Unit IV
EOS

Bootloaders

- Role of a Bootloader

- The boot loader provides this early initialization code and is responsible for initializing the board so that other programs can run. This early initialization code is almost always written in the processor's native assembly language.
- It is responsible for locating, loading, and passing execution to the primary operating system. In addition, the boot loader might have advanced features, such as the capability to validate an OS image, the capability to upgrade itself or an OS image, and the capability to choose from among several OS images based on a developer-defined policy. Unlike the traditional PC-BIOS model, when the OS takes control, the bootloader is overwritten and ceases to exist.

Bootloader Challenges

- DRAM Controller
- DRAM chips cannot be directly read from or written to like other microprocessor bus resources. They require specialized hardware controllers to enable read and write cycles. To further complicate matters, DRAM must be constant refreshed or the data contained within will be lost. Refresh is accomplished by sequentially reading each location in DRAM in a systematic manner and within the timing specifications set forth by the DRAM manufacturer. Modern DRAM chips support many modes of operation, such as burst mode and dual data rate for high-performance applications.
- It is the DRAM controller's responsibility to configure DRAM, keep it refreshed within the manufacturer's timing specifications, and respond to the various read and write commands from the processor.

Bootloader Challenges

- Flash Versus RAM
- In a fully operational computer system running an operating system such as Linux, it is relatively easy to compile a program and invoke it from nonvolatile storage. The runtime libraries, operating system, and compiler work together to create the infrastructure necessary to load a program from nonvolatile storage into memory and pass control to it.

Bootloader Challenges

- Image Complexity
- As application developers, we do not need to concern ourselves with the layout of a binary executable file when we develop applications for our favorite platform. The compiler and binary utilities are pre configured to build a binary executable image containing the proper components needed for a given architecture.

Bootloader Challenges

- Execution Context
- The other primary reason for boot loader image complexity is the lack of execution context.
- resources available to the running program are nearly zero.
- Indeed, most processors have no DRAM available at startup for temporary storage of variables or, worse, for a stack that is required to use C program calling conventions.
- As a result, one of the first tasks the boot loader performs on startup is to configure enough of the hardware to enable at least some minimal amount of RAM.

- Execution Context
- Some processors designed for embedded use have small amounts of on-chip static RAM available.
- This is the case with the PPC 405GP. When RAM is available, a stack can be allocated using part of that RAM, and a proper context can be constructed to run higher-level languages such as C.

Other Bootloaders

- **Lilo**
- The Linux Loader, or Lilo, was widely used in commercial Linux distributions for desktop PC platforms; as such, it has its roots in the Intel x86/IA32 architecture. Lilo has several components.
- It has a primary bootstrap program that lives on the first sector of a bootable disk drive.
- The primary loader is limited to a disk sector size, usually 512 bytes. Therefore, its primary purpose is simply to load and pass control to a secondary loader.
- The secondary loader can span multiple

Continue..

Sample Lilo configuration : lilo.conf

```
# This is the global lilo configuration section
# These settings apply to all the "image" sections
boot = /dev/hda
timeout=50
default=linux
# This describes the primary kernel boot image
# Lilo will display it with the label 'linux'
image=/boot/myLinux-2.6.11.1
label=linux
initrd=/boot/myInitrd-2.6.11.1.img
read-only
append="root=LABEL=/"
# This is the second OS in a dual-boot configuration
# This entry will boot a secondary image from /dev/hda1
other=/dev/hda1
optional
label=that_other_os
```

Continue…

- **GRUB**
- Many current commercial Linux distributions now ship with the GRUB bootloader. GRUB, or Grand Unified Bootloader, is a GNU project.
- It has many enhanced features not found in Lilo.
- The biggest difference between GRUB and Lilo is GRUB's capability to understand file systems and kernel image formats.
- GRUB can read and modify it configuration at boot time. GRUB also supports booting across a network, which can be a tremendous asset in an embedded environment.
- GRUB offers a command line interface at boot time to modify the boot configuration

Continue...

- Like Lilo, GRUB is driven by a configuration file.
- Unlike Lilo's static configuration however, the GRUB bootloader reads this configuration at boot time.
- This means that the configured behavior can be modified at boot time for different system configurations.

Continue…

Sample GRUB configuration file :grub.config

```
default=0
timeout=3
splashimage=(hd0,1)/grub/splash.xpm.gz
title Fedora Core 2 (2.6.9)
root (hd0,1)
kernel /bzImage-2.6.9 ro root=LABEL=/ rhgb proto=imps
   quiet
initrd /initrd-2.6.9.img
title Fedora Core (2.6.5-1.358)
root (hd0,1)
kernel /vmlinuz-2.6.5-1.358 ro root=LABEL=/ rhgb quiet
title That Other OS
rootnoverify (hd0,0)
chainloader +1
```

Still More Bootloaders

- Numerous other bootloaders have found their way into specific niches.
- For example, **Redboot** is another open-source bootloader that Intel and the XScale community have adopted for use on various evaluation boards based on the Intel IXP and PXA processor families.
- **Micromonitor** is in use by board vendors such as Cogent and others.
- **YAMON** has found popularity in MIPs circles.
- **Linux BIOS** is used primarily in X86 environments

Continue..

In general, when you consider a boot loader, you should consider some important factors up front:

- Does it support my chosen processor?
- Has it been ported to a board similar to my own?
- Does it support the features I need?
- Does it support the hardware devices I intend to use?
- Is there a large community of users where I might get support?
- Are there any commercial vendors from which I can purchase support?

Device Driver Architecture

- Device drivers are broadly classified into two basic categories: *character devices and block devices.*
- *Character devices can be thought of as serial streams of* sequential data.
- Examples of character devices include serial ports and keyboards.
- Block devices are characterized by the capability to read and write blocks of data to and from random locations on an addressable medium.
- Examples of block devices include hard drives and floppy disk drives.

Minimal Device Driver Example

- Because Linux supports loadable device drivers, it is relatively easy to demonstrate a simple device driver skeleton.
- Listing illustrates a loadable device driver module that contains the bare minimum structure to be loaded and unloaded by a running kernel.

Continue....

Listing 8-1. Minimal Device Driver

```
/* Example Minimal Character Device Driver */
#include <linux/module.h>
static int __init hello_init(void)
{
printk("Hello Example Init\n");
return 0;
}
static void __exit hello_exit(void)
{
printk("Hello Example Exit\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

Module Build

The following changes were made to the stock Linux kernel source tree to enable building this sample driver. We explain each step in detail.

1. Starting from the top-level Linux source directory, create a directory under .../drivers/char called examples.
2. Add a menu item to the kernel configuration to enable building examples and to specify built-I or loadable kernel module.
3. Add the new examples subdirectory to the .../drivers/char/Makefile conditional on the menu item created in step 2.
4. Create a makefile for the new examples directory, and add the hello1.o module object to be compiled conditional on the menu item created in step 2.
5. Finally, create the driver hello1.c source file

Continue…

- The makefile for examples is quite trivial. It will contain this single line:

  obj-$(CONFIG_EXAMPLES) += hello1.o

Continue..

Listing: Makefile Patch for Examples

```
diff -u ~/base/linux-2.6.14/drivers/char/Makefile
./drivers/char/Makefile
--- ~/base/linux-2.6.14/drivers/char/Makefile
+++ ./drivers/char/Makefile
@@ -88,6 +88,7 @@
obj-$(CONFIG_DRM) += drm/
obj-$(CONFIG_PCMCIA) += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
+obj-$(CONFIG_EXAMPLES) += examples/
obj-$(CONFIG_HANGCHECK_TIMER) +=
   hangcheck-timer.o
```

Installing Your Device Driver

- Alternatively, if your target embedded system uses NFS root mount to a directory on your local development workstation, you can install the modules directly to the target file system. The following example assumes the latter.

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- \
INSTALL_MOD_PATH=/home/chris/sandbox/coyote-target modules_install
```

Loading Your Module

- **Listing 8-5. Loading and Unloading a Module**

```
# modprobe hello1                <<< Load the driver
Hello Example Init
# modprobe -r hello1             <<< Unload the driver
Hello Example Exit
#
```

- We specify the initialization function that will be executed on module insertion using the module_init() macro. We declared it as follows:

```
module_init(hello_init);
```

Module Parameters

- Many device driver modules can accept parameters to modify their behavior.
- Examples include enabling debug mode, setting verbose reporting, or specifying module-specific options.
- The insmod utility accepts parameters (also called *options in some contexts) by specifying them after the module* name.
- Listing 8-6 shows our modified hello1.c example, adding a single module parameter to enable debug mode.

