IBM

# Linux system development on an embedded device

**Tinkering with PDAs for fun and profit**

Especially if you're just starting out in embedded development, the wealth of available bootloaders, scaled-down distributions, filesystems, and GUIs can seem overwhelming. But this wealth of options is actually a boon, allowing you to tailor your development or user environment exactly to your needs. This overview of embedded development on Linux will help you make sense of it all.

Anand K Santhanam has a Bachelor of Engineering degree in Computer Science from Madras University, India. He has been working for IBM Global Services (Software Labs), India since July 1999. He is a member of the Linux Group at IBM, where he concentrates primarily on ARM-Linux, device drivers, and Power Management in embedded systems. Other areas of interest are O/S internals, and networking.

Vishal Kulkarni has a Bachelor's degree in Electronic Engineering from Shivaji University; Maharashtra, India. He has been working for IBM Global Services (Software Labs), India since March 1999. Before this, he was working at IBM Austin in the US for more than 1.5 years. He is a member of the IBM Linux Group, where he concentrates primarily on ARM-Linux, device drivers and GUI on embedded devices. Other areas of interest are O/S internals and networking. He can be reached at kvishal@in.ibm.com.

01 March 2002
Also available in Japanese

Linux is making steady progress in the embedded arena. Because Linux is covered under the GPL (see Resources later in this article), anyone interested in customizing Linux to his PDA, palmtop, or wearable device can download the kernel and applications freely from the Internet and begin porting or developing. Many Linux flavors cater to the embedded/realtime market. These include RTLinux (Real-Time Linux), uclinux (Linux for MMUless devices), Montavista Linux (Linux distributions for ARM, MIPS, PPC), ARM-Linux (Linux on ARM), and others (see Resources for links to these and other terms and products mentioned in this article).

Embedded Linux development broadly involves three tiers: the bootloader, the Linux kernel, and the graphical user interface (or GUI). In this article, we will focus on some basic concepts involving these three tiers; we will provide some insights into how the bootloader, kernel, and filesystem interact; and we will investigate some of the numerous options available for the filesystem, GUI, and bootloaders.

## Bootloaders

The bootloader is usually the first piece of code that will be executed on any hardware. In conventional systems like desktops, the bootloader is normally loaded into the MBR (Master Boot Record), or the first sector of the disk where Linux resides. Normally, BIOS will transfer control to the bootloader in the case of desktops or other systems. This poses an interesting question: who loads the bootloader onto the embedded devices, which (in most cases) don't have BIOS?

Two general techniques are used to address this problem: specialized software and tiny bootcode.

**Specialized software** can directly interact with the flash device on the system remotely and install the bootloader at the given location in flash. *Flash devices* are special chips that act like storage devices, and they are persistent -- that is, the contents are not erased on reboot.

This software uses the JTAG port on the target (in embedded development, the embedded device is often referred to as the *target*), which is an interface used to execute instructions from an external input -- usually from the host machine. JFlash-linux is a popular tool for directly writing to flash. It supports a wide range of flash chips; it executes on the host machine (usually an i386 machine -- we will refer to an i386 machine as the *host* throughout this article) and accesses the flash chip of the target using the parallel port through the JTAG interface. Of course, this means that the target needs to have a parallel interface to enable it to communicate with the host. Jflash-linux is available in both Linux as well as Windows versions and is started on the command line as follows:

```
Jflash-linux <bootloader>
```

Some classes of embedded devices have **tiny bootcode** -- on the order of a few bytes -- that will initialize some DRAM settings and enable a serial (or USB or ethernet) port on the target to communicate with host programs. The host programs or loaders can then use this connection to transfer the bootloader onto the target, where it is written to flash.

After it is installed and given control, the bootloader performs the following types of functions:

- Initialize CPU speed
- Initialize memory, which includes enabling memory banks, initializing memory configuration registers, and so on
- Initialize serial port (if present on the target)
- Enable instruction/data caches
- Set up stack pointer
- Set up parameter area and construct parameter structures and tags (this is an important step, as boot parameters are used by the kernel in identifying root device, page size, memory size and more)
- Perform POST (Power On Self Test) to identify the devices present and to report any problems
- Provide support for suspend/resume for power management
- Jump to start of kernel

A typical memory layout of the system with the bootloader, parameter structure, kernel, and filesystem might be as follows:

**Listing 1. Typical memory layout**

```
        /* Top Of Memory */

                Bootloader
                Parameter Area
                Kernel
                Filesystem

        /* End Of Memory */
```

Some of the popular and freely-available bootloaders for Linux on embedded devices are Blob, Redboot, and Bootldr (see Resources for links). All these bootloaders are for Linux on ARM-based devices and require the Jflash-linux tool for installation.

Once the bootloader is installed in the flash of the target, it performs all the initializations we mentioned before. Then it is ready to receive the kernel and the filesystem from the host. Once the kernel is loaded, the bootloader transfers control to the kernel.

