Contents

1 Overview .................................................................................................................................................. 1
   1.1 Introduction .................................................................................................................................. 1
   1.2 Block Data and TrueFFS ........................................................................................................... 3
       1.2.1 How TrueFFS Manages Block Data ................................................................................. 3
       1.2.2 Virtual Blocks Make Wear Leveling Possible ............................................................... 4
       1.2.3 Virtual Blocks Simplify Garbage Collection ................................................................. 4
       1.2.4 How Garbage Collection is Triggered ............................................................................ 5
       1.2.5 Block Allocation and the Clustering of Related Data ....................................................... 5
   1.3 Fault Recovery in TrueFFS ........................................................................................................... 6
       1.3.1 “Erase after Write” Guarantees Data Integrity ............................................................... 6
       1.3.2 Recovering from Failures During Write or Erase Operations ....................................... 7
       1.3.3 Recovering from Failures During Garbage Collection ................................................... 7
       1.3.4 Recovering from Failures During Formatting ................................................................. 8
   1.4 Sharing Flash Memory Between TrueFFS and a Boot Image ..................................................... 8
   1.5 TrueFFS is a Layered Product ..................................................................................................... 10
   1.6 TrueFFS Configuration Display Utilities .................................................................................... 11
1.7 About I/O Sequencing ................................................................................. 11

2 Getting Started .............................................................................................. 13
2.1 Introduction .................................................................................................. 13
2.2 An Overview of Configuring and Using TrueFFS ..................................... 14
  2.2.1 Configuring VxWorks and TrueFFS ..................................................... 14
    Changing the Project ............................................................................... 15
    Changing the configuration ................................................................. 15
    Changing sysLib.c ............................................................................. 15
    Changing sysTffs.c ........................................................................... 15
  2.2.2 Formatting Flash ............................................................................... 19
  2.2.3 Creating TrueFFS Block Devices ..................................................... 21

2.3 BSP-Specific Examples of Using TrueFFS ............................................... 23
  2.3.1 ads860 with a Board-Resident Flash Array and a PCMCIA Slot .......... 23
  2.3.2 hkbaja47 a Board-Resident Flash Array .......................................... 25
  2.3.3 iq960rp with a Board-Resident Flash Array ...................................... 25
  2.3.4 mv177 with a Board-Resident Flash Array ...................................... 26
  2.3.5 pid7t or pid7t_t with a Board-Resident Flash Array ......................... 27
  2.3.6 ss5 with a Board-Resident Flash Array .......................................... 27

2.4 Socket Layer Options for the pc386 and pc486 BSPs ................................. 28
  2.4.1 Configuring the pc*86 for TrueFFS and the PCMCIA Driver ........... 29
  2.4.2 x86 w/DiskOnChip & 2 PCMCIA Slots Using INCLUDE_PCMCIA 31
       Adding Boot Code to the DiskOnChip 2000 ........................................ 32
       X-86 Pitfalls—CMOS Setup ............................................................. 34

3 Writing Socket Component Drivers and MTDs ......................................... 35
3.1 Introduction .................................................................................................. 35
3.2 About the FLFlash and FLSocket Structures ......................................... 37
1 Overview

1.1 Introduction

TrueFFS for Tornado is a VxWorks-compatible implementation of M-Systems FLite, Version 2.0. It provides a block device interface to a wide variety of flash memory devices. This system is reentrant, thread-safe, and supported on most of the more popular CPU architectures.

Using TrueFFS for Tornado, applications can read and write from flash memory just as they would from an MS-DOS file system resident on a magnetic-medium mechanical disk drive. This is despite the fact that the underlying storage media are radically different. TrueFFS for Tornado hides these differences from the high-level developer.

However, when designing an embedded system, it is critical that you be aware of how flash memory compares to mechanical magnetic-media disk drives as a storage device. Although flash memory might not be practical for all embedded systems, there are many environments in which its unique qualities make it the ideal choice.