Continue..

Listing: Makefile Patch for Examples

```
diff -u ~/base/linux-2.6.14/drivers/char/Makefile
./drivers/char/Makefile
--- ~/base/linux-2.6.14/drivers/char/Makefile
+++ ./drivers/char/Makefile
@@ -88,6 +88,7 @@
obj-$(CONFIG_DRM) += drm/
obj-$(CONFIG_PCMCIA) += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
+obj-$(CONFIG_EXAMPLES) += examples/
obj-$(CONFIG_HANGCHECK_TIMER) +=
   hangcheck-timer.o
```

Module Utilities

1. Insmod

- The insmod utility is the simplest way to insert a module into a running kernel.
- $ **insmod /lib/modules/2.6.14/kernel/drivers/char/examples/hello1.ko**

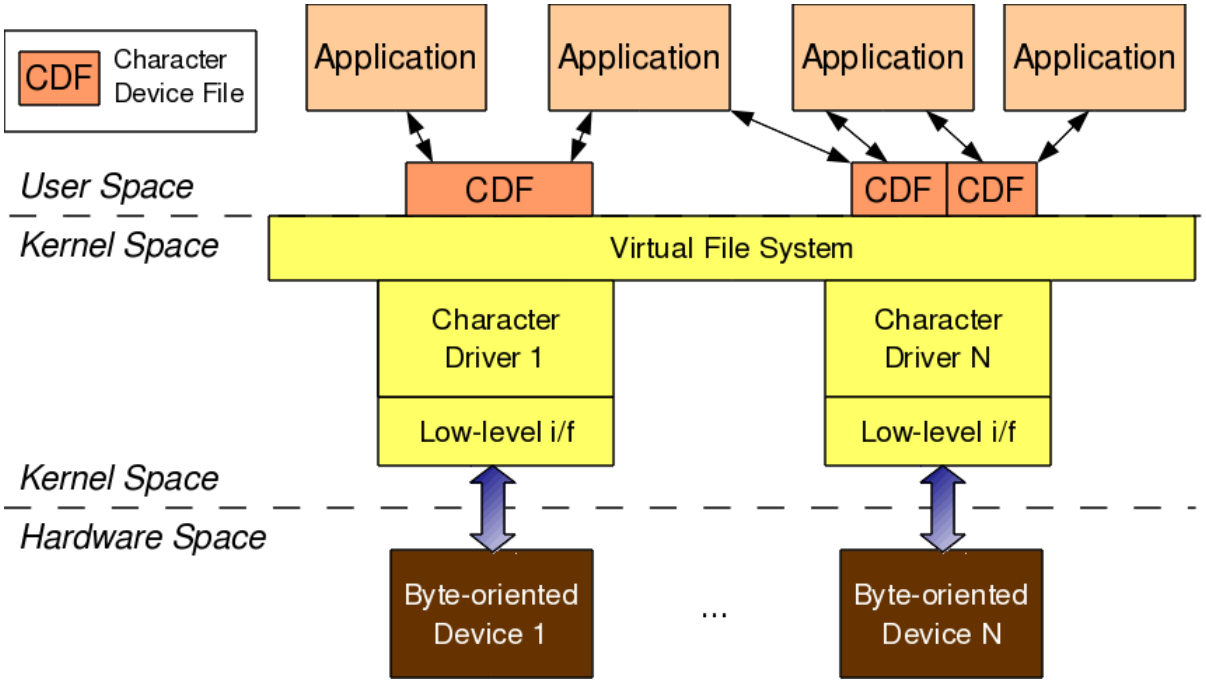
2. lsmod

- It simply displays a formatted list of the modules that are inserted into the kernel.

CHARACTER DEVICE DRIVER

We already know what are drivers and why we need them. Then, what is so special about character drivers? If we write drivers for byte-oriented operations or in the C-lingo the character-oriented operations, we refer to them as character drivers. And as the majority of devices are byte-oriented, the majority of device drivers are character device drivers. Take for example, serial drivers, audio drivers, video drivers, camera drivers, basic I/O drivers, …. In fact, all device drivers which are neither storage nor network device drivers are one form or the other form of character drivers. Let's look into the commonalities of these character drivers and how Shweta wrote one of them.

CHARACTER DEVICE DRIVER



The PCI Device Driver

Although many computer users think of PCI as a way of laying out electrical wires, it is actually a complete set of specifications defining how different parts of a computer should interact.

The PCI specification covers most issues related to computer interfaces. We are not going to cover it all here; in this section, we are mainly concerned with how a PCI driver can find its hardware and gain access to it. The PCI architecture was designed as a replacement for the ISA standard, with three main goals: to get better performance when transferring data between the computer and its peripherals, to be as platform independent as possible, and to simplify add- ing and removing peripherals to the system

The PCI Device Driver

Although many computer users think of PCI as a way of laying out electrical wires, it is actually a complete set of specifications defining how different parts of a computer should interact.

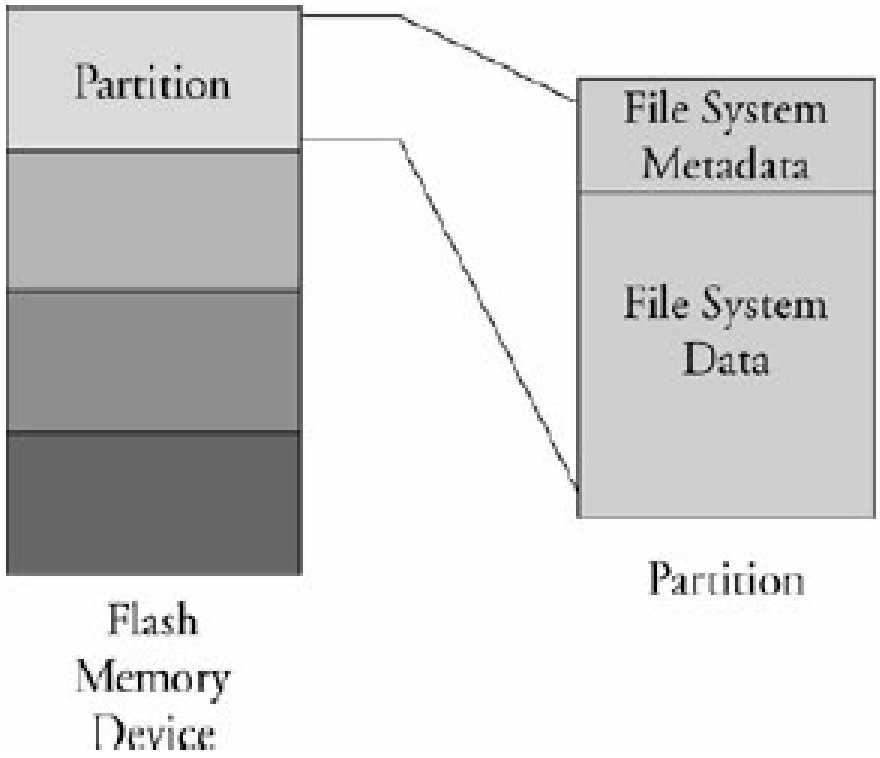
The PCI specification covers most issues related to computer interfaces. We are not going to cover it all here; in this section, we are mainly concerned with how a PCI driver can find its hardware and gain access to it. The PCI architecture was designed as a replacement for the ISA standard, with three main goals: to get better performance when transferring data between the computer and its peripherals, to be as platform independent as possible, and to simplify add- ing and removing peripherals to the system

File Systems

- **Linux File System Concepts - Partitions**
- A partition is a logical division of the physical medium (hard disk, Flash memory) whose data is organized following the specifications of a given partition type.

