partitions can be mounted from devices not loaded into the kernel.

Booting from the network is one example of such behavior. `/linuxrc` can then be a shell script that initializes the network, allowing the kernel to mount the required root volume from the network. In my case, Linux ends up executing `/bin/sh`, which invokes its startup script in `.profile`, but it's always good to know that other options exist.

Even though I created a 1-MB RAM disk image, I'm only using about 25% of the file system for this example. Also, the boot disk I built uses less than 700 KB. So, it should be possible to build some pretty neat applications and still have them fit in a flash disk of 1–2 MB.

Although my example didn't use any of these, it's possible to use kernel-level modules and dynamic libraries in embedded applications and run them from the RAM disk.

Whether or not you want to depends on your application. For the smallest apps, you'll probably end up building a custom kernel that has only the minimum of what you need statically built in. Also, your application will be statically linked. If you're using a conventional disk, you can

think about reducing the run-time memory requirements by possibly using dynamically loaded libraries or kernel modules.

## TAKING OFF

In this article, I've shown you that, although Linux is traditionally a desktop and server OS with many bells and whistles, it's entirely possible to build tiny embedded applications using Linux.

The examples I gave here are all possible with the RedHat distribution, which includes both the Lilo and `loadlin` loader. In fact, you can boot Linux right off their CD-ROM using `loadlin`.

Linux has the advantages of being almost freely available and familiar to quite a few people. I also included some resources of where to find Linux support. Of course, only you can decide if Linux is suitable for your applications. [RPC.EPC](http://RPC.EPC)

*Ingo Cyliax has been writing for INK for two years on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego-based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. Before joining DSI, Ingo worked for over 12 years as a system and research engineer for several universities and as an independent consultant. You may reach him at [cyliax@derivation.com](mailto:cyliax@derivation.com).*

## REFERENCES

[comp.arch.embedded](http://comp.arch.embedded)  
[comp.os.linux](http://comp.os.linux)  
 Linux information, [www.linux.org](http://www.linux.org), [www.linuxhq.com](http://www.linuxhq.com)  
 RedHat, [www.redhat.com](http://www.redhat.com)

## SOURCE

**Flash-based disk module**  
 SanDisk  
 (408) 542-0500  
 Fax: (408) 542-0503  
[www.sandisk.com](http://www.sandisk.com)