Linux 正在嵌入式开发领域稳步发展。因为 Linux 使用 GPL（请参阅本文后面的 参考资料），所以任何对将 Linux 定制于 PDA、掌上机或者可佩带设备感兴趣的人都可以从因特网免费下载其内核和应用程序，并开始移植或开发。许多 Linux 改良品种迎合了嵌入式 / 实时市场。它们包括 RTLinux（实时 Linux）、uclinux（用于非 MMU 设备的 Linux）、Montavista Linux（用于 ARM、MIPS、PPC 的 Linux 分发版）、ARM-Linux（ARM 上的 Linux）和其它 Linux 系统（请参阅 参考资料以链接到本文中提到的这些和其它术语及产品。）

嵌入式 Linux 开发大致涉及三个层次：引导装载程序、Linux 内核和图形用户界面（或称 GUI）。在本文中，我们将集中讨论涉及这三层的一些基本概念；深入了解引导装载程序、内核和文件系统是如何交互的；并将研究可用于文件系统、GUI 和引导装载程序的众多选项中的一部分。

引导装载程序

引导装载程序通常是在任何硬件上执行的第一段代码。在象台式机这样的常规系统中，通常将引导装载程序装入主引导记录（Master Boot Record，(MBR)）中，或者装入 Linux 驻留的磁盘的第一个扇区中。通常，在台式机或其它系统上，BIOS 将控制移交给引导装载程序。这就提出了一个有趣的问题：谁将引导装载程序装入（在大多数情况中）没有 BIOS 的嵌入式设备上呢？

解决这个问题有两种常规技术：专用软件和微小的引导代码（tiny bootcode）。

专用软件可以直接与远程系统上的闪存设备进行交互并将引导装载程序安装在闪存的给定位置中。*闪存设备*是与存储设备功能类似的特殊芯片，而且它们能持久存储信息 — 即，在重新引导时不会擦除其内容。

这个软件使用目标（在嵌入式开发中，嵌入式设备通常被称为 *目标*）上的 JTAG 端口，它是用于执行外部输入（通常来自主机机器）的指令的接口。JFlash-linux 是一种用于直接写闪存的流行工具。它支持为数众多的闪存芯片；它在主机机器（通常是 i386 机器 — 本文中我们把一台 i386 机器称为 *主机*）上执行并通过 JTAG 接口使用并行端口访问目标的闪存芯片。当然，这意味着目标需要有一个并行接口使它能与主机通信。Jflash-linux 在 Linux 和 Windows 版本中都可使用，可以在命令行中用以下命令启动它：

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

某些种类的嵌入式设备具有 微小的引导代码— 根据几个字节的指令 — 它将初始化一些 DRAM 设置并启用目标上的一个串行（或者 USB，或者以太网）端口与主机程序通信。然后，主机程序或装入程序可以使用这个连接将引导装载程序传送到目标上，并将它写入闪存。

在安装它并给予其控制后，这个引导装载程序执行下列各类功能：

- 初始化 CPU 速度
- 初始化内存，包括启用内存库、初始化内存配置寄存器等
- 初始化串行端口（如果在目标上有的话）
- 启用指令 / 数据高速缓存
- 设置堆栈指针
- 设置参数区域并构造参数结构和标记（这是重要的一步，因为内核在标识根设备、页面大小、内存大小以及更多内容时要使用引导参数）
- 执行 POST（加电自检）来标识存在的设备并报告任何问题
- 为电源管理提供挂起 / 恢复支持
- 跳转到内核的开始

带有引导装载程序、参数结构、内核和文件系统的系统典型内存布局可能如下所示：

清单 1. 典型内存布局

```
1 /* Top Of Memory */
2         Bootloader
3         Parameter Area
4         Kernel
5         Filesystem
  /* End Of Memory */
```

嵌入式设备上一些流行的并可免费使用的 Linux 引导装载程序有 Blob、Redboot 和 Bootldr（请参阅 参考资料获得链接）。所有这些引导装载程序都用于基于 ARM 设备上的 Linux，并需要 Jflash-linux 工具用于安装。
一旦将引导装载程序安装到目标的闪存中，它就会执行我们上面提到的所有初始化工作。然后，它准备接收来自主机的内核和文件系统。一旦装入了内核，引导装载程序就将控制转给内核。

设置工具链

设置工具链在主机机器上创建一个用于编译将在目标上运行的内核和应用程序的构建环境 — 这是因为目标硬件可能没有与主机兼容的二进制执行级别。
工具链由一套用于编译、汇编和链接内核及应用程序的组件组成。 这些组件包括：