Figure 9-1 shows the relationship between partitions and file systems.

Figure 9-1. Partitions and file systems



- Linux uses a utility called fdisk to manipulate partitions on block devices. A recent fdisk utility found on many Linux distributions has knowledge of more than 90 different partition types.

- In practice, only a few are commonly used on Linux systems. Some common partition types include Linux, FAT32, and Linux Swap.

- Listing 9-1 displays the output of the fdisk utility targeting a CompactFlash device connected to a USB port. On this particular target system.
- On this particular target system, the USB subsystem assigned the CompactFlash physical device to the device node /dev/sdb.

- **Listing 9-1. Displaying Partition Information Using fdisk**
- # **fdisk /dev/sdb**
- Command (m for help): p
- Disk /dev/sdb: 49 MB, 49349120 bytes
- 4 heads, 32 sectors/track, 753 cylinders
- Units = cylinders of 128 * 512 = 65536 bytes
- Device Boot Start End Blocks Id System
- /dev/sdb1 * 1 180 11504 83 Linux
- /dev/sdb2 181 360 11520 83 Linux
- /dev/sdb3 361 540 11520 83 Linux
- /dev/sdb4 541 753 13632 83 Linux

- For this discussion, we have created four partitions on the device using the fdisk utility.
- When a partition is formatted with a given file system type, Linux can mount the corresponding file system from that partition.

9.2. ext2

- we need to format the partitions created with fdisk. To do so, we use the Linux mke2fs utility. mke2fs is similar to the familiar DOS format command. This utility makes a file system of type ext2 on the specified partition. mke2fs is specific to the ext2 file system;
- other file systems have their own versions of these utilities. Listing 9-2 captures the output of this process.

- **Listing 9-2. Formatting a Partition Using mke2fs**
- # **mke2fs /dev/sdb1 -L CFlash_Boot_Vol**
- mke2fs 1.37 (21-Mar-2005)
- Filesystem label=CFlash_Boot_Vol
- OS type: Linux
- Block size=1024 (log=0)
- Fragment size=1024 (log=0)
- 2880 inodes, 11504 blocks
- 575 blocks (5.00%) reserved for the super user
- First data block=1
- Maximum filesystem blocks=11796480
- 2 block groups
- 8192 blocks per group, 8192 fragments per group
- 1440 inodes per group
- Superblock backups stored on blocks:
- 8193
- Writing inode tables: done
- Writing superblocks and filesystem accounting information: done
- This filesystem will be automatically checked every 39 mounts or
- 180 days, whichever comes first. Use tune2fs -c or -i to override.
- #

- Note that this partition was formatted as type ext2 with a volume label of CFlash_Boot_Vol. It was created on a Linux partition (OS Type:) with a block size of 1024 bytes. An inode is the fundamental data structure representing a single file.

- **Mounting a File System**
- After a file system has been created, we can mount that file system on a running Linux system.
- The following command mounts the previously created ext2 file system on a mount point that we specify:
- # **mount /dev/sdb1 /mnt/flash**

- This example assumes that we have a directory created on our target Linux machine called /mnt/flash. This is called the mount point because we are installing (mounting) the file system rooted at this point in our file system hierarchy.

- Listing 9-3 displays the directory contents of a Flash device configured for an arbitrary embedded system.

- **Listing 9-3. Flash Device Listing**
- $ **ls -l /mnt/flash**
- total 24
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 bin
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 boot
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 dev
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 etc
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 home
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 lib
- drwx------ 2 root root 12288 Jul 17 13:02 lost+found
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 proc
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 root
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 sbin
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 tmp
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 usr
- drwxr-xr-x 2 root root 1024 Jul 18 20:18 var

Checking file system integrity

- The e2fsck command is used to check the integrity of an ext2 file system.
- Listing 9.4 shows the output of e2fsck run on our Compact Flash from the previous examples. It has been formatted and properly unmounted; there should be no errors

- **Listing 9-4. Clean File System Check**
- # **e2fsck /dev/sdb1**
- e2fsck 1.37 (21-Mar-2005)
- CFlash_Boot_Vol: clean, 23/2880 files, 483/11504 blocks
- #

- The e2fsck utility checks several aspects of the file system for consistency. If no issues are found, e2fsck issues a message similar to that shown in Listing 9-4. Note that e2fsck should be run only on an unmounted file system. Although it is possible to run it on a mounted file system, doing so can cause significant damage to internal file system structures on the disk or Flash device.

- We intentionally created a file and editing session on that file before removing it from the system. This can result in corruption of the data structures describing the file, as well as the actual data blocks containing the file's data.
- From Listing 9-5, you can see that e2fsck detected that the CompactFlash was not cleanly unmounted

- **Listing 9-5. Corrupted File System Check**
- # **e2fsck -y /dev/sdb1**
- e2fsck 1.37 (21-Mar-2005)
- /dev/sdb1 was not cleanly unmounted, check forced.
- Pass 1: Checking inodes, blocks, and sizes
- Inode 13, i_blocks is 16, should be 8. Fix? yes
- Pass 2: Checking directory structure
- Pass 3: Checking directory connectivity
- Pass 4: Checking reference counts
- Pass 5: Checking group summary information
- /dev/sdb1: ***** FILE SYSTEM WAS MODIFIED *****
- /dev/sdb1: 25/2880 files (4.0% non-contiguous), 488/11504 blocks

- The ext2 file system has matured as a fast, efficient, and robust file system for Linux systems.
- However, if you need the additional reliability of a journaling file system.

ReiserFS

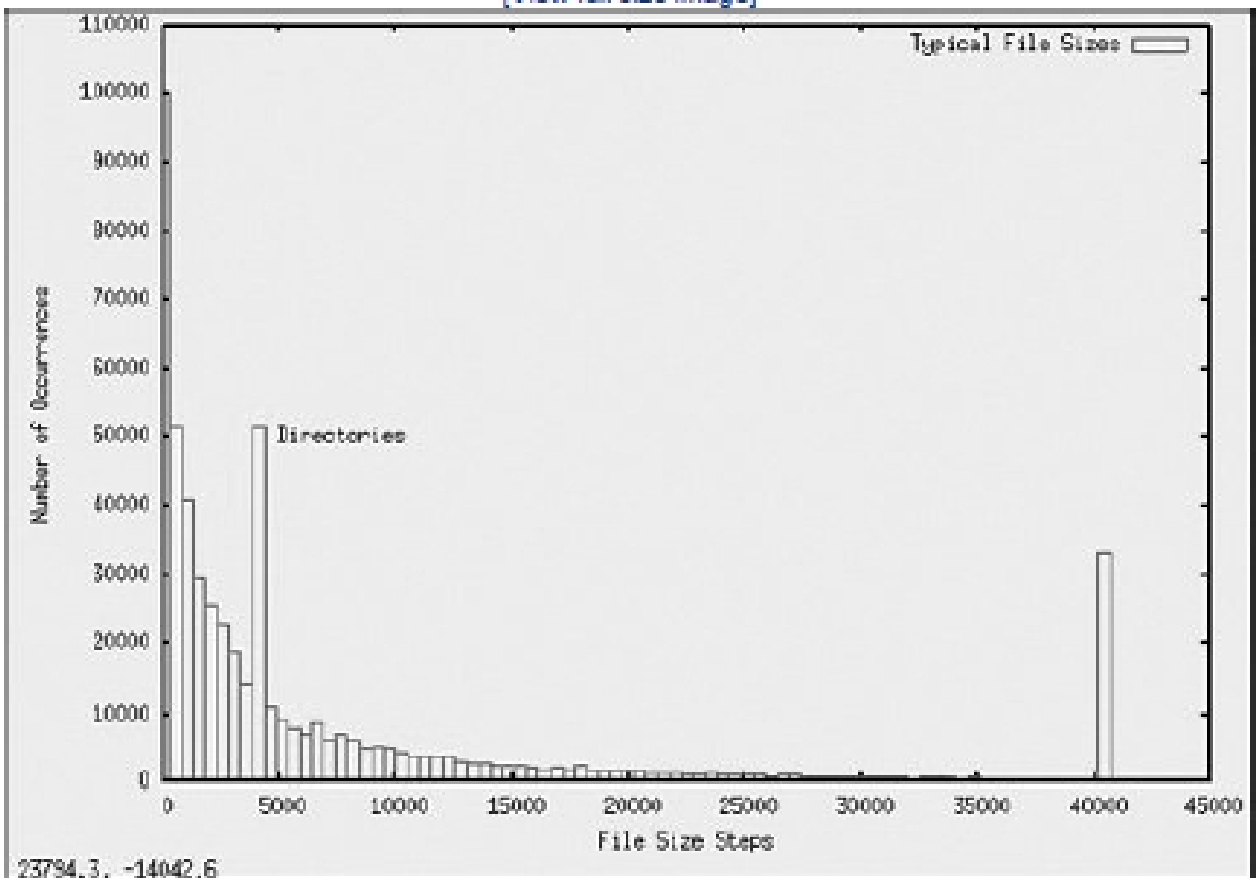
- The ReiserFS file system has enjoyed popularity among some desktop distributions such as SuSE and Gentoo.
- Unlike ext3, Reiser4 has introduced an API for system programmers to guarantee the atomicity of a file system transaction.
- Reiser4 implements high-performance "atomic" file system operations designed to protect both the state of the file system (its consistency) and the data involved in a file system operation.
- Thus guaranteeing not only that file system consistency is maintained, but that no partial data or garbage data remains in files after system crash.