About Flash Memory as a Storage Medium

Flash memory stores data indefinitely, even while unpowered. In addition, the physical components of flash memory are solid-state devices (no moving parts) that consume little energy and take up little space. Thus, flash memory is ideal for mobile devices, hand-held devices, and any other embedded system that needs reliable non-volatile data storage in environments too harsh for mechanical disks.

However, the physical characteristics of flash memory differ considerably from the more common magnetic-medium mechanical disks. When using magnetic storage
media, overwriting a previously written storage area simply obliterates the existing data. Such is not the case with flash memory. Flash memory requires a separate erase cycle before new data can be written to an area. Using the erase cycle is further complicated by the fact that one cannot erase individual bytes or even blocks. Instead, the underlying flash technology requires that you erase entire regions known as erase units. The size of these erase units depends on the specific flash technology, but a typical size is 64KB.

Flash also limits how often you can erase and rewrite the same area. This number, (known as the cycling limit) depends on the specific flash technology, but it ranges from a hundred thousand to a million times per block. As a region of flash approaches its cycling limit, it begins to suffer from sporadic erase failures. With further use, these erase failures become more and more frequent. Eventually, the medium is no longer erasable and thus no longer writable, although the data already resident is still readable.

In addition to the cycling limit, there is also a limit on immediate successive erase/rewrite cycles. Constantly erasing and rewriting the same area puts flash memory into an over-programmed state. In this state, flash responds only very slowly to write requests. With rest, this condition eventually fixes itself, but the life of the flash memory is shortened.

Finally, using flash memory is further complicated by the fact that writing to flash is not a simple physical act. In fact, storing data in flash memory requires the use of a programming algorithm (defined in a software module known as an MTD) that differs depending on the manufacturer (and, sometimes, the device family) of the flash memory.2

Data Integrity and Removable Flash Media

A robust block driver running over flash memory must deal with all the complexities associated with removable media (flash cards). For example, it is entirely possible that a user could pull out a flash card while the driver was in the middle of a write. Such an interruption could be catastrophic if the data being written happens to be the directory structure. Fortunately, TrueFFS has been carefully designed to take into account all the peculiarities of flash memory, power losses, and impatient or curious users who yank out flash cards just to see what happens.

1. This number is merely statistical in nature and should not be taken as an exact figure.
2. Because writing data to flash memory requires the use of a manufacturer-supplied programming algorithm, writing flash is sometimes referred to as programming flash.
1.2 Block Data and TrueFFS

Flash memory is not an infinitely reusable storage medium. The number of erase cycles per erase unit of flash memory is large but limited. Thus, with use, flash eventually degrades to a read-only state. As failure modes go, this one is uncommonly kind to data. Still, even if this failure mode is not catastrophic, it is best to delay it for as long as possible.

To maximize the lifetime of flash, it is necessary to balance usage over the entire medium so that no one part of flash is over-used and driven prematurely into a read-only state. This technique is known as wear leveling. The technique of wear leveling also helps avoid over-programming. To implement wear leveling, TrueFFS uses a block-to-flash translation system that is based on a dynamically maintained map. This map is adjusted as blocks are modified, moved, or garbage collected.

1.2.1 How TrueFFS Manages Block Data

As is required of a block device driver, TrueFFS maps flash memory into an apparently contiguous array of storage blocks to which a file system can read and write data. These blocks are numbered from zero to one less than the total number of blocks.

Reading the data from a block is straightforward. The file system requests the contents of a particular block. In response, TrueFFS translates the block number into flash memory coordinates, retrieves the data at those coordinates, and returns the data to the file system. Writing data to a block can be just as straightforward if the target block is as yet unwritten. All TrueFFS needs to do is translate the block number into flash memory coordinates and write to the location.\(^3\)

However, if the write request seeks to modify the contents of a previously written block, the situation is more complex. Flash memory requires an erase cycle before it can be overwritten, and the erase cycle applies on a scale much larger than a single block. Thus, to simulate overwriting a particular block, TrueFFS first finds a writable area of flash (one that is already erased and ready to receive data). TrueFFS then writes the modified block data to that free area. After the data is safely written, TrueFFS updates its block-to-flash mapping structures so that the block now maps to the area of flash that contains the modified data.