- Binutils — 用于操作二进制文件的实用程序集合。它们包括诸如 `ar` 、 `as` 、 `objdump` 、 `objcopy` 这样的实用程序。
- Gcc— GNU C 编译器。
- Glibc— 所有用户应用程序都将链接到的 C 库。避免使用任何 C 库函数的内核和其它应用程序可以在没有该库的情况下进行编译。

构建工具链建立了一个交叉编译器环境。 *本地编译器*编译与本机同类的处理器的指令。 *交叉编译器*运行在某一种处理器上，却可以编译另一种处理器的指令。重头设置交叉编译器工具链可不是一项简单的任务：它包括下载源代码、修补补丁、配置、编译、设置头文件、安装以及很多很多的操作。另外，这样一个彻底的构建过程对内存和硬盘的需求是巨大的。如果没有足够的内存和硬盘空间，那么在构建阶段由于相关性、配置或头文件设置等问题会突然冒出许多问题。
因此能够从因特网上获得已预编译的二进制文件是一件好事（但不太好的一点是，目前它们大多数只限于基于 ARM 的系统，但迟早会改变的）。一些比较流行的已预编译的工具链包括那些来自 Compaq（Familiar Linux ）、LART（LART Linux）和 Embedian（基于 Debian 但与它无关）的工具链 — 所有这些工具链都用于基于 ARM 的平台。

内核设置

Linux 社区正积极地为新硬件添加功能部件和支持、在内核中修正错误并且及时地进行常规改进。这导致大约每 6 个月（或 6 个月不到）就有一个稳定的 Linux 树的新发行版。不同的维护者维护针对特定体系结构的不同内核树和补丁。当为一个项目选择了一个内核时，您需要评估最新发行版的稳定性如何、它是否符合项目要求和硬件平台、从编程角度来看它的舒适程度以及其它难以确定的方面。还有一点也非常重要：找到需要应用于基本内核的所有补丁，以便为特定的体系结构调整内核。

内核布局

内核布局分为特定于体系结构的部分和与体系结构无关的部分。内核中特定于体系结构的部分首先执行，设置硬件寄存器、配置内存映射、执行特定于体系结构的初始化，然后将控制转给内核中与体系结构无关的部分。系统的其余部分在这第二个阶段期间进行初始化。内核树下的目录 arch/ 由不同的子目录组成，每个子目录用于一个不同的体系结构（MIPS、ARM、i386、SPARC、PPC 等）。每一个这样的子目录都包含 kernel/ 和 mm/ 子目录，它们包含特定于体系结构的代码来完成象初始化内存、设置 IRQ、启用高速缓存、设置内核页面表等操作。一旦装入内核并给予其控制，就首先调用这些函数，然后初始化系统的其余部分。
根据可用的系统资源和引导装载程序的功能，内核可以编译成 vmlinux、Image 或 zImage。vmlinux 和 zImage 之间的主要区别在于 *vmlinux*是实际的（未压缩的）可执行文件，而 *zImage*是或多或少包含相同信息的自解压压缩文件 — 只是压缩它以处理（通常是 Intel 强制的）640 KB 引导时间的限制。有关所有这些的权威性解释，请参阅 *Linux Magazine*的文章“Kernel Configuration: dealing with the unexpected”（请参阅 参考资料）。

内核链接和装入

一旦为目标系统编译了内核后，通过使用引导装载程序（它已经被装入到目标的闪存中），内核就被装入到目标系统的内存（在 DRAM 中或者在闪存中）。通过使用串行、USB 或以太网端口，引导装载程序与主机通信以将内核传送到目标的闪存或 DRAM 中。在将内核完全装入目标后，引导装载程序将控制传递给装入内核的地址。

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init 数据、bass 等等。这些对象文件都是由一个称为 *链接器脚本*的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。 *vmlinux.lds*是存在于 arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。典型的 vmlinux.lds 看起来象这样：

清单 2. 典型的 vmlinux.lds 文件

```
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 secti
                                __bss_start = .;
                                        *(.bss)
                                _end = . ;               /* BSS section ends here */
                            }
    }
```

LMA 是装入模块地址；它表示将要装入内核的目标虚拟内存中的地址。 TEXTADDR 是内核的虚拟起始地址，并且在 arch/<target>/ 下的 Makefile 中指定它的值。这个地址必须与引导装载程序使用的地址相匹配。

一旦引导装载程序将内核复制到闪存或 DRAM 中，内核就被重新定位到 TEXTADDR — 它通常在 DRAM 中。然后，引导装载程序将控制转给这个地址，以便内核能开始执行。