## Setting up a toolchain

Setting up a toolchain creates a build environment on a host machine for compiling the kernel and those applications that are to be executed on the target -- this is because the target hardware may not have binary execution-level compatibility with the host.

A toolchain consists of a set of components used for compiling, assembling, and linking the kernel and applications. These components include:

**Binutils** -- A collection of utilities for manipulating binary files. They include utilities like `ar`, `as`, `objdump`, `objcopy`, and so on.

**Gcc** -- The GNU C compiler.

**Glibc** -- The C library that all user applications will link to. The kernel and other things that avoid using any C library functions can be compiled without it.

Building a toolchain establishes a cross-compiler environment. A *native compiler* compiles instructions for the same sort of processor as the one it is running on. A *cross-compiler* runs on one type of processor, but compiles instructions for another. Setting up a cross-compiler toolchain from scratch is not an easy task: it involves downloading the sources, patching, configuring, compiling, setting up headers, installation, and much, much more. In addition, the memory and hard disk requirements for such an exhaustive build procedure are huge. As if that weren't enough, numerous problems can crop up during the build phase due to problems with dependencies, configuration, or header setup.

So it's a good thing that pre-compiled binaries are available on the Internet (it's less good that they're mostly limited to ARM-based systems at this time, but in time that too shall change). Some of the more popular pre-compiled toolchains include those from Compaq (Familiar Linux), LART (LART Linux), and Embedian (based on but not related to Debian) -- all for ARM-based platforms.

### Kernel setup

The Linux community is very active in adding features and support for new hardware, in fixing bugs in the kernel, and making general improvements in a timely manner. This results in having a new release of a stable Linux tree roughly every 6 months or less. Different kernel trees and patches for specific architectures are maintained by different maintainers. When choosing a kernel for a project, you need to evaluate how stable the latest release is, whether it caters to the project requirements and the hardware platform, the comfort level from a programming point of view, and other intangibles. It is also very important to find out about all of the patches that need to be applied to the base kernel to tune it for your specific architecture.

### Kernel layout

The kernel layout is divided into architecture-specific and architecture-independent parts. The architecture-specific part of the kernel executes first and sets up hardware registers, configures the memory map, performs architecture-specific initialization, and then transfers control to the architecture-independent part of the kernel. It is during this second phase that the rest of the system is initialized. The directory arch/ under the kernel tree consists of different subdirectories, each for a different architecture (MIPS, ARM, i386, SPARC, PPC, and so on). Each of these subdirectories includes kernel/ and mm/ subdirectories, which contain architecture-specific code to do things like initialize memory, set up IRQs, enable cache, set up kernel page tables, and so on. These functions are called first once the kernel is loaded and given control, then the rest of the system is initialized.

The kernel can be compiled either as vmlinux, Image, or zImage depending on the available system resources and the functionality of the bootloader. The main difference between vmlinux and zImage is that *vmlinux* is the real (uncompressed) executable, while *zImage* is a self-extracting compressed file containing more or less the same information -- but compressed to deal with the (usually Intel-imposed) 640 KB boot-time limit. For a definitive explanation of all this, please see the *Linux Magazine* article "Kernel Configuration: dealing with the unexpected" (see Resources).

## Kernel linking and loading

Once the kernel is compiled for the target system, it is loaded into the target system's memory (either in DRAM or in flash) using the bootloader (which has already been loaded onto the target's flash). The bootloader communicates with the host using the serial, ESB, or ethernet port to transfer the kernel into the target's flash or DRAM. After the kernel is fully loaded to the target, the bootloader passes control to the address where the kernel was loaded.

The kernel executable consists of many object files linked together. The object files have many sections such as text, data, init data, bass, and so forth. These object files are linked and loaded by a file known as a *linker script*. The function of the linker script is to map the sections of the input object files into an output file; in other words, it links all the input object files into a single executable whose sections are loaded at specified addresses. *vmlinux.lds* is the kernel linker script present in the arch/<target>/ directory, and it is responsible for linking the various sections of the kernel and loading them at a particular offset in memory. A typical vmlinux.lds looks like this:

**Listing 2. Typical vmlinux.lds file**

```
OUTPUT_ARCH(<arch>)            /* <arch> includes architecture type */
ENTRY(stext)                   /* stext is the kernel entry point */
SECTIONS                       /* SECTIONS command describes the layout
                                  of the output file */
{
    .  = TEXTADDR;             /* TEXTADDR is LMA for the kernel */
    .init : {                  /* Init code and data*/
             _stext = .;       /* First section is stext followed
                                  by __init data section */
             __init_begin = .;
                    *(.text.init)
             __init_end = .;
            }
    .text : {          /* Real text segment follows __init_data section */
             _text = .;
                    *(.text)
             _etext = .;       /* End of text section*/
            }
    .data :{
             _data=.;          /* Data section comes after text section */
                    *(.data)
             _edata=.;
            }                  /* Data section ends here */
    .bss : {                   /* BSS section follows symbol table section */
             __bss_start = .;
                    *(.bss)
             _end = . ;        /* BSS section ends here */
            }
 }
```

`LMA` is the load module address; it signifies the address in the target's virtual memory where the kernel will be loaded. `TEXTADDR` is the virtual start address of the kernel and its value is specified in the Makefile under arch/<target>/. This address has to match the address the bootloader uses.