---

3. Writing data to flash does require the mediation of a software module called an MTD, but that detail is not immediately relevant.
1.2.2 Virtual Blocks Make Wear Leveling Possible

The remapping of modified blocks to new flash memory coordinates does guarantee a certain degree of wear leveling. However, if some of the data stored in flash is essentially static, wear leveling only at modification time gives rise to a problem known as static file locking.

The areas of flash that store static data would not be cycled at all, at the expense of the remaining areas. This would lower the medium’s life-expectancy significantly. TrueFFS overcomes static file locking by forcing transfers of static areas. Because the block-to-flash map is dynamic, TrueFFS is able to manage these wear-leveling transfers in a way that is invisible to the file system.

However, implementing absolute wear-leveling can have a negative impact on performance because of all the required data moves. To avoid this performance hit, TrueFFS implements a wear-leveling algorithm that is not quite absolute. The result is that all erase zones are used approximately equally. In the long run, you can expect the same number of erase cycles per erase unit without a degradation of performance. Given the large number of allowed erase cycles, this less than absolute approach to wear-leveling is good enough.

Finally, the TrueFFS wear leveling algorithm is further enhanced to break a failure mode known as dead locks. Some simple wear-leveling algorithms have been shown to “flip-flop” transfers between only two or more units for very long periods, neglecting all other units. The wear-leveling mechanism used by TrueFFS makes sure that such loops are curbed.4

1.2.3 Virtual Blocks Simplify Garbage Collection

Modifying blocks leaves behind block-sized regions of flash memory that no longer contain valid data. These regions of flash memory are also unwritable until erased. If there were no mechanism to reclaim these blocks, the flash medium would quickly revert to a read-only state. Unfortunately, reclaiming these blocks is complicated by the fact that blocks are not individually erasable. Erasure is limited to much larger regions (for example, 64 kilobytes for Intel Flash File devices) known as erase units.

To reclaim (erase) blocks that no longer contain valid data, TrueFFS uses a mechanism called garbage collection. This mechanism copies all the valid data blocks from a source erase unit into another erase unit known as a transfer unit. TrueFFS then updates the block-to-flash map and then erases the old erase unit.

4. For optimal wear-leveling performance, TrueFFS requires at least 12 erase units.
The virtual block presented to the outside world still appears to contain the same data even though that data now resides in a different part of flash.

### 1.2.4 How Garbage Collection is Triggered

If garbage collection is done too frequently, it defeats the wear-leveling algorithm and degrades the overall performance of the flash disk. Thus, within TrueFFS, garbage collection is triggered only as needed by the block allocation algorithm, which tries to keep available a pool of free consecutive blocks resident in the same erase unit. If the pool gets too small, the block allocation algorithm launches the garbage collection algorithm. The garbage collection algorithm then finds and reclaims an erase unit that best matches the following criteria:

- the largest number of garbage blocks
- the least number of erase cycles
- the most static areas

In addition to these measurable criteria, the garbage collection algorithm also factors in a random selection process. This helps guarantee that the reclamation process covers the entire medium evenly and is not biased due to the way applications use data.

### 1.2.5 Block Allocation and the Clustering of Related Data

To promote more efficient data retrieval, TrueFFS uses an allocation strategy that clusters related data (such as the blocks that comprise the sectors of a file) into a contiguous area in a single erase unit. To prepare for the clustering of related data, TrueFFS tries to maintain a pool of physically consecutive free blocks that are resident in the same erase unit. If a pool of physically consecutive blocks is impossible, TrueFFS tries to assure that all the blocks in the pool reside in the same erase unit. If that too is impossible, TrueFFS tries to allocate a pool of blocks in the erase unit that has the most space available.

This approach of clustering related data has several benefits. If TrueFFS must access flash through a small memory window, the clustered related data cuts down on the number of calls required to map physical blocks into the window. This allows faster retrieval times for files accessed sequentially (the usual case).