JFFS2

- Unexpected power loss is a common occurrence in embedded systems.
- Enter JFFS2. These issues just discussed and other problems have been largely reduced or eliminated by the design of the second-generation Journaling Flash File System, or JFFS2.
- The original JFFS was designed by Axis Communications and was targeted specifically at the commonly available Flash memory devices at the time.
- The JFFS had knowledge of the Flash architecture and, more important, architectural limitations imposed by the devices.

Figure 9-2. File sizes in bytes

[View full size image]



Listing 9-8. Directory Layout for JFFS2 File System

```
$ ls -l
total 44
drwxr-xr-x 2 root root 4096 Aug 14 11:27 bin
drwxr-xr-x 2 root root 4096 Aug 14 11:27 dev
drwxr-xr-x 2 root root 4096 Aug 14 11:27 etc
drwxr-xr-x 2 root root 4096 Aug 14 11:27 home
drwxr-xr-x 2 root root 4096 Aug 14 11:27 lib
drwxr-xr-x 2 root root 4096 Aug 14 11:27 proc
drwxr-xr-x 2 root root 4096 Aug 14 11:27 root
drwxr-xr-x 2 root root 4096 Aug 14 11:27 sbin
drwxr-xr-x 2 root root 4096 Aug 14 11:27 tmp
drwxr-xr-x 2 root root 4096 Aug 14 11:27 usr
drwxr-xr-x 2 root root 4096 Aug 14 11:27 var
$
```

Listing 9-9. mkfs.jffs2 Command Example

```
# mkfs.jffs2 -d ./jffs2-image-dir -o jffs2.bin
# ls -l
total 4772
-rw-r--r-- 1 root root 1098640 Sep 17 22:03 jffs2.bin
drwxr-xr-x 13 root root 4096 Sep 17 22:02 jffs2-image-dir
#
```

The mkfs.jffs2 command produces a properly formatted JFFS2 file system image from a directory tree such as that in Listing 9-8. Command line parameters are used to pass mkfs.jffs2 the directory location as well as the name of the output file to receive the JFFS2 image.

The default is to create the JFFS2 image from the current directory. Listing 9-9 shows the command for building the JFFS2 image.

cramfs

The cramfs file system is very useful for embedded systems that contain a small ROM or FLASH memory that holds static data and programs. Borrowing again from the cramfs README file, "cramfs is designed to be simple and small, and compress things well.“

The cramfs file system is read only. It is created with a command line utility called mkcramfs.

mkcramfs
usage: mkcramfs [-h] [-v] [-b blksize] [-e edition] [-i file] [-n name]
dirname outfile
-h print this help
-E make all warnings errors (non-zero exit status)
-b blksize use this blocksize, must equal page size
-e edition set edition number (part of fsid)
-i file insert a file image into the filesystem (requires >= 2.4.0)
-n name set name of cramfs filesystem
-p pad by 512 bytes for boot code
-s sort directory entries (old option, ignored)
-v be more verbose
-z make explicit holes (requires >= 2.3.39)
dirname root of the directory tree to be compressed
outfile output file
#
mkcramfs . ../cramfs.image
warning: gids truncated to 8 bits (this may be a security concern)
ls -l ../cramfs.image
-rw-rw-r-- 1 chris chris 1019904 Sep 19 18:06 ../cramfs.image

Listing 9-11. Examining the cramfs File System

```
# mount -o loop cramfs.image /mnt/flash
# ls -l /mnt/flash
total 6
drwxr-xr-x 1 root root 704 Dec 31 1969 bin
drwxr-xr-x 1 root root 0 Dec 31 1969 dev
drwxr-xr-x 1 root root 416 Dec 31 1969 etc
drwxr-xr-x 1 root root 0 Dec 31 1969 home
drwxr-xr-x 1 root root 172 Dec 31 1969 lib
drwxr-xr-x 1 root root 0 Dec 31 1969 proc
drws------ 1 root root 0 Dec 31 1969 root
drwxr-xr-x 1 root root 272 Dec 31 1969 sbin
drwxrwxrwt 1 root root 0 Dec 31 1969 tmp
drwxr-xr-x 1 root root 124 Dec 31 1969 usr
drwxr-xr-x 1 root root 212 Dec 31 1969 var
#
```

The cramfs file system is ideal for boot ROMS, in which read-only operation and fast compression are ideally suited.

Network File System

NFS enables you to export a directory on an NFS server and mount that directory on a remote client machine as if it were a local file system.

Using NFS on your target board, an embedded developer can have access to a

huge number of files, libraries, tools, and utilities during development and debugging, even if the target embedded system is resource constrained

Network File System

Listing 9-12. Contents of /etc/exports

```
$ cat /etc/exports
# /etc/exports
/home/chris/sandbox/coyote-target
*(rw,sync,no_root_squash)
/home/chris/workspace
*(rw,sync,no_root_squash)
$
```

On an embedded system with NFS enabled, the following command mounts the .../workspace directory exported by the NFS server on a mount point of our choosing:

$ **mount -t nfs pluto:/home/chris/workspace /workspace**

The entry in the target's /etc/hosts file would look like this:

192.168.10.9 pluto

Root File System on NFS

Listing 9-13. Target File System Example Summary

$ **du -h --max-depth=1**

724M ./usr

4.0K ./opt

39M ./lib

12K ./dev

27M ./var

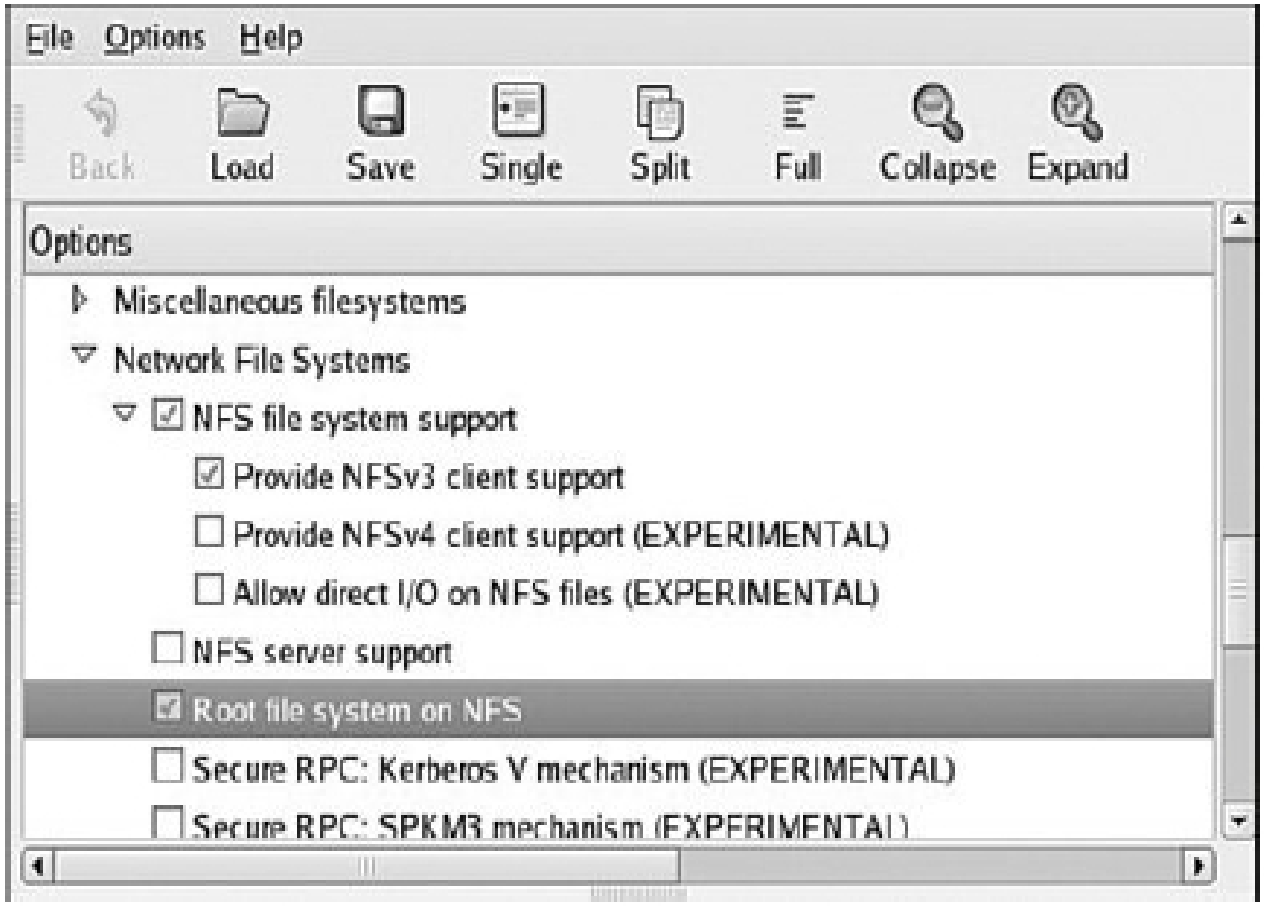
4.0K ./tmp

3.6M ./boot

4.0K ./workspace

1.8M ./etc

```
4.0K ./home
4.0K ./mnt
8.0K ./root
29M ./bin
32M ./sbin
4.0K ./proc
64K ./share
855M .
$
$ find -type f | wc -l
29430
```


File Options Help
Back
Load
Save
Single
Split
Full
Collapse
Expand
Options
Miscellaneous filesystems
Network File Systems
NFS file system support
Provide NFSv3 client support
Provide NFSv4 client support (EXPERIMENTAL)
Allow direct I/O on NFS files (EXPERIMENTAL)
NFS server support
Root file system on NFS
Secure RPC: Kerberos V mechanism (EXPERIMENTAL)
Secure RPC: SPKM3 mechanism (EXPERIMENTAL)

A typical kernel command line might look like this:

console=ttyS0,115200 ip=bootp root=/dev/nfs

If you are statically configuring your target's IP address, your kernel command line might look like this

console=ttyS0,115200 \

ip=192.168.1.139:192.168.1.1:192.168.1.1:255.255.255.0:coyote1:eth0:off \

nfsroot=192.168.1.1:/home/chris/sandbox/coyote-target \

root=/dev/nfs

ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<PROTO>

Pseudo File Systems

1. **Proc File System**

The /proc file system took its name from its original purpose, an interface that allows the kernel to communicate information about each running process on a Linux system.

Listing 9-14. Mount Dependency on /proc

```
# mount
rootfs on / type rootfs (rw)
/dev/root on / type nfs
(rw,v2,rsize=4096,wsize=4096,hard,udp,nolock,addr=1
92.168.1.19)
tmpfs on /dev/shm type tmpfs (rw)
/proc on /proc type proc (rw,nodiratime)
< Now unmount proc and try again ...>
# umount /proc
# mount
#
```

To mount the /proc file system, use the mount command as with any other file system:

$ **mount -t proc /proc /proc**

The general form of the mount command, from the man page, is

$ **mount [-t fstype] something somewhere**.

In the previous invocation, we could have substituted none for /proc, as follows:

$ **mount -t proc none /proc**

Listing 9-15. init Process /proc EnTRies

```
# ls -l /proc/1
total 0
-r-------- 1 root root 0 Jan 1 00:25 auxv
-r--r--r-- 1 root root 0 Jan 1 00:21 cmdline
lrwxrwxrwx 1 root root 0 Jan 1 00:25 cwd -> /
-r-------- 1 root root 0 Jan 1 00:25 environ
lrwxrwxrwx 1 root root 0 Jan 1 00:25 exe -> /sbin/init
dr-x------ 2 root root 0 Jan 1 00:25 fd
-r--r--r-- 1 root root 0 Jan 1 00:25 maps
-rw------- 1 root root 0 Jan 1 00:25 mem
-r--r--r-- 1 root root 0 Jan 1 00:25 mounts
```

-rw-r--r-- 1 root root 0 Jan 1 00:25 oom_adj

-r--r--r-- 1 root root 0 Jan 1 00:25 oom_score

lrwxrwxrwx 1 root root 0 Jan 1 00:25 root -> /

-r--r--r-- 1 root root 0 Jan 1 00:21 stat

-r--r--r-- 1 root root 0 Jan 1 00:25 statm

-r--r--r-- 1 root root 0 Jan 1 00:21 status

dr-xr-xr-x 3 root root 0 Jan 1 00:25 task

-r--r--r-- 1 root root 0 Jan 1 00:25 wchan

Listing 9-16. init Process Memory Segments from /proc

cat /proc/1/maps

00008000-0000f000 r-xp 00000000 00:0a 9537567 **/sbin/init**

00016000-00017000 rw-p 00006000 00:0a 9537567 **/sbin/init**

00017000-0001b000 rwxp 00017000 00:00 0

40000000-40017000 r-xp 00000000 00:0a 9537183 **/lib/ld-2.3.2.so**

40017000-40018000 rw-p 40017000 00:00 0

4001f000-40020000 rw-p 00017000 00:0a 9537183 **/lib/ld-2.3.2.so**

40020000-40141000 r-xp 00000000 00:0a 9537518 **/lib/libc-2.3.2.so**

40141000-40148000 ---p 00121000 00:0a 9537518 **/lib/libc-2.3.2.so**

40148000-4014d000 rw-p 00120000 00:0a 9537518 **/lib/libc-2.3.2.so**

4014d000-4014f000 rw-p 4014d000 00:00 0

befeb000-bf000000 rwxp befeb000 00:00 0

#

You can also see the memory segments used by the shared library objects being used by the init process. The format is as follows:

vmstart-vmend attr pgoffset devname inode filename

MTD Subsystem

- **MTD Overview**

- MTD is a device driver layer that provides a uniform API for interfacing with raw Flash devices.
- MTD devices have a limited write life cycle, and MTD has logic to spread the write operations over the devices life span to increase the device’s life span.This is called *wear leveling.*

- **Enabling MTD Services**

- To illustrate the mechanics of the MTD subsystem and how it fits in with the system, we begin with some very simple examples that you can perform on your Linux development workstation.
- Figure 10-1 shows the kernel configuration (invoked per the usual make ARCH=<arch> gconfig) necessary to enable the bare-minimum MTD functionality.
- Listing 10-1 displays the .config file entries resulting from the selections shown in Figure 10-1.