Once the bootloader copies the kernel to flash or DRAM, the kernel is relocated to the `TEXTADDR` -- which is usually in DRAM. The bootloader then transfers control to this address so that the kernel can start executing.

## Parameter passing and kernel boot up

`stext` is the kernel entry point, which means that the code under this section is the first to execute when the kernel boots up. It is usually written in assembly, and is normally present under the arch/<target>/ kernel directory. The code sets up the kernel page directory, creates identity kernel mapping, identifies architecture and processor, and branches to `start_kernel` (the main routine whereby the system is initialized).

`start_kernel` calls `setup_arch` as the first step where the architecture-specific setup is done. This includes initializing hardware registers, identifying the root device and the amount of DRAM and flash available in the system, specifying the number of pages available in the system, the filesystem size, and so on. All this information is passed in parameter form from the bootloader to the kernel.

There are two ways to pass parameters from bootloader to kernel: parameter_structure and a tag list. Of these, param structure is deprecated as it imposes restrictions by specifying that each and every parameter has to be at a particular offset within param_struct in memory. Recent kernels expect parameters to be passed as a tag list and will convert parameters to a tagged format. `param_struct` is defined in include/asm/setup.h. Some of its important fields are:

**Listing 3. Sample parameter structure**

```
struct param_struct  {
 unsigned long page_size;     /* 0:  Size of the page  */
 unsigned long nr_pages;      /* 4:  Number of pages in the system */
 unsigned long ramdisk        /* 8: ramdisk size */
 unsigned long rootdev;       /* 16: Number representing the root device */
 unsigned long initrd_start;  /* 64: starting address of initial ramdisk */
                                          /* This can be either in flash/dram */
 unsigned long initrd_size;   /* 68: size of initial ramdisk */
}
```

Note that the numbers represent the offset within the param structure where the fields are to be defined. This means that if the bootloader places the parameter structure at address 0xc0000100, then the rootdev parameter is placed at 0xc0000100 + 16, initrd_start is placed at 0xc0000100 + 64, and so on -- otherwise, the kernel will encounter difficulties in interpreting the correct parameters.

As mentioned earlier, most of the 2.4.x series kernels expect the parameters to be passed in a tagged list format, because of the constraints involved in parameter passing from bootloader to kernel. In a tagged list, each tag consists of a tag_header that identifies the parameter passed, followed by values for the parameter. The general format for a tag in the tag list could be as follows:

**Listing 4. Sample tag format. The kernel identifies each tag by the <ATAG_TAGNAME> header.**

```
#define <aTAG_TAGNAME>  <Some Magic number>

struct <tag_tagname> {
        u32 <tag_param>;
        u32 <tag_param>;
};

/* Example tag for passing memory information */
```

```
#define ATAG_MEM        0x54410002  /* Magic number */

struct tag_mem32 {
        u32     size;               /* size of memory */
        u32     start;              /* physical start address of memory*/
};
```

The `setup_arch` also needs to perform any memory mappings for flash banks, system registers, and other specific devices. Once the architecture-specific setup is done, control returns to the start_kernel function where the rest of the system is initialized. These additional initialization tasks include:

- Setting up traps
- Initializing interrupts
- Initializing timers
- Initializing console
- Calling `mem_init`, which calculates the number of pages in various zones, high memory, and so on
- Initializing the slab allocator and creating slab caches for VFS, buffer cache, etc.
- Setting up various filesystems like proc, ext2, JFFS2
- Creating `kernel_thread`, which execs the init command in the filesystem and displays the lign prompt. If no init program is present in /bin, /sbin, or /etc, the kernel will execute the shell present in /bin of the filesystem.

## Device drivers

Embedded systems usually have a number of devices for user interaction, like touchscreens, keypads, roller wheels, sensors, RS232 interfaces, LCDs, and so on. In addition to these, there are many other specialized devices including flash, USB, GSM, and more. The kernel controls -- and user applications including GUIs access -- all of these devices through their respective device drivers. This section focuses on device drivers for some important devices that are normally used in almost every embedded environment.

### Framebuffer driver

This is one of the most important drivers, because it is through this driver that the system screen comes alive. The framebuffer driver normally has three layers. The lowest layer is the basic console driver drivers/char/console.c, which provides a portion of the generic interface for the text console. Using console driver functions, we can print text on the screen -- but not pictures or animation (for that we need to use video mode functionality, typically present in the middle layer, or drivers/video/fbcon.c). This second driver provides the generic interface for drawing in video mode.