参数传递和内核引导

stext 是内核入口点，这意味着在内核引导时将首先执行这一节下的代码。它通常用汇编语言编写，并且通常它在 arch/<target>/ 内核目录下。这个代码设置内核页面目录、创建身份内核映射、标识体系结构和处理器以及执行分支 start_kernel （初始化系统的主例程）。

start_kernel 调用 setup_arch 作为执行的第一步，在其中完成特定于体系结构的设置。这包括初始化硬件寄存器、标识根设备和系统中可用的 DRAM 和闪存的数量、指定系统中可用页面的数目、文件系统大小等等。所有这些信息都以参数形式从引导装载程序传递到内核。

将参数从引导装载程序传递到内核有两种方法：parameter_structure 和标记列表。在这两种方法中，不赞成使用参数结构，因为它强加了限制：指定在内存中，每个参数必须位于 param_struct 中的特定偏移量处。最新的内核期望参数作为标记列表的格式来传递，并将参数转化为已标记格式。 param_struct 定义在 include/asm/setup.h 中。它的一些重要字段是：

清单 3. 样本参数结构

```
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 eithe
  unsigned long initrd_size;     /* 68: size of initial ramdisk */
}
```

请注意：这些数表示定义字段的参数结构中的偏移量。这意味着如果引导装载程序将参数结构放置在地址 0xc0000100，那么 rootdev 参数将放置在 0xc0000100 + 16，initrd_start 将放置在 0xc0000100 + 64 等等 — 否则，内核将在解释正确的参数时遇到困难。

正如上面提到的，因为从引导装载程序到内核的参数传递会有一些约束条件，所以大多数 2.4.x 系列内核期望参数以已标记的列表格式传递。在已标记的列表中，每个标记由标识被传递参数的 tag_header 以及其后的参数值组成。标记列表中标记的常规格式可以如下所示：

清单 4. 样本标记格式。内核通过 <ATAG_TAGNAME> 头来标识每个标记。

```
#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 memo
  };
```

setup_arch 还需要对闪存存储库、系统寄存器和其它特定设备执行内存映射。一旦完成了特定于体系结构的设置，控制就返回到初始化系统其余部分的 start_kernel 函数。这些附加的初始化任务包含：

- 设置陷阱
- 初始化中断
- 初始化计时器
- 初始化控制台
- 调用 mem_init ，它计算各种区域、高内存区等内的页面数量
- 初始化 slab 分配器并为 VFS、缓冲区高速缓存等创建 slab 高速缓存
- 建立各种文件系统，如 proc、ext2 和 JFFS2
- 创建 kernel_thread ，它执行文件系统中的 init 命令并显示 lign 提示符。 如果在 /bin、/sbin 或 /etc 中没有 init 程序，那么内核将执行文件系统的 /bin 中的 shell。

设备驱动程序

嵌入式系统通常有许多设备用于与用户交互，象触摸屏、小键盘、滚动轮、传感器、RA232 接口、LCD 等等。除了这些设备外，还有许多其它专用设备，包括闪存、USB、GSM 等。内核通过所有这些设备各自的设备驱动程序来控制它们，包括 GUI 用户应用程序也通过访问这些驱动程序来访问设备。本节着重讨论通常几乎在每个嵌入式环境中都会使用的一些重要设备的设备驱动程序。

帧缓冲区驱动程序

这是最重要的驱动程序之一，因为通过这个驱动程序才能使系统屏幕显示内容。帧缓冲区驱动程序通常有三层。最底层是基本控制台驱动程序 drivers/char/console.c，它提供了文本控制台常规接口的一部分。通过使用控制台驱动程序函数，我

们能将文本打印到屏幕上 — 但图形或动画还不能（这样做需要使用视频模式功能，通常出现在中间层，也就是 drivers/video/fbcon.c 中）。这个第二层驱动程序提供了视频模式中绘图的常规接口。

帧缓冲区是显卡上的内存，需要将它内存映射到用户空间以便可以将图形和文本能写到这个内存段上：然后这个信息将反映到屏幕上。帧缓冲区支持提高了绘图的速度和整体性能。这也是顶层驱动程序引人注意之处：顶层是非常特定于硬件的驱动程序，它需要支持显卡不同的硬件方面 — 象启用 / 禁用显卡控制器、深度和模式的支持以及调色板等。所有这三层都相互依赖以实现正确的视频功能。与帧缓冲区有关的设备是 /dev/fb0（主设备号 29，次设备号 0）。