Clustering related data also cuts down on fragmentation. This is because deleting a file tends to free up complete blocks that can be easily reclaimed. This means that garbage collection is faster. Clustering related data also has the effect of localizing...
blocks that belong to static files. This makes it much easier to transfer these blocks when the wear-leveling algorithm decides to move static areas.

1.3 Fault Recovery in TrueFFS

A fault can occur whenever data is written to flash. Thus, a fault can occur in response to a write request from the file system, during garbage collection, or even when TrueFFS formats or erases the flash. In all cases, TrueFFS can recover from the fault. In the case of new data being written to flash for the first time, the data being written at the time of the fault is lost. However, TrueFFS takes great care to assure that all the data already resident in flash is recoverable. Thus, the file and directory structures of the disk are retained.

1.3.1 “Erase after Write” Guarantees Data Integrity

The key to the robustness of TrueFFS is the fact that it uses an “erase after write” algorithm. When updating a sector on the flash medium, the previous data is not erased until after the update operation has completed and the newly stored data has been verified. One consequence of this “erase after write” algorithm is that a data sector cannot be in a partially written state. Either the operation completed, in which case the new sector is valid, or the operation did not complete, in which case the old sector is still valid.

This has obvious advantages for the stability of user data already written to the flash medium. If the write phase of an update operation fails, the old data is not lost or in any way corrupted. This “erase after write” algorithm also has positive consequences for the coherency of the flash memory map. For, although TrueFFS does use a RAM-resident mapping table to track the contents of flash memory, TrueFFS is careful never to store critical mapping information in RAM only. Thus, if an update to the mapping information is interrupted by a power loss or the removal of the flash medium, the old version of the mapping information is still valid.

When power is restored (or the medium reconnected), TrueFFS is able to use information resident in flash to reconstruct or verify the RAM-resident version of the flash mapping table. Given that the mapping information could reside anywhere on the medium, this might, at first, seem to be an impossible task.
Fortunately, each erase unit in flash memory maintains header information at a predictable location. By carefully cross-checking the header information in each erase unit, TrueFFS is able to rebuild or verify a RAM copy of the flash mapping table. Thus, after data is safely written to flash, that data is essentially immune to power failures. In fact, the only consequence of such interruptions is the need to restart any garbage collection that might have been under way at the time of the failure.  

1.3.2 Recovering from Failures During Write or Erase Operations

A write or erase operation can fail because of a hardware problem or a power failure. To prevent the possible loss of data, the success of each write operation is monitored and verified in TrueFFS. Most flash components use an on-chip register to report the success of an operation. TrueFFS uses this register (if available) to verify successful writing and erasing. In addition, TrueFFS verifies the operation by reading back the actual written data and comparing it to the user data.

If the first attempt at a write operation fails, this failure is usually not reported back to the user. Instead, TrueFFS uses its dynamic mapping ability to retry the write operation again to a different location on the flash medium. This ensures the integrity of the data by making failure recovery automatic. This write-error recovery mechanism becomes particularly valuable as the flash medium approaches its cycling limit (end-of-life). At that time, flash write/erase failures become more frequent, but the only user-observed effect is a gradual decline in performance (because of the need for write retries).

1.3.3 Recovering from Failures During Garbage Collection

TrueFFS reclaims garbage space (space occupied by sectors that have been deleted by the host) by moving data from one unit to another (a transfer unit), and erasing the original unit. If, because of a consistent flash failure, it is not possible to do the necessary write operations to move data, or it is not possible to erase the old unit, the garbage collection operation fails.

To minimize the possibly that the write part of a transfer fails, TrueFFS formats the flash medium to contain more than one transfer unit. Thus, if the write to one transfer unit fails, TrueFFS retries the write using a different transfer unit. If all transfer units fail, this does not have a direct effect on the user data (all of which is

5. The header information, normally at offset 0 of an erase unit, can also reside at an alternate offset if offset 0 is unusable. As a result, the location of the header information is dynamic.
already safely stored). However, the medium no longer accepts new data and becomes a read-only device.