The framebuffer is the memory on the video card, which needs to be memory-mapped onto the user space so that pictures and text can be written on this memory segment: this information will then be reflected on the screen. Framebuffer support speeds both drawing and overall performance. It is also where the top layer driver comes into the picture: the top layer is a very hardware-specific driver, which needs to support the different hardware aspects of the video card -- like enabling/disabling the video card controller, the depths and modes supported, the palettes, etc. All three layers are interdependent for proper video functionality. The device associated with the frame buffer is /dev/fb0 (Major Number 29, Minor Number 0).

## Input device drivers

Touchable panels are one of the most basic user interaction devices for embedded devices -- keypads, sensors, and roller wheels also are included in many different devices for various purposes.

The main functionality of the touch panel device is to report any time that the user touches it, and to identify the coordinates of the touch. This is commonly done by generating an interrupt whenever a touch is made.

The role of the device driver, then, is to query the touch screen controller whenever an interrupt occurs, and to ask the controller to send the coordinates of the touch. Once the driver receives the coordinates, it signals the user applications about the touch and the availability of any data, and sends the applications the data (if that is possible). The user application then processes the data according to its needs.

Almost all input devices -- including the keypad -- work on similar principles.

## Flash MTD Drivers

MTD devices are those class of devices like flash chips, compact flash cards, memory sticks, and so on, which are increasingly finding their way into embedded devices.

MTD drivers are a new class of drivers developed under Linux specifically for the embedded environment. The main advantage of using MTD drivers over conventional block device drivers is that MTD drivers are designed specifically for flash-based devices, so they generally have better support, better management, and a better interface for sector-based erases, reads, and writes. The MTD driver interface under Linux is classified into two modules: the user module and the hardware module.

### User modules

These modules provide interfaces to be used directly from userspace: raw character access, raw block access, FTL (Flash Transition Layer -- a type of filesystem used on flash), and JFS (or Journaled File System -- which provides a filesystem directly on the flash rather than emulating a block device). The current version of JFS on flash is JFFS2 (described later in this article).

### Hardware modules

These provide physical access to memory devices, and are not used directly. They are accessed through the user modules above. These provide the actual routines for read, erase, and write on flash.

### MTD driver setup

In order to access a specific flash device and put a filesystem on top of it, the MTD subsystem needs to be compiled into the kernel. This includes selecting the appropriate MTD hardware and user modules. Currently, the MTD subsystem supports a wide range of flash devices -- and more and more drivers are being added for different flash chips.

Two popular user modules that enable access to flash are `MTD_CHAR` and `MTD_BLOCK`.

`MTD_CHAR` provides raw character access to the flash, while `MTD_BLOCK` projects the flash as a normal block device (like an IDE disk), on which a filesystem can be created. The devices associated with `MTD_CHAR` are /dev/mtd0, mtd1, mtd2 (etc), while the devices associated with `MTD_BLOCK` are /dev/mtdblock0, mtdblock1 (etc). Since `MTD_BLOCK` devices provide block-device-like emulation, it is often preferable to create filesystems like FTL and JFFS2 on top of this emulation.

To do this, it may be necessary to create a partition table to separate the flash device into a bootloader section, a kernel section, and a filesystem section. A sample partition table might include the following information:

**Listing 5. A simple flash device partition for MTD**

```
struct mtd_partition sample_partition = {
     {
                                        /* First partition */
          name : bootloader,            /* Bootloader section */
          size    : 0x00010000,         /* Size  */
          offset  : 0,          /* Offset from start of flash- location 0x0*/
          mask_flags : MTD_WRITEABLE    /* This partition is not writable */
     },
     {                                  /* Second partition */
          name : Kernel,                /* Kernel section */
          size    :  0x00100000,        /* Size */
          offset : MTDPART_OFS_APPEND,  /* Append after bootloader section */
          mask_flags : MTD_WRITEABLE    /* This partition is not writable */
     },
     {                                  /* Third partition */
          name : JFFS2,                 /* JFFS2 filesystem */
          size    :  MTDPART_SIZ_FULL,  /* Occupy rest of flash */
          offset :  MTDPART_OFS_APPEND  /* Append after kernel section */
     }
}
```

The above partition table uses the `MTD_BLOCK` interface to partition the flash device. The device nodes for these partitions are:

**Device nodes for our simple flash partition**

```
 User          device node             Major number    Minor number

 Bootloader    /dev/mtdblock0          31              0
 Kernel        /dev/mtdblock1          31              1
 Filesystem    /dev/mtdblock2          31              2
```

In this case, the bootloader has to pass the correct parameters to the kernel regarding the root device node (/dev/mtdblock2) and the address in flash where the filesystem is found (in this case, `FLASH_BASE_ADDRESS + 0x04000000`). Once the partition is done, the flash device is ready for loading or mounting a filesystem.

The main aim of the MTD subsystem in Linux is to provide a generic interface between the hardware drivers and the upper layers, or user modules, of the system. Hardware drivers need not know about the methods employed by user modules like JFFS2 and FTL. All they really need to provide is a set of simple routines for `read`, `write` and `erase` operations on the underlying flash system.