输入设备驱动程序

可触摸板是用于嵌入式设备的最基本的用户交互设备之一 — 小键盘、传感器和滚动轮也包含在许多不同设备中以用于不同的用途。

触摸板设备的主要功能是随时报告用户的触摸，并标识触摸的坐标。这通常在每次发生触摸时，通过生成一个中断来实现。

然后，这个设备驱动程序的角色是每当出现中断时就查询触摸屏控制器，并请求控制器发送触摸的坐标。一旦驱动程序接收到坐标，它就将有关触摸和任何可用数据的信号发送给用户应用程序，并将数据发送给应用程序（如果可能的话）。然后用户应用程序根据它的需要处理数据。

几乎所有输入设备 — 包括小键盘 — 都以类似原理工作。

闪存 MTD 驱动程序

MTD 设备是象闪存芯片、小型闪存卡、记忆棒等之类的设备，它们在嵌入式设备中的使用正在不断增长。

MTD 驱动程序是在 Linux 下专门为嵌入式环境开发的新的一类驱动程序。相对于常规块设备驱动程序，使用 MTD 驱动程序的主要优点在于 MTD 驱动程序是专门为基于闪存的设备所设计的，所以它们通常有更好的支持、更好的管理和基于扇区的擦除和读写操作的更好的接口。Linux 下的 MTD 驱动程序接口被划分为两类模块：用户模块和硬件模块。

用户模块

这些模块提供从用户空间直接使用的接口：原始字符访问、原始块访问、FTL（闪存转换层，Flash Transition Layer — 用在闪存上的一种文件系统）和 JFS（即日志文件系统，Journaled File System — 在闪存上直接提供文件系统而不是模拟块设备）。用于闪存的 JFS 的当前版本是 JFFS2（稍后将在本文中描述）。

硬件模块

这些模块提供对内存设备的物理访问，但并不直接使用它们。通过上述的用户模块来访问它们。这些模块提供了在闪存上读、擦除和写操作的实际例程。

MTD 驱动程序设置

为了访问特定的闪存设备并将文件系统置于其上，需要将 MTD 子系统编译到内核中。这包括选择适当的 MTD 硬件和用户模块。当前，MTD 子系统支持为数众多的闪存设备 — 并且有越来越多的驱动程序正被添加进来以用于不同的闪存芯片。有两个流行的用户模块可启用对闪存的访问： MTD_CHAR 和 MTD_BLOCK 。

MTD_CHAR 提供对闪存的原始字符访问，而 MTD_BLOCK 将闪存设计为可以在上面创建文件系统的常规块设备（象 IDE 磁盘）。与 MTD_CHAR 关联的设备是 /dev/mtd0、mtd1、mtd2（等等），而与 MTD_BLOCK 关联的设备是 /dev/mtdblock0、mtdblock1（等等）。由于 MTD_BLOCK 设备提供象块设备那样的模拟，通常更可取的是在这个模拟基础上创建象 FTL 和 JFFS2 那样的文件系统。

为了进行这个操作，可能需要创建分区表将闪存设备分拆到引导装载程序节、内核节和文件系统节中。样本分区表可能包含以下信息：

清单 5. MTD 的简单闪存设备分区

```
1  struct mtd_partition sample_partition = {
2              {
3                                                                   /* First p
4                          name : bootloader,                    /* Bootloader section */
5                          size       : 0x00010000,              /* Size   */
6                          offset   : 0,                  /* Offset from start of flash- location
7                          mask_flags : MTD_WRITEABLE       /* This partition is not writable */
8              },
9              {                                                                /* Second pa
10                         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 par
                        name : JFFS2,                                       /* JFFS2 filesystem */
                        size       :   MTDPART_SIZ_FULL,      /* Occupy rest of flash */
                        offset :   MTDPART_OFS_APPEND     /* Append after kernel section */
                }
        }
```

上面的分区表使用了 MTD_BLOCK 接口对闪存设备进行分区。这些分区的设备节点是：

简单闪存分区的设备节点

```
User           device node                Major number       Minor number
  Bootloader       /dev/mtdblock0                31                          0
  Kernel               /dev/mtdblock1                31                          1
  Filesystem       /dev/mtdblock2                31                          2
```

在本例中，引导装载程序必须将有关 root 设备节点（/dev/mtdblock2）和可以在闪存中找到文件系统的地址（本例中是 FLASH_BASE_ADDRESS + 0x04000000 ）的正确参数传递到内核。一旦完成分区，闪存设备就准备装入或挂装文件系统。Linux 中 MTD 子系统的主要目标是在系统的硬件驱动程序和上层，或用户模块之间提供通用接口。硬件驱动程序不需要知道象 JFFS2 和 FTL 那样的用户模块使用的方法。所有它们真正需要提供的就是一组对底层闪存系统进行 read 、 write 和 erase 操作的简单例程。