### 1.3.4 Recovering from Failures During Formatting

In some cases, sections of the flash medium could be found unusable (typically because they are unerasable) when flash is first formatted. Provided the number of bad units does not exceed the number of transfer units, formatting can succeed and the medium is usable. The only adverse effect observed by the user is that the formatted capacity of the flash medium is reduced.

### 1.4 Sharing Flash Memory Between TrueFFS and a Boot Image

By default, the TrueFFS formatting utilities format the entire flash medium for use under TrueFFS. Although the translation services of TrueFFS provide many advantages for managing the data associated with a file system, those same services also complicate the use of flash memory as a boot device. The only practical solution to this problem is to exclude TrueFFS from the boot area of flash.

Fortunately, the formatting utility shipped with TrueFFS for Tornado lets you start the formatting operation at an offset. This creates a fallow area within flash that is totally outside the reach of TrueFFS. TrueFFS for Tornado also includes utilities that you can use to manage this fallow area. This functionality includes the ability to write a boot image into the fallow area as well as erase it.

However, when using these utilities, you must exercise caution. These utilities bypass all the protections built into TrueFFS. Thus, not only is it possible to overprogram the flash medium, it is also possible to write outside the fallow area. Handled carelessly, such a write could irretrievably corrupt the TrueFFS-managed data.
Running the VxWorks Image out of Flash Memory

The discussion above assumes that you want to use the flash device to store a VxWorks image that is retrieved from the flash device and then run out of RAM. However, because it is often possible to map a flash device directly into the target’s memory, it is also possible run the VxWorks image out of flash memory. Still, there are some restrictions:

- The flash device must be non-NAND.
- Only the text segment of the VxWorks image (\texttt{vxWorks.res\_rom}) may run out of flash memory. The data segment of the image must reside in standard RAM.
- No part of the flash device may be erased while the VxWorks image is running from flash memory.

This last restriction means that you cannot run a VxWorks boot image out of a flash device that must also support a writable file system (although a read-only file system is OK). A writable flash-resident file system requires garbage collection, which, in turn, requires an erase cycle.

Unfortunately, the current physical construction of flash memory devices does not allow access to the device while an erase is in progress anywhere on the flash device. As a result, if TrueFFS tries to erase a portion of the flash device, the entire device becomes inaccessible to all other users. If that other user happens to be the VxWorks image looking for its next instruction, the VxWorks image crashes.

Thus, if you want to implement both a writable file system and a \texttt{vxWorks.res\_rom} in flash, you must implement them on separate sockets. Because this conflict arises only during a write to the flash device, the \texttt{vxWorks.res\_rom} image can share the flash device with a file system if the file system is read only.
1.5 TrueFFS is a Layered Product

As shown in Figure 1-1, every configuration of TrueFFS consists of three layers, the translation layer, the MTD layer, and the socket layer.

The translation layer handles the high-level interaction between TrueFFS and dosFs. This layer also contains all the intelligence needed to handle mapping the flash into blocks, wear-leveling, garbage collection, and data integrity. Currently, there are three different translation layer modules to choose from. Which you use depends on whether the flash medium uses a NOR-based, a NAND-based, or an SFFDC-based technology.

The socket layer provides the interface between TrueFFS and the board hardware. The name derives from the physical socket into which users would plug flash cards. Now that flash memory has been used to implement “disk on chip” devices, the name is somewhat outdated, but it persists. Currently, WRS supplies socket layer code for a variety of BSPs. If you want to use TrueFFS with a BSP that does not currently support TrueFFS, you must port the socket layer. For more information, see 3. Writing Socket Component Drivers and MTDs.

The third layer of TrueFFS consists of standard Memory Technology Drivers (MTDs). TrueFFS already includes MTDs for the flash devices from Intel, AMD, and Samsung. As new devices requiring new MTDs are developed, you can use a standard interface to add these MTDs as plug-in drivers.