## Filesystems for embedded devices

The system needs a way to store and retrieve information in a structured format; this is where the filesystem comes in. Ramdisk (see Resources) is a mechanism for creating and mounting filesystems using the computer's RAM as the device, and it is usually used in diskless systems (including, of course, tiny embedded devices that contain only flash chips as media for persistent storage).

The user can choose the type of filesystem based on the need for reliability, robustness, and/or enhanced features. The following section discusses a few of the available choices, as well as their advantages and disadvantages.

### The second Extended Filesystem (Ext2fs)

Ext2fs is the de facto standard filesystem for Linux, having ousted its predecessor, the Extended File System (or Extfs). Extfs supported a maximum file size of 2 gigabytes and a maximum file name size of

255 characters -- and it did not support inodes (including data modification timestamps). Ext2fs does much better; it has these **advantages**:

Ext2fs supports up to 4 terabytes of memory.

Ext2fs filenames can be up to 1012 characters in length.

The administrator can choose the logical block size when creating the filesystem (typical sizes are 1024, 2048, and 4096 bytes).

Ext2fs implements fast symbolic links: no data blocks need to be allocated for this purpose, and the target name is directly stored in the inode table. This leads to an increase in performance, especially in speed.

Because of its stability, reliability, and robustness, the Ext2 filesystem is used on almost all Linux-based systems including desktops, servers, and workstations -- and even some embedded devices. However, Ext2fs has some **disadvantages** when it comes to embedded devices:

Ext2fs is designed for block devices like IDE devices, where the logical block size will be on the order of 512 bytes, 1 kilobyte, and so on. This is not well suited for flash devices where sector sizes vary.

The Ext2 filesystem does not provide good management of sector-based erase/writes. To erase a single byte in a sector in Ext2fs, a whole sector has to be copied to RAM, erased, then rewritten. Considering that flash devices have a limited erase lifecycle (about 100,000 erases) after which they can't be used, this is not a particularly good practice.

Ext2fs is not crash-proof in the case of a power failure.

The Ext2 filesystem does not support wear levelling, thereby reducing sector/flash life. (Wear levelling ensures that different areas of the address range are used for writes and/or erases in rotation to extend flash life.)

Ext2fs does not have particularly brilliant sector management, making the designing of a block driver extremely difficult.

For these reasons, an MTD/JFFS2 combination is generally preferred over Ext2fs for the embedded environment.

**Mounting Ext2fs with Ramdisk**

The Ext2 filesystem (and for that matter, any filesystem) can be created and mounted onto an embedded device using the concept of Ramdisk.

**Listing 6. Creating a simple Ext2fs-based Ramdisk**

```
mke2fs -vm0 /dev/ram 4096
mount -t ext2 /dev/ram /mnt
cd /mnt
cp /bin, /sbin, /etc, /dev ... files in mnt
cd ../
umount /mnt
dd if=/dev/ram bs=1k count=4096 of=ext2ramdisk
```

`mke2fs` is the utility used to create an ext2 filesystem -- creating the super block, inodes, inode table, and etc -- on any device.

In the above usage, `/dev/ram` is the device on which an ext2 filesystem of 4096 blocks is built. Then the device (`/dev/ram`) is mounted on a temporary directory named `/mnt` and all the necessary files are copied. Once these files are copied, the filesystem is unmounted and the contents of the device (`/dev/ram`) is dumped into a file (ext2ramdisk), which is the required Ramdisk (the Ext2 filesystem).

The above sequence creates a Ramdisk of 4 MB and fills it with the necessary file utilities.

Some of the important directories to include in Ramdisk are:

**/bin** -- Holds most of the binaries like `init`, `busybox`, `shell`, file management utilities, and etc.

**/dev** -- Contains all the device nodes used in the device

**/etc** -- Contains all the configuration files for the system

**/lib** -- Contains all the necessary libraries like libc, libdl, and so on

## Journaling Flash File System, version 2 (JFFS2)

The original JFFS was developed by Axis Communications of Sweden, and was improved upon by David Woodhouse at Red Hat. The second version, JFFS2, is emerging as the de facto filesystem for raw flash chips for tiny embedded devices. The JFFS2 filesystem is log-structured, meaning that it is basically a long list of nodes. Each node contains some information about the file of which it is part -- possibly the name of the file, maybe some data. JFFS2 is being increasingly favored over Ext2fs for diskless embedded devices for these **advantages**:

JFFS2 performs flash erase/write/read on the sector level better than the Ext2 filesystem.

JFFS2 provides better crash/power-down-safe protection than Ext2fs. When a little amount of data needs to be changed, the Ext2 filesystem copies the whole sector to memory (DRAM), merges the new data in memory, and writes back the whole sector. This means that for changing a single word, the read/erase/write routine has to be done for the whole (64 KB) sector -- which is highly inefficient. On the off chance that there is a power failure or other catastrophe while data is being merged in DRAM, the entire collection of data is lost, since the flash sector is erased after the data has been read to DRAM. JFFS2 appends files rather than rewriting whole sectors, and features crash/power-done-safe protection.