嵌入式设备的文件系统

系统需要一种以结构化格式存储和检索信息的方法；这就需要文件系统的参与。Ramdisk（请参阅 参考资料）是通过将计算机的 RAM 用作设备来创建和挂装文件系统的一种机制，它通常用于无盘系统（当然包括微型嵌入式设备，它只包含作为永久存储媒质的闪存芯片）。

用户可以根据可靠性、健壮性和 / 或增强的功能的需求来选择文件系统的类型。下一节将讨论几个可用选项及其优缺点。

第二版扩展文件系统（Ext2fs）

Ext2fs 是 Linux 事实上的标准文件系统，它已经取代了它的前任 — 扩展文件系统（或 Extfs）。Extfs 支持的文件大小最大为 2 GB，支持的最大文件名称大小为 255 个字符 — 而且它不支持索引节点（包括数据修改时间标记）。Ext2fs 做得更好；它的 优点是：

- Ext2fs 支持达 4 TB 的内存。
- Ext2fs 文件名称最长可以到 1012 个字符。
- 当创建文件系统时，管理员可以选择逻辑块的大小（通常大小可选择 1024、2048 和 4096 字节）。
- Ext2fs 了实现快速符号链接：不需要为此目的而分配数据块，并且将目标名称直接存储在索引节点（inode）表中。这使性能有所提高，特别是在速度上。

因为 Ext2 文件系统的稳定性、可靠性和健壮性，所以几乎在所有基于 Linux 的系统（包括台式机、服务器和工作站 — 并且甚至一些嵌入式设备）上都使用 Ext2 文件系统。然而，当在嵌入式设备中使用 Ext2fs 时，它有一些 缺点：

- Ext2fs 是为象 IDE 设备那样的块设备设计的，这些设备的逻辑块大小是 512 字节，1 K 字节等这样的倍数。这不太适合于扇区大小因设备不同而不同的闪存设备。
- Ext2 文件系统没有提供对基于扇区的擦除 / 写操作的良好管理。在 Ext2fs 中，为了在一个扇区中擦除单个字节，必须将整个扇区复制到 RAM，然后擦除，然后重写入。考虑到闪存设备具有有限的擦除寿命（大约能进行 100,000 次擦除），在此之后就不能使用它们，所以这不是一个特别好的方法。
- 在出现电源故障时，Ext2fs 不是防崩溃的。

- Ext2 文件系统不支持损耗平衡，因此缩短了扇区 / 闪存的寿命。（损耗平衡确保将地址范围的不同区域轮流用于写和 / 或擦除操作以延长闪存设备的寿命。）
- Ext2fs 没有特别完美的扇区管理，这使设计块驱动程序十分困难。

由于这些原因，通常相对于 Ext2fs，在嵌入式环境中使用 MTD/JFFS2 组合是更好的选择。

用 Ramdisk 挂装 Ext2fs

通过使用 Ramdisk 的概念，可以在嵌入式设备中创建并挂装 Ext2 文件系统（以及用于这一目的的任何文件系统）。

清单 6. 创建一个简单的基于 Ext2fs 的 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 是用于在任何设备上创建 ext2 文件系统的实用程序 — 它创建超级块、索引节点以及索引节点表等等。

在上面的用法中， /dev/ram 是上面构建有 4096 个块的 ext2 文件系统的设备。然后，将这个设备（ /dev/ram ）挂装在名为 /mnt 的临时目录上并且复制所有必需的文件。一旦复制完这些文件，就卸装这个文件系统并且设备（ /dev/ram ）的内容被转储到一个文件（ext2ramdisk）中，它就是所需的 Ramdisk（Ext2 文件系统）。

上面的顺序创建了一个 4 MB 的 Ramdisk，并用必需的文件实用程序来填充它。

一些要包含在 Ramdisk 中的重要目录是：

- /bin — 保存大多数象 init 、 busybox 、 shell 、文件管理实用程序等二进制文件。
- /dev— 包含用在设备中的所有设备节点
- /etc— 包含系统的所有配置文件
- /lib— 包含所有必需的库，如 libc、libdl 等

日志闪存文件系统，版本 2（JFFS2）

瑞典的 Axis Communications 开发了最初的 JFFS，Red Hat 的 David Woodhouse 对它进行了改进。 第二个版本，JFFS2，作为用于微型嵌入式设备的原始闪存芯片的实际文件系统而出现。JFFS2 文件系统是日志结构化的，这意味着它基本上是一长列节点。每个节点包含有关文件的部分信息 — 可能是文件的名称、也许是一些数据。相对于 Ext2fs，JFFS2 因为有以下这些 优点而在无盘嵌入式设备中越来越受欢迎：