When configuring TrueFFS for use under VxWorks, you must include at least one software module for each layer. For more information on configuring TrueFFS for use with VxWorks, see 2. Getting Started.
1.6 TrueFFS Configuration Display Utilities

Included with TrueFFS for Tornado are the utilities *tffsShow()* and *tffsShowAll()*.
The *tffsShow()* routine prints device information for a specified socket interface.
It is particularly useful when trying to determine the number of erase units
required to write a boot image. The *tffsShowAll()* routine provides the same
information for all socket interfaces registered with VxWorks.

1.7 About I/O Sequencing

VxWorks favors task priorities over I/O sequence when servicing I/O requests.
Thus, if a low priority task and a high priority task request an I/O service while
the resource is busy, the high priority task gets the resource when it becomes
available—even if the low priority got its request in before the high priority task.
To VxWorks, a flash device is just another resource.
2

Getting Started

2.1 Introduction

This chapter tells you how to configure TrueFFS for Tornado and include it in VxWorks. For information on creating new socket component drivers and MTDs, see 3. Writing Socket Component Drivers and MTDs.

To include TrueFFS for Tornado in a VxWorks image, you must use the project configuration facility to include “TrueFFS Flash File System” support (configuration macro: INCLUDE_TFFS). This configures the VxWorks initialization code to call tffsDrv(), the utility that sets up all the structures and global variables needed to manage TrueFFS. This call also registers socket component drivers for all the flash devices attached to your target. At link time, resolving the symbols associated with tffsDrv() should pull all necessary TrueFFS software modules into the VxWorks image.

To support TrueFFS, each BSP must include a sysTffs.c file. It is this file that pulls together all the layers of TrueFFS (translation, socket, and MTD) and binds them with VxWorks. Because of this central role, sysTffs.c is the file you must edit to configure which MTDs and translation layer modules are included in TrueFFS. In addition, if your target includes a memory management unit (an MMU), you will need to edit the sysPhysMemDesc[ ] array defined in sysLib.c.

After rebuilding the VxWorks image and rebooting the target, you should have access to all the functionality necessary to format flash, create a TrueFFS block device, and bind that block device to dosFs.
2.2 An Overview of Configuring and Using TrueFFS

Configuring VxWorks to include TrueFFS for Tornado is a matter of rebuilding the VxWorks image after editing the following:

- **The project**—to add `sysTffs.c` to the list of project files
- `sysLib.c`—to adjust the description of the ROM area
- `sysTffs.c`—to determine the features included in TrueFFS

When you boot this image, it automatically runs `tffsDrv()`. This function call automatically registers a socket component for each flash device. At this point, there is not yet a block device driver for the flash, but the socket component driver does provide enough of a connection to support a call to `tffsDevFormat()`. Use this routine to format the flash medium for use with TrueFFS. To create a TrueFFS block device on top of the socket component and then mount dosFs on that block device, call `usrTffsConfig()`.

2.2.1 Configuring VxWorks and TrueFFS

As mentioned above, configuring VxWorks to include TrueFFS requires edits to:

- **The project**—to add `sysTffs.c` to the list of project files
- `sysLib.c`—to adjust the description of the ROM area
- `sysTffs.c`—to determine the features included in TrueFFS

For a BSP that supports TrueFFS, all these files must reside within your BSP configuration directory under `target/config`. However, `sysTffs.c` is not automatically shipped in this directory. Instead, several versions of `sysTffs.c` are shipped in `src/drv/tffs/sockets` as `ads860-sysTffs.c`, `mv177-sysTffs.c`, `koba47-sysTffs.c`, and so on.

Read `src/drv/tffs/sockets/README` to determine which `bspname-sysTffs.c` is appropriate to your BSP. This README file also contains brief descriptions of all the BSP-specific changes needed to support TrueFFS. After choosing a `sysTffs.c` file, copy it to your BSP’s `target/proj/projectname` directory.
Changing the Project

To add the `sysTffs.c` file that you have moved to your project’s directory to the list of objects it built for your BSP, use the Tornado Project Facility.