Perhaps most importantly, JFFS2 was specifically created for embedded devices like flash chips, so its overall design provides better flash management.

Having been written primarily for use with flash devices, the **disadvantages** of using JFFS2 in an embedded environment are few:

JFFS2 can tend to slow down a great deal when the filesystem is full or nearly full. This is because of garbage collection issues (see Resources for more information).

### Creating a JFFS2 filesystem

A JFFS2 filesystem (basically a Ramdisk using JFFS2) is created under Linux with the `mkfs.jffs2` command:

**Listing 7. Creating a JFFS2 filesystem**

```
mkdir jffsfile
cd jffsfile
```

```
 /* copy all the /bin, /etc, /usr/bin, /sbin/ binaries and /dev entries
that are needed for the filesystem here */

 /* Type the following command under jffsfile directory to create the JFFS2 Image */

 ./mkfs.jffs2 -e 0x40000 -p -o ../jffs.image
```

The above shows the typical use of mkfs.jffs2. The `-e` option specifies the erase sector size of the flash (typically 64 kilobytes). The `-p` option is used to pad the remaining space in the image with zeroes. The `-o` option is used for the output file, which is usually the JFFS2 filesystem image -- in this case, jffs.image. Once the JFFS2 filesystem is created, it is loaded into the appropriate location in flash (at the address where the bootloader tells the kernel to look for the filesystem) so that the kernel can mount it.

## tmpfs

Once an embedded device becomes a fully functional unit with Linux running on top of it, lots of daemons tend to run in the background generating lots of log messages. In addition, all of the kernel logging mechanisms like syslogd, dmesg, and klogd, generate a lot of messages under the /var and /tmp directories. Since a huge amount of data is produced by these processes, it is not advisable to allow all of these writes to happen to flash. As these messages need not be persistent across reboots, the solution to this problem is to use tmpfs.

Tmpfs is a memory-based filesystem, which is mainly used for the sole purpose of reducing unnecessary flash writes to the system. Since tmpfs resides in RAM, operations to write/read/erase happen in RAM rather than in flash. Therefore, log messages go to RAM rather than to flash, and they are not preserved across reboots. Tmpfs also makes use of the disk swap space for storage, and of the virtual memory (VM) subsystem when requesting pages for storing files.

**Advantages** of tmpfs include:

Dynamic filesystem size -- The filesystem size can shrink or grow depending on the number of files or directories that are copied or created or deleted. This results in optimal usage of memory.

Speed -- As tmpfs resides in RAM, the reads and writes are almost instantaneous. Even if files are stored in swap, the I/O operations are at very high speed.

One **disadvantage** of tmpfs is that all data is lost when the system reboots. Therefore, important data can't be stored on tmpfs.

### Mounting tmpfs

Unlike most other filesystems like Ext2fs and JFFS2, which reside on top of the underlying block device, tmpfs sits directly on top of the VM. Thus mounting the tmpfs filesystem is a simple affair:

**Listing 8. Mounting tmpfs**

```
/* Entries in /etc/rc.d/rc.sysinit for creating/using tmpfs */

# mount -t tmpfs tmpfs /var -o size=512k
# mkdir -p /var/tmp
# mkdir -p /var/log
# ln -s /var/tmp /tmp
```

The above commands will create a tmpfs on /var and limit the maximum size of tmpfs to 512 K. Also, the tmp/ and log/ directories are made part of tmpfs so that log messages are stored in RAM.

If you were to add an entry for tmpfs to /etc/fstab, it might look something like this:

```
tmpfs /var tmpfs size=32m 0 0
```

This will mount a new tmpfs filesystem at /var.

## Graphical User Interface (GUI) options

The Graphical User Interface (GUI) is the most crucial system aspect from the user's point of view: the user interacts with the system through the GUI. So the GUI should be easy to use and pretty reliable. But it also needs to be memory-conscious in order to execute seamlessly on memory-constrained, tiny embedded devices. As a result, it needs to be lightweight, and very fast during loading.

Another important aspect to consider is the licensing issues involved. Some GUI distributions have licenses that allow them to be used, even in commercial products, free of charge. Others require that royalties be paid if the GUI is incorporated into a project.

In the end, most developers will probably opt for XFree86 because it provides a familiar environment for them to use their favorite tools. But newer GUIs in the market like Century Software's Microwindows (Nano-X) and Trolltech's QT/Embedded are giving X a run for its money in the embedded Linux arena mainly because of their small footprint, speed of execution, and custom widget support.

Let's look at each of these options.