- JFFS2 在扇区级别上执行闪存擦除 / 写 / 读操作要比 Ext2 文件系统好。
- JFFS2 提供了比 Ext2fs 更好的崩溃 / 掉电安全保护。当需要更改少量数据时，Ext2 文件系统将整个扇区复制到内存（DRAM）中，在内存中合并新数据，并写回整个扇区。这意味着为了更改单个字，必须对整个扇区（64 KB）执行读 / 擦除 / 写例程 — 这样做的效率非常低。要是运气差，当正在 DRAM 中合并数据时，发生了电源故障或其它事故，那么将丢失整个数据集合，因为在将数据读入 DRAM 后就擦除了闪存扇区。JFFS2 附加文件而不是重写整个扇区，并且具有崩溃 / 掉电安全保护这一功能。
- 这可能是最重要的一点：JFFS2 是专门为象闪存芯片那样的嵌入式设备创建的，所以它的整个设计提供了更好的闪存管理。

因为本文主要是写关于闪存设备的使用，所以在嵌入式环境中使用 JFFS2 的 缺点很少：

- 当文件系统已满或接近满时，JFFS2 会大大放慢运行速度。这是因为垃圾收集的问题（更多信息，请参阅 参考资料）。

创建 JFFS2 文件系统

在 Linux 下，用 mkfs.jffs2 命令创建 JFFS2 文件系统（基本上是使用 JFFS2 的 Ramdisk）。

清单 7. 创建 JFFS2 文件系统

```
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
```

上面显示了 mkfs.jffs2 的典型用法。 -e 选项确定闪存的擦除扇区大小（通常是 64 千字节）。 -p 选项用来在映像的剩余空间用零填充。 -o 选项用于输出文件，通常是 JFFS2 文件系统映像 — 在本例中是 jffs.image。一旦创建了 JFFS2 文件系统，它就被装入闪存中适当的位置（引导装载程序告知内核查找文件系统的地址）以便内核能挂装它。

tmpfs

当 Linux 运行于嵌入式设备上时，该设备就成为功能齐全的单元，许多守护进程会在后台运行并生成许多日志消息。另外，所有内核日志记录机制，象 syslogd、dmesg 和 klogd，会在 /var 和 /tmp 目录下生成许多消息。由于这些进程产生了大量数据，所以允许将所有这些写操作都发生在闪存是不可取的。由于在重新引导时这些消息不需要持久存储，所以这个问题的解决方案是使用 tmpfs。

tmpfs 是基于内存的文件系统，它主要用于减少对系统的不必要的闪存写操作这一唯一目的。因为 tmpfs 驻留在 RAM 中，所以写 / 读 / 擦除的操作发生在 RAM 中而不是在闪存中。因此，日志消息写入 RAM 而不是闪存中，在重新引导时不会保留它们。tmpfs 还使用磁盘交换空间来存储，并且当为存储文件而请求页面时，使用虚拟内存（VM）子系统。

tmpfs 的 优点包括：

- 动态文件系统大小 — 文件系统大小可以根据被复制、创建或删除的文件或目录的数量来缩放。使得能够最理想地使用内存。
- 速度 — 因为 tmpfs 驻留在 RAM，所以读和写几乎都是瞬时的。即使以交换的形式存储文件，I/O 操作的速度仍非常快。

tmpfs 的一个 缺点是当系统重新引导时会丢失所有数据。因此，重要的数据不能存储在 tmpfs 上。

挂装 tmpfs

诸如 Ext2fs 和 JFFS2 等大多数其它文件系统都驻留在底层块设备之上，而 tmpfs 与它们不同，它直接位于 VM 上。因而，挂装 tmpfs 文件系统是很简单的事：