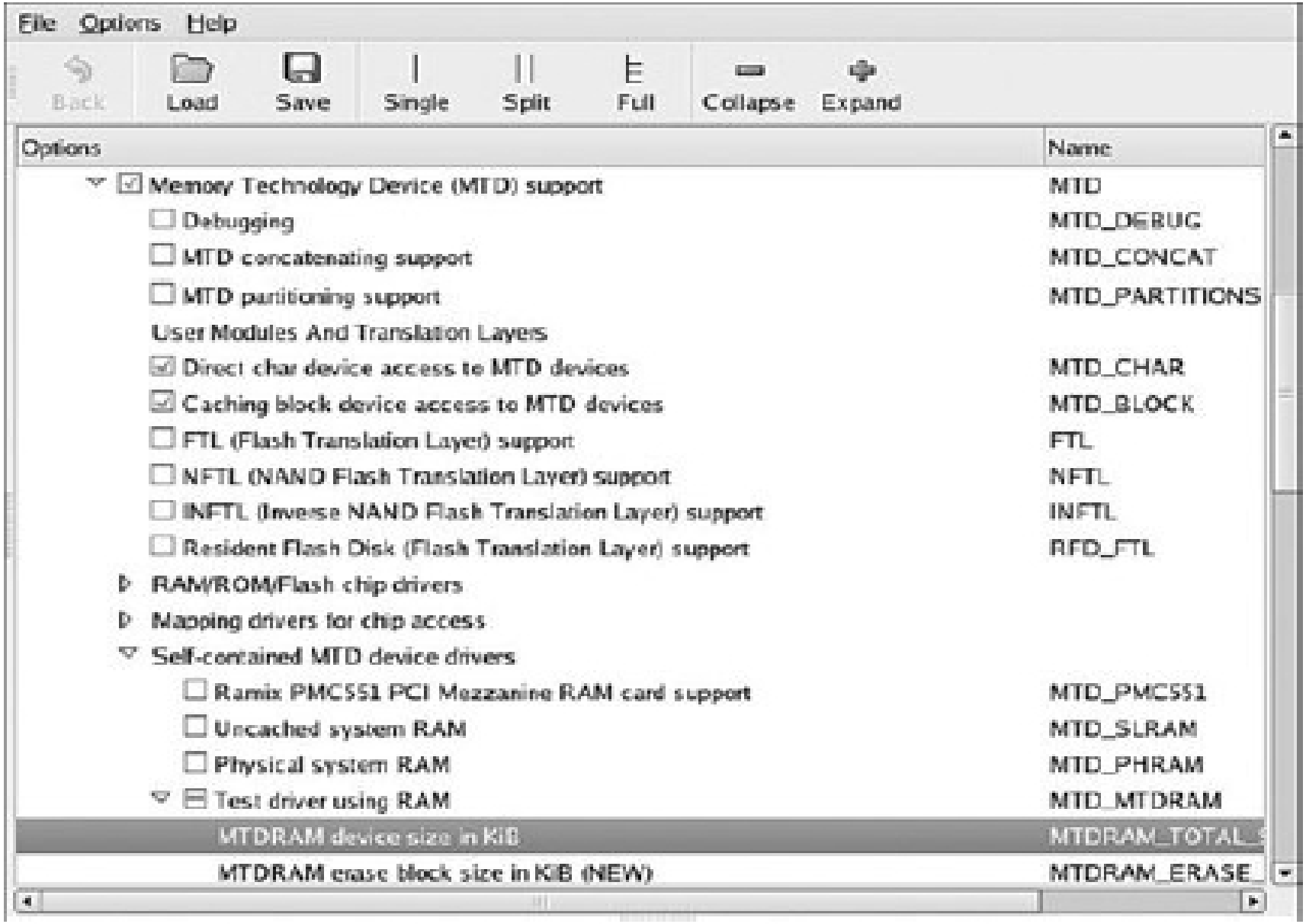


Fig. MTD Configuration

Continue…

- **Listing 10-1. Basic MTD Configuration from .config**
- CONFIG_MTD=y
- CONFIG_MTD_CHAR=y
- CONFIG_MTD_BLOCK=y
- CONFIG_MTD_MTDRAM=m
- CONFIG_MTDRAM_TOTAL_SIZE=8192
- CONFIG_MTDRAM_ERASE_SIZE=128

- The MTD subsystem is enabled via the first configuration option, which is selected via the first check box shown in Figure 10-1, Memory Technology Device (MTD) Support. The next two entries from the configuration shown in Figure 10-1 enable special device-level access to the MTD devices, such as Flash memory, from user space.
- The first one (CONFIG_MTD_CHAR) enables character device mode access, essentially a sequential access characterized by byte-at-a-time sequential read and write access.

- The second (CONFIG_MTD_BLOCK) enables access to the MTD device in block device mode, the access method used for disk drives, in which blocks of multiple bytes of data are read or written at a time.
- These access modes allow the use of familiar Linux commands to read and write data to the Flash memory.
- The CONFIG_MTD_MTDRAM element enables a special test driver that enables us to examine the MTD subsystem even if we don't have any MTD devices (such as Flash memory) available.

- **MTD Basics**
- Now that we have enabled a simple MTD configuration in our kernel, we can examine how this subsystem works on our Linux development workstation. Using the test RAM driver we just configured in the previous section, we can mount a JFFS2 image using an MTD device.
- The Linux kernel does not support mounting a JFFS2 file system image directly on a loopback device, such as is possible with ext2 and other file system images. So we must use a different method. This can be achieved using the MTD RAM test driver on our development Linux workstation with MTD enabled Listing 10-3 illustrates the steps.

- **Listing 10-3. Mounting JFFS2 on an MTD RAM Device**
- # **modprobe jffs2**
- # **modprobe mtdblock**
- # **modprobe mtdram**
- # **dd if=jffs2.bin of=/dev/mtdblock0**
- 4690+1 records in
- 4690+1 records out
- # **mkdir /mnt/flash**
- # **mount -t jffs2 /dev/mtdblock0/mnt/flash**
- # **ls -l /mnt/flash**
- total 0
- drwxr-xr-x 2 root root 0 Sep 17 22:02 bin
- drwxr-xr-x 2 root root 0 Sep 17 21:59 dev
- drwxr-xr-x 7 root root 0 Sep 17 15:31 etc
- drwxr-xr-x 2 root root 0 Sep 17 15:31 home
- drwxr-xr-x 2 root root 0 Sep 17 22:02 lib
- drwxr-xr-x 2 root root 0 Sep 17 15:31 proc
- drws------ 2 root root 0 Sep 17 15:31 root
- drwxr-xr-x 2 root root 0 Sep 17 22:02 sbin
- drwxrwxrwt 2 root root 0 Sep 17 15:31 tmp
- drwxr-xr-x 9 root root 0 Sep 17 15:31 usr
- drwxr-xr-x 14 root root 0 Sep 17 15:31 var
- #

- From Listing 10-3, first we install the loadable modules that the Linux kernel requires to support JFFS2 and the MTD subsystem. We load the JFFS2 module followed by the mTDblock and mtdram modules.
- After the necessary device drivers are loaded, we use the Linux dd command to copy our JFFS2 file system image into the MTD RAM test driver using the mTDblock device. In essence, we are using system RAM as a backing device to emulate an MTD block device.