### Xfree86 4.X (X11R6.4 with framebuffer support)

The XFree86 Project, Inc. is an organization that produces XFree86, a freely redistributable, open source X Window System. The X Window System (X11) provides resources for applications to display themselves in a graphical manner, and is the most commonly used windowing system on UNIX and UNIX-like boxes. It is small and efficient, it runs on a wide range of hardware, it is network-transparent, and it is well documented. X11 offers powerful facilities for window management, event handling, synchronization, and inter-client communication -- and most developers are already familiar with its APIs. It has built-in support for kernel framebuffer, and a very small footprint -- which is very helpful for devices with less memory. The X Server supports VGA and non-VGA graphic cards, has support for depths 1, 2, 4, 8, 16, and 32, and has built-in support for rendering. The latest release is XFree86 4.1.0.

Its **advantages** include:

- Use of framebuffer architecture speeds performance.
- Relatively small footprint -- size is in the range of 600 to 700 Kilobytes, which makes it easy to run on small devices.
- Very good support: lots of documentation is available online, there are also a number of mailing lists dedicated to XFree86 development.
- The X API is very extensive.

Its **disadvantages** include:

- Slower performance than the most recent crop of embedded GUI ware.
- Again, when compared with the newest developments in GUIs -- like Nano-X or QT/Embedded, which are specifically designed for the embedded environment -- XFree86 seems to have rather large memory

requirements.

## Microwindows

Microwindows is an open source project from Century Software that is designed for tiny devices with small display units. It has a lot of features aimed at the modern graphical windowing environment. Like X, Microwindows is supported on variety of platforms.

Microwindows architecture is client/server based and has a layered design. At the lowest level are the screen and input device drivers (as for the keyboard or mouse) to interact with the actual hardware. At the middle level, a portable graphics engine provides support for line draws, area fills, polygons, clipping, and color models.

At the uppermost level, Microwindows supports two APIs: the Win32/WinCE API implementation also known as Microwindows, and the other API mostly resembles that of GDK and is known as Nano-X. Nano-X is used on Linux. It is an X-like API intended for low-footprint applications.

Microwindows has support for 1, 2, 4, and 8 bpp (bits per pixel) palletized displays, as well as 8, 15, 24, and 32 bpp trucolor displays. Microwindows also supports framebuffer, which makes it pretty fast. The footprint of the Nano-X server is somewhere around 100 to 150 kilobytes.

The average raw Nano-X app is in 30 to 60 K in size. Since Nano-X is designed for low-end devices with memory constraints, it does not have support for a large number of functions as X has, and so it can't really serve as a replacement for Tiny X (Xfree86 4.1).

You can run FLNX, a version of the FLTK (Fast Light Toolkit) application development environment modified to target Nano-X rather than X, on top of Microwindows. FLTK is described in this article.

Nano-X's **advantages** include:

- Unlike the Xlib implementation, Nano-X still runs synchronously per client, meaning that once a client request packet is sent, the server waits until the whole packet has arrived before servicing another client. This keeps the server code immensely simple, while still running very quickly.
- Small footprint

Nano-Xs' **disadvantages** include:

- Networking features are not properly tuned as yet (especially network transparency).
- Not many ready-to-use applications are available.
- Compared to X, Nano-X has not as much documentation and not as well supported -- although development is going on at lightening speed lately, so this may change.

## FLTK API on Microwindows

FLTK is a simple but flexible GUI toolkit that is gaining increasing attention in the Linux world, especially for low-footprint environments. It provides most of the widgets you would expect from a GUI toolkit like buttons, dialog boxes, text boxes, and a nice selection of "valuators" (widgets used for the input of numeric values). These include sliders, scroll bars, dials, and a few others.

The Linux version of FLTK targeted for the Microwindows GUI engine is known as FLNX. FLNX is made up of two components: Fl_Widget and FLUID. Fl_Widget consists of all of the basic widget APIs. FLUID (Fast Light User Interface Designer) is a graphical editor used to produce FLTK source code. In all, FLNX is an excellent UI builder that can be used to create applications for embedded environments.

The footprint of Fl_Widget is about 40 to 48 K and FLUID (which includes every widget) weighs in at around 380 K. These very small footprints are making Fl_Widget and FLUID very popular in the world of embedded development.

**Advantages** include:

- Anyone accustomed to developing GUI-based applications under more established environments like Windows will adjust to the FLTK environment quite easily.
- Its documentation includes a very complete and well-written manual.
- It is distributed under the LGPL, so developers have flexibility in how they license their applications.
- FLTK is a C++ library (Perl and Python bindings are also available). The choice of an object-oriented model is a good one, as most modern GUI environments are object-oriented; this should also facilitate the porting of applications written to similar APIs.
- Century's environment provides several useful facilities, such as ScreenTop and the ViewML browser.

Its **disadvantage** is:

- While vanilla FLTK works with both X and Windows APIs, FLNX does not. Its incompatibility with X hinders its adoption in many projects.

## Qt/Embedded

Qt/Embedded is Trolltech's new graphical user interface system for embedded Linux. Trolltech initially created Qt for the Linux desktop as a cross-platform development tool. It supports a variety of UNIX flavors, as well as Microsoft Windows. KDE, one of the most popular Linux desktop environments, is written with Qt.