清单 8. 挂装 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
```

上面的命令将在 /var 上创建 tmpfs 并将 tmpfs 的最大大小限制为 512 K。同时，tmp/ 和 log/ 目录成为 tmpfs 的一部分以便在 RAM 中存储日志消息。

如果您想将 tmpfs 的一个项添加到 /etc/fstab，那么它可能看起来象这样：

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

这将在 /var 上挂装一个新的 tmpfs 文件系统。

图形用户界面（GUI）选项

从用户的观点来看，图形用户界面（GUI）是系统的一个最至关重要的方面：用户通过 GUI 与系统进行交互。所以 GUI 应该易于使用并且非常可靠。但它还需要是有内存意识的，以便在内存受限的、微型嵌入式设备上可以无缝执行。所以，它应该是轻量级的，并且能够快速装入。

另一个要考虑的重要方面涉及许可证问题。一些 GUI 分发版具有允许免费使用的许可证，甚至在一些商业产品中也是如此。另一些许可证要求如果想将 GUI 合并入项目中则要支付版税。
最后，大多数开发人员可能会选择 XFree86，因为 XFree86 为他们提供了一个能使用他们喜欢的工具的熟悉环境。但是市场上较新的 GUI，象 Century Software 的 Microwindows（Nano-X）和 Trolltech 的 QT/Embedded，与 X 在嵌入式 Linux 的竞技舞台中展开了激烈竞争，这主要是因为它们占用很少的资源、执行的速度很快并且具有定制窗口构件的支持。
让我们看一看这些选项中的每一个。

Xfree86 4.X（带帧缓冲区支持的 X11R6.4）

XFree86 Project, Inc. 是一家生产 XFree86 的公司，该产品是一个可以免费重复分发、开放源码的 X Window 系统。X Window 系统（X11）为应用程序以图形方式进行显示提供了资源，并且它是 UNIX 和类 UNIX 的机器上最常用的窗口系统。它很小但很有效，它运行在为数众多的硬件上，它对网络透明并且有良好的文档说明。X11 为窗口管理、事件处理、同步和客户机间通信提供强大的功能 — 并且大多数开发人员已经熟悉了它的 API。它具有对内核帧缓冲区的内置支持，并占用非常少的资源 — 这非常有助于内存相对较少的设备。X 服务器支持 VGA 和非 VGA 图形卡，它对颜色深度 1、2、4、8、16 和 32 提供支持，并对渲染提供内置支持。最新的发行版是 XFree86 4.1.0。
它的 优点包括：

- 帧缓冲区体系结构的使用提高了性能。
- 占用的资源相对很小 — 大小在 600 K 到 700 K 字节的范围内，这使它很容易在小型设备上运行。
- 非常好的支持：在线有许多文档可用，还有许多专用于 XFree86 开发的邮递列表。
- X API 非常适合扩展。

它的 缺点包括：

- 比最近出现的嵌入式 GUI 工具性能差。
- 此外，当与 GUI 中最新的开发 — 象专门为嵌入式环境设计的 Nano-X 或 QT/Embedded — 相比时，XFree86 似乎需要更多的内存。

Microwindows

Microwindows 是 Century Software 的开放源代码项目，设计用于带小型显示单元的微型设备。它有许多针对现代图形视窗环境的功能部件。象 X 一样，有多种平台支持 Microwindows。
Microwindows 体系结构是基于客户机 / 服务器的并且具有分层设计。最底层是屏幕和输入设备驱动程序（关于键盘或鼠标）来与实际硬件交互。在中间层，可移植的图形引擎提供对线的绘制、区域的填充、多边形、裁剪以及颜色模型的支持。
在最上层，Microwindows 支持两种 API：Win32/WinCE API 实现，称为 Microwindows；另一种 API 与 GDK 非常相似，它称为 Nano-X。Nano-X 用在 Linux 上。它是象 X 的 API，用于占用资源少的应用程序。
Microwindows 支持 1、2、4 和 8 bpp（每像素的位数）的 palletized 显示，以及 8、16、24 和 32 bpp 的真彩色显示。Microwindows 还支持使它速度更快的帧缓冲区。Nano-X 服务器占用的资源大约在 100 K 到 150 K 字节。
原始 Nano-X 应用程序的平均大小在 30 K 到 60 K。由于 Nano-X 是为有内存限制的低端设备设计的，所以它不象 X 那样支持很多函数，因此它实际上不能作为微型 X（Xfree86 4.1）的替代品。
可以在 Microwindows 上运行 FLNX，它是针对 Nano-X 而不是 X 进行修改的 FLTK（快速轻巧工具箱(Fast Light Toolkit)）应用程序开发环境的一个版本。本文中描述 FLTK。
Nano-X 的 优点包括：