Continue…

- **Listing 10-1. Basic MTD Configuration from .config**
- CONFIG_MTD=y
- CONFIG_MTD_CHAR=y
- CONFIG_MTD_BLOCK=y
- CONFIG_MTD_MTDRAM=m
- CONFIG_MTDRAM_TOTAL_SIZE=8192
- CONFIG_MTDRAM_ERASE_SIZE=128

- It is important to realize the limitations of using this method to examine the contents of a JFFS2 file system. Consider what we did: We copied the contents of a file (the JFFS2 file system binary image) into a kernel block device (/dev/mtdblock0). Then we mounted the kernel block device (/dev/mtdblock) as a JFFS2 file system.
- After we did this, we could use all the traditional file system utilities to examine and even modify the file system. Tools such as ls,df,dh,mv,rm, and cp can all be used to examine and modify the file system. However, unlike the loopback device, there is no connection between the file we copied and the mounted JFFS2 file system image.
- Therefore, if we unmount the file system after making changes, the changes will be lost. If you want to save the changes, you must copy them back into a file. One such method is the following:

- **# dd if=/dev/mtdblock0 of=./your-modified-fs-image.bin**

- This command creates a file called your-modified-fs-image.bin that is the same size as the mtdblock0 device which was specified during configuration.

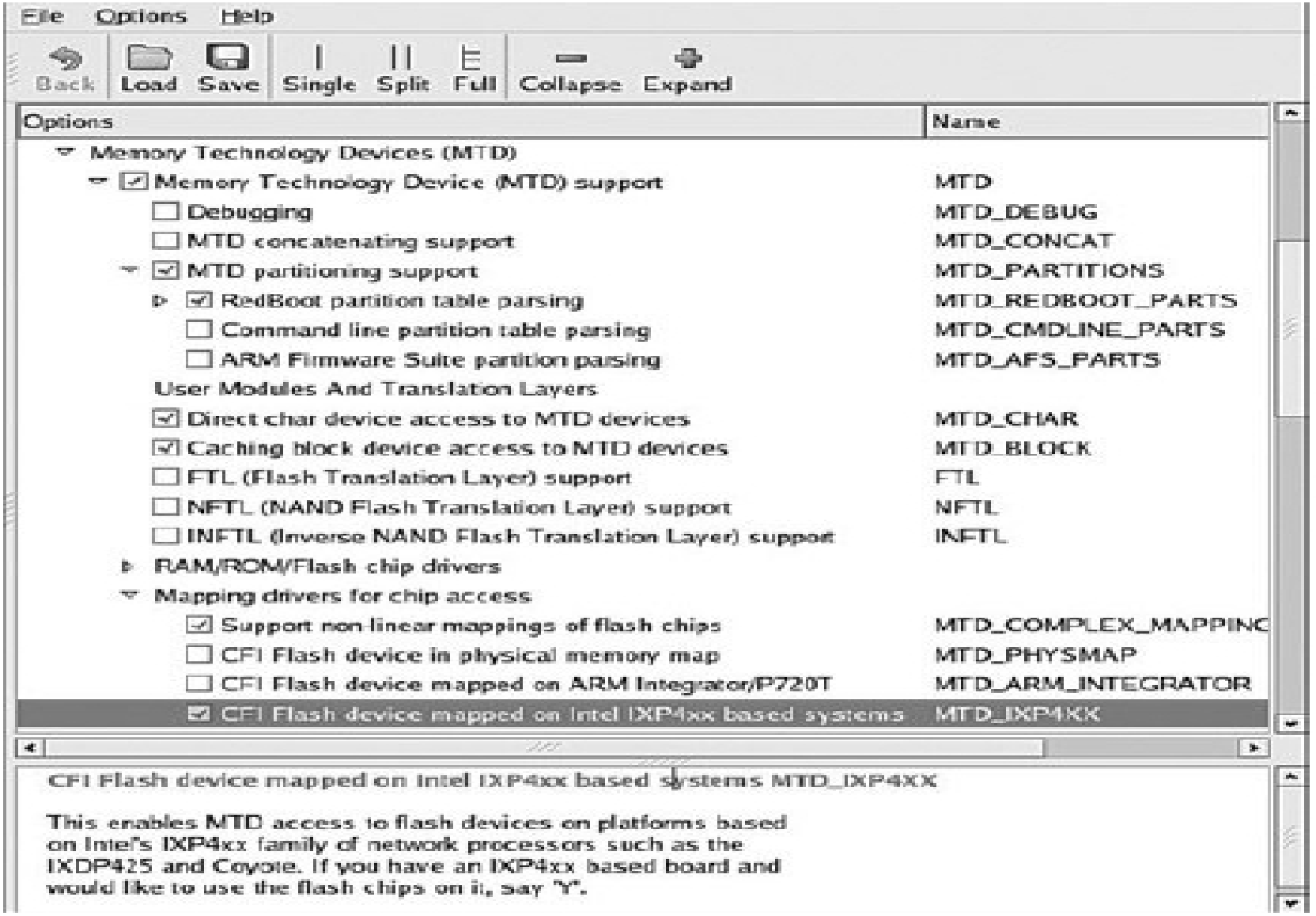
- **Configuring MTD on your Target**

- To use MTD with the Flash memory on your board, you must have MTD configured correctly. The following list contains the requirements that must be satisfied to configure MTD for your board, Flash, and Flash layout.
- **Specify the partitioning on your Flash device**
- **Specify the type of Flash and location**
- **Configure the proper Flash driver for your chosen chip**
- **Configure the kernel with the appropriate driver(s)**

- Each of these steps is explored in the following

MTD Partitions

- Most Flash devices on a given hardware platform are divided into several sections, called partitions, similar to the partitions found on a typical desktop workstation hard drive. The MTD subsystem provides support for such Flash partitions. The MTD subsystem must be configured for MTD partitioning support.
- Figure 10-2 illustrates the configuration options for MTD partitioning support.


File Options Help
Back Load Save Single Split Full Collapse Expand
Options
Name
Memory Technology Devices (MTD)
Memory Technology Device (MTD) support
MTD
Debugging
MTD_DEBUG
MTD concatenating support
MTD_CONCAT
MTD partitioning support
MTD_PARTITIONS
RedBoot partition table parsing
MTD_REDBOOT_PARTS
Command line partition table parsing
MTD_CMDLINE_PARTS
ARM Firmware Suite partition parsing
MTD_AFS_PARTS
User Modules And Translation Layers
Direct char device access to MTD devices
MTD_CHAR
Caching block device access to MTD devices
MTD_BLOCK
FTL (Flash Translation Layer) support
FTL
NFTL (NAND Flash Translation Layer) support
NFTL
INFTL (Inverse NAND Flash Translation Layer) support
INFTL
RAM/ROM/Flash chip drivers
Mapping drivers for chip access
Support non-linear mappings of flash chips
MTD_COMPLEX_MAPPING
CFI Flash device in physical memory map
MTD_PHYSMAP
CFI Flash device mapped on ARM Integrator/P720T
MTD_ARM_INTEGRATOR
CFI Flash device mapped on Intel IXP4xx based systems
MTD_IXP4XX
CFI Flash device mapped on Intel IXP4xx based systems MTD_IXP4XX
This enables MTD access to flash devices on platforms based on Intel's IXP4xx family of network processors such as the IXDP425 and Coyote. If you have an IXP4xx based board and would like to use the flash chips on it, say 'Y'.

Continue..

- Several methods exist for communicating the partition data to the Linux kernel. The following methods are currently supported. You can see the configuration options for each in Figure 10-2 under MTD Partition in Support.
- Redboot partition table parsing
- Kernel command-line partition table definition
- Board-specific mapping drivers
- T1 AR7 partitioning support
- MTD also allows configurations without partition data. In this case, MTD simply treats the entire Flash memory as a single device.

Redboot Partition Table Partitioning

- In the case of the Redboot partitions, the developer reserves and specifies a Flash erase block that holds the partition definitions. A mapping driver is selected that calls the partition parsing functions during boot to detect the partitions on the Flash device.
- Figure 10-2 shows the mapping driver for our example board; it is the final highlighted entry defining CONFIG_MTD_IXP4xx.