Qt/Embedded is based on the original Qt, with a lot of fine tuning for the embedded environment. Qt Embedded directly interacts with the Linux I/O facilities via the Qt API. Those who are conversant and comfortable with object-oriented programming will find it an ideal environment. Also the object-oriented architecture makes the code structured, reusable, and fast. The Qt GUI is pretty fast compared to other GUIs, and its lack of layering makes Qt/Embedded the most compact environment for running Qt-based programs.

Trolltech has also come up with a Qt Palmtop Environment, popularly known as Qpe. Qpe provides a basic desktop window, and the environment provides an easy-to-use interface for development. Qpe includes a full set of Personal Information Management (PIM) applications, Internet clients, utilities, and more. However, you need to obtain a commercial license from Trolltech in order to integrate Qt/Embedded or Qpe into a product. (The original Qt has been available under the GPL since version 2.2.)

Its **advantages** include:

Object-oriented architecture, which makes for faster execution

Small footprint, about 800 K

Anti-aliased text and alpha blended pixmaps

Its **disadvantage** is:

Qt/Embedded and Qpe are available under commercial licenses only.

## Conclusion

Embedded Linux development is evolving rapidly. You must study and choose from a variety of options for everything from the bootloader and distribution to the filesystem and GUI. But thanks to this freedom of choice and to a very active Linux community, embedded development on Linux has reached new vistas, and tailoring modules to your specifications has never been simpler. This has resulted in many modern handheld and miniature devices coming as open boxes, which is a very good thing -- as is the fact that you needn't be an expert to choose from among these modules to tailor your device to your own needs and wants.

We hope that this introductory overview of the embedded Linux space has whet your appetite for experimentation, and that you will find tinkering with tiny devices to your liking. To aid you further in your projects, see the Resources below for links to even more in-depth information on the technologies we have surveyed here.

## Resources

For an excellent explanation of the differences between vmlinux and zimage, scroll down to the "Booting your kernel" section of Kernel Configuration: dealing with the unexpected by Alessandro Rubini (*Linux Magazine*).

The Embedded Linux Distributions Quick Reference Guide covers many commercial and open source distributions (*Linux Devices*, August 2001).

The Wiki toolchain page includes links to -- and comments on -- all three of the toolchains mentioned in this article.

The Memory Technology Device (MTD) Subsystem for Linux aims to simplify the creation of drivers for memory devices (especially flash devices).

The Linux MTD, JFFS HOWTO by Vipin Malik will help you get MTD and JFFS2 working together.

Linux for PowerPC Embedded Systems HOWTO has a good list of device drivers.

Understanding Linux device drivers is easy with this introductory tutorial (*Linux Planet*).

To gain an intimate familiarity with Linux Device Drivers, consult O'Reilly's Linux Device Drivers, 2nd Edition.

Binutils, GCC, and Glibc are all available for download from the Free Software Foundation.

Many useful downloads are available at Netwinder.org, a volunteer site devoted to development on the NetWinder platform.

Read all about Ramdisk in Mark Nielsen's excellent How to use a Ramdisk for Linux.

FLNX is based on FLTK (The Fast Light Toolkit).

The Second Extended Filesystem Ext2fs makes its home at SourceForge.

Extensive background information on JFFS2: The Journalling Flash File System, version 2 is outlined by David Woodhouse of Red Hat UK.

You can read more about tmpfs at Linux HeadQuarters.

Flash Filesystems for Embedded Linux Systems by Cliff Brake and Jeff Sutherland goes over even more filesystems for flash devices (*Embedded Linux Journal*).

Xfree86 is the home for X development.

Information about Microwindows and Nano-X are at the Microwindows site.

Check out an older discussion of some of the disadvantages of Microwindows (GNOME gtk developers' mailing list).

The Embedded Linux GUI/Windowing Quick Reference Guide has a wealth of links (*Linux Devices*, February 2002).

The General Public License, or GPL guarantees a user's right to copy, distribute and modify software.

Penguinppc.org is the home for Linux on the PowerPC family of processors. This site has a very informative tutorial on setting up toolchain for PPC-based architectures.

Linux Devices is a comprehensive site with press releases, quick references, news, and feature reporting on all things Linux and embedded.

The Silicon Penguin listings site has an exhaustive collection of embedded Linux resources.

The Embedded Linux Consortium is a non-profit mutual benefit association that welcomes membership by individual developers working in the embedded Linux space.

Visit IBM's home for embedded Linux for news, products, and developer resources.

The IBM Linux wristwatch is an example of a tiny embedded device running Linux; one of the authors of this article, Vishal Kulkarni, was involved. Read about it in IBM's Linux Watch: The Challenge of Miniaturization (in PDF format), or in this article (*FreeOS.com*, March 2001).

Browse more Linux resources on *developerWorks*.

Browse more Wireless resources on *developerWorks*.