- 与 Xlib 实现不同，Nano-X 仍在每个客户机上同步运行，这意味着一旦发送了客户机请求包，服务器在为另一个客户机提供服务之前一直等待，直到整个包都到达为止。这使服务器代码非常简单，而运行的速度仍非常快。
- 占用很小的资源

Nano-X 的 缺点包括：

- 联网功能部件至今没有经过适当地调整（特别是网络透明性）。
- 还没有太多现成的应用程序可用。
- 与 X 相比，Nano-X 虽然近来正在加速开发，但仍没有那么多文档说明而且没有很好的支持，但这种情形会有所改变。

Microwindows 上的 FLTK API

FLTK 是一个简单但灵活的 GUI 工具箱，它在 Linux 世界中赢得越来越多的关注，它特别适用于占用资源很少的环境。它提供了您期望从 GUI 工具箱中获得的大多数窗口构件，如按钮、对话框、文本框以及出色的“赋值器”选择（用于输入数值的窗口构件）。还包括滑动器、滚动条、刻度盘和其它一些构件。
针对 Microwindows GUI 引擎的 FLTK 的 Linux 版本被称为 FLNX。FLNX 由两个组件构成：Fl_Widget 和 FLUID。
Fl_Widget 由所有基本窗口构件 API 组成。FLUID（快速轻巧的用户界面设计器(Fast Light User Interface Designer, FLUID)）是用来产生 FLTK 源代码的图形编辑器。总的来说，FLNX 是能用来为嵌入式环境创建应用程序的一个出色的 UI 构建器。
Fl_Widget 占用的资源大约是 40 K 到 48 K，而 FLUID（包括了每个窗口构件）大约占用 380 K。这些非常小的资源占用率使 Fl_Widget 和 FLUID 在嵌入式开发世界中非常受欢迎。
优点包括：

- 习惯于在象 Windows 这样已建立得较好的环境中开发基于 GUI 的应用程序的任何人都会非常容易地适应 FLTK 环境。
- 它的文档包括一本十分完整且编写良好的手册。
- 它使用 LGPL 进行分发，所以开发人员可以灵活地发放他们应用程序的许可证。
- FLTK 是一个 C++ 库（Perl 和 Python 绑定也可用）。面向对象模型的选择是一个好的选择，因为大多数现代 GUI 环境都是面向对象的；这也使将编写的应用程序移植到类似的 API 中变得更容易。
- Century Software 的环境提供了几个有用的工具，诸如 ScreenToP 和 ViewML 浏览器。

它的 缺点是：

- 普通的 FLTK 可以与 X 和 Windows API 一同工作，而 FLNX 不能。它与 X 的不兼容性阻碍了它在许多项目中的使用。

Qt/Embedded

Qt/Embedded 是 Trolltech 新开发的用于嵌入式 Linux 的图形用户界面系统。Trolltech 最初创建 Qt 作为跨平台的开发工具用于 Linux 台式机。它支持各种有 UNIX 特点的系统以及 Microsoft Windows。KDE — 最流行的 Linux 桌面环境之一，就是用 Qt 编写的。
Qt/Embedded 以原始 Qt 为基础，并做了许多出色的调整以适用于嵌入式环境。Qt Embedded 通过 Qt API 与 Linux I/O 设施直接交互。那些熟悉并已适应了面向对象编程的人员将发现它是一个理想环境。而且，面向对象的体系结构使代码结构化、可重用并且运行快速。与其它 GUI 相比，Qt GUI 非常快，并且它没有分层，这使得 Qt/Embedded 成为用于运行基于 Qt 的程序的最紧凑环境。
Trolltech 还推出了 Qt 掌上机环境（Qt Palmtop Environment，俗称 Qpe）。Qpe 提供了一个基本桌面窗口，并且该环境为开发提供了一个易于使用的界面。Qpe 包含全套的个人信息管理（Personal Information Management (PIM)）应用程序、因特网客户机、实用程序等等。然而，为了将 Qt/Embedded 或 Qpe 集成到一个产品中，需要从 Trolltech 获得商业许可证。（原始 Qt 自版本 2.2 以后就可以根据 GPL 获得 。）
它的 优点包括：

- 面向对象的体系结构有助于更快地执行

- 占用很少的资源，大约 800 K
- 抗锯齿文本和混合视频的象素映射

它的 缺点是：

- Qt/Embedded 和 Qpe 只能在获得商业许可证的情况下才能使用。

结束语

嵌入式 Linux 开发正迅速地发展着。您必须学习并从引导装载程序和分发版到文件系统和 GUI 中的每一个事物的各种选项中作出选择。但是要感谢有这种选择自由度以及非常活跃的 Linux 社区，Linux 上的嵌入式开发已经达到了新的境界，并且调整模块以适合您的规范从未比现在更简单。这已经导致出现了许多时新的手持和微型设备作为开放盒，这是件好事 — 因为事实是您不必成为一个专家从这些模块中进行选择来调整您的设备以满足您自己的要求和需要。
我们希望这篇对嵌入式 Linux 领域的介绍性概述能激起您进行试验的欲望，并且希望您将体会摆弄微型设备的乐趣以满足您的爱好。为进一步有助于您的项目，请参阅下面的“参考资料”，链接到有关我们这里已经概述的技术的更深入的信息。