Continue…

- Listing 10-4 captures some of the output of the Redboot bootloader upon power-up.
- **Listing 10-4. Redboot Messages on Power-Up**
- Platform: ADI Coyote (XScale)
- IDE/Parallel Port CPLD Version: 1.0
- Copyright (C) 2000, 2001, 2002, Red Hat, Inc.
- RAM: 0x00000000-0x04000000, 0x0001f960-0x03fd1000 available
- FLASH: 0x50000000 - 0x51000000, 128 blocks of 0x00020000 bytes each.

- Redboot contains a command to create and display partition information on the Flash. Listing 10-5 contains the output of the fis list command, part of the Flash Image System family of commands available in the Redboot bootloader.
- **Listing 10-5. Redboot Flash Partition List**
- RedBoot> **fis list**
- Name FLASH addr Mem addr Length Entry point
- RedBoot 0x50000000 0x50000000 0x00060000 0x00000000
- RedBoot config 0x50FC0000 0x50FC0000 0x00001000 0x00000000
- FIS directory 0x50FE0000 0x50FE0000 0x00020000 0x00000000
- RedBoot>

- From Listing 10-5, we see that the Coyote board has three partitions defined on the Flash. The partition named RedBoot contains the executable Redboot bootloader image. The partition named RedBoot config contains the configuration parameters maintained by the bootloader.
- The final partition named FIS directory holds information about the partition table itself. When properly configured, the Linux kernel can detect and parse this partition table and create MTD partitions representing the physical partitions on Flash.
- Listing 10-6 reproduces a portion of the boot messages that are output from the aforementioned ADI Engineering Coyote board, booting a Linux kernel configured with support for detecting Redboot partitions.

- From Listing 10-5, we see that the Coyote board has three partitions defined on the Flash. The partition named RedBoot contains the executable Redboot bootloader image.
- The partition named RedBoot config contains the configuration parameters maintained by the bootloader.
- The final partition named FIS directory holds information about the partition table itself

- . **Listing 10-6. Detecting Redboot Partitions on Linux Boot**
- ...
- IXP4XX-Flash0: Found 1 x16 devices at 0x0 in 16-bit bank
- Intel/Sharp Extended Query Table at 0x0031
- Using buffer write method
- cfi_cmdset_0001: Erase suspend on write enabled
- Searching for RedBoot partition table in IXP4XX-Flash0 at offset 0xfe0000
- 3 RedBoot partitions found on MTD device IXP4XX-Flash0
- Creating 3 MTD partitions on "IXP4XX-Flash0":
- 0x00000000-0x00060000: "RedBoot"
- 0x00fc0000-0x00fc1000: "RedBoot config"
- 0x00fe0000-0x01000000: "FIS directory"

Continue..

- Several methods exist for communicating the partition data to the Linux kernel. The following methods are currently supported. You can see the configuration options for each in Figure 10-2 under MTD Partition in Support.
- Redboot partition table parsing
- Kernel command-line partition table definition
- Board-specific mapping drivers
- T1 AR7 partitioning support
- MTD also allows configurations without partition data. In this case, MTD simply treats the entire Flash memory as a single device.

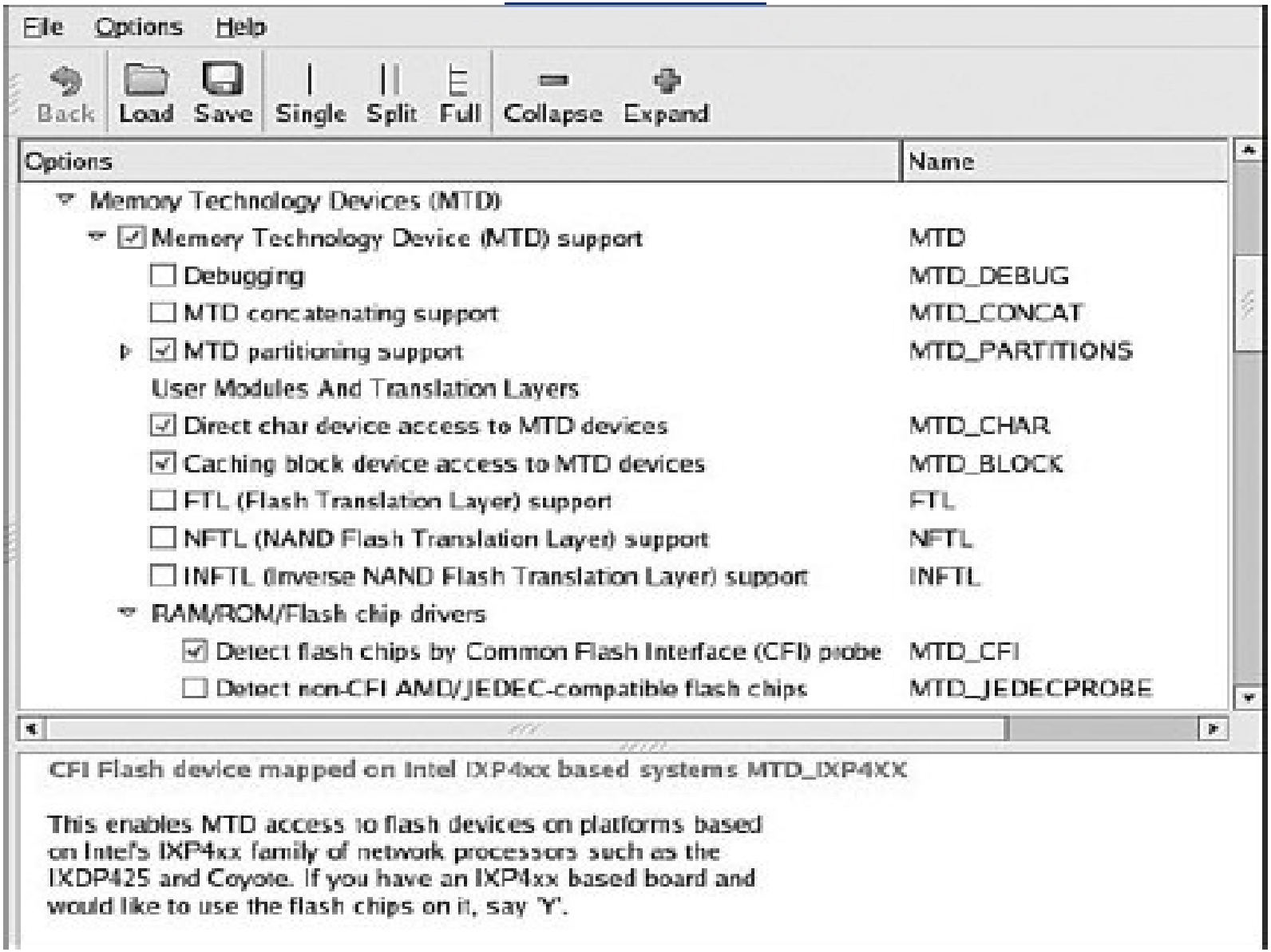


Fig. Kernel configuration for MTD CFI support

- As shown in Listing 10-6, the Flash chip is detected via the CFI interface.
- Because we also enable CONFIG_MTD_REDBOOT_PARTS (see Figure 10-2), MTD scans for the Redboot partition table on the Flash chip. Notice also that the chip has been enumerated with the device name IXP4XX-Flash0.
- You can see from Listing 10-6 that the Linux kernel has detected three partitions on the Flash chip, as enumerated previously using the fis list command in Redboot.

- First, we load the image we will use to create the new partition. We will use our kernel image for the example. We load it to memory address 0x01008000. Then we create the new partition using the Redboot fis create command.
- We have instructed Redboot to create the new partition in an area of Flash starting at 0x50100000. You can see the action as Redboot first erases this area of Flash and then programs the kernel image.
- In the final sequence, Redboot unlocks its directory area and updates the FIS Directory with the new partition information.
- Listing 10-9 shows the output of fis list with the new partition.