Changing the configuration

To configure VxWorks to use TrueFFS, use the Tornado project facility to add “TrueFFS Flash File System” support (configuration constant: `INCLUDE_TFFS`). Because TrueFFS provides a block driver for use with dosFs, you also want to configure VxWorks with this setting defined (configuration constant: `INCLUDE_DOSFS`).

Changing `sysLib.c`

If your target includes an MMU, its BSP defines a `sysPhysMemDesc[ ]` table in its `sysLib.c`. Typically, this table tells the MMU that the memory area that contains the boot image is `WRITABLE_NOT`, or, in other words, ROM. Until recently, ROM was the only technology that could reliably store a memory-resident boot image. As a result, VxWorks has always defaulted to the assumption that the memory area containing a boot image is ROM.

However, with the advent of flash technology, the possibilities have been expanded. Flash memory is both writable and capable of reliably storing a boot image. Thus, if you intend to store a boot image in flash memory, you must edit `sysPhysMemDesc[ ]` and reassign the boot image memory area as `WRITABLE`.

Changing `sysTffs.c`

The primary function of this file is to define the BSP-specific socket code that bridges the gap between the flash hardware and VxWorks. You also use `sysTffs.c` to determine which software modules you want to include in TrueFFS. By default, a WRS-supplied `sysTffs.c` includes all translation layer modules, all MTD modules, as well as code for `tffsBootImagePut()`, `tffsShow()`, and `tffsShowAll()`. To reduce the image size, you can edit `sysTffs.c` to exclude modules that you know to be unnecessary to your particular application.
Choosing Translation Layer Modules

Use the symbolic constants shown in Table 2-1 to determine which translation layer modules are included in `sysTffs.o`.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Associated Flash Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCLUDE_TL_NFTL</td>
<td>NAND-based flash</td>
</tr>
<tr>
<td>INCLUDE_TL_FTL</td>
<td>NOR-based flash</td>
</tr>
<tr>
<td>INCLUDE_TL_SSFDC</td>
<td>SSFDC flash</td>
</tr>
</tbody>
</table>

If you need to save on memory space and know that your embedded system does not need to support the hardware associated with a specific module, redefine the associated constant. This excludes that module from `sysTffs.o`. For example, if the only flash device included on your target were a flash array implemented on AMD 29F040 flash, a NOR-based technology, you would define `INCLUDE_TL_FTL` and undefine `INCLUDE_TL_SSFDC` and `INCLUDE_TL_NFTL`.

Choosing MTDs

Use the symbolic constants shown in Table 2-2 to determine which MTDs are included in `sysTffs.o`. By default, `sysTffs.c` defines all the constants shown in Table 2-2 and thus includes all MTDs.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Associated Flash Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCLUDE_MTD_I28F016</td>
<td>Intel 28f016</td>
</tr>
<tr>
<td>INCLUDE_MTD_I28F008</td>
<td>Intel 28f008</td>
</tr>
<tr>
<td>INCLUDE_MTD_I28F008_BAJA</td>
<td>Intel 28f008 on the Heurikon Baja 4000</td>
</tr>
<tr>
<td>INCLUDE_MTD_AMD</td>
<td>AMD, Fujitsu: 29F0[40,80,16] 8-bit devices</td>
</tr>
<tr>
<td>INCLUDE_MTD_CDSN</td>
<td>Toshiba, Samsung: NAND CDSN devices</td>
</tr>
<tr>
<td>INCLUDE_MTD_DOC2</td>
<td>Toshiba, Samsung: NAND DiskOnChip 2000 devices</td>
</tr>
<tr>
<td>INCLUDE_MTD_CFISCS</td>
<td>CFI/SCS device</td>
</tr>
<tr>
<td>INCLUDE_MTD_WAMD</td>
<td>AMD, Fujitsu 29F0[40,80,16] 16-bit devices</td>
</tr>
</tbody>
</table>
If you need to save on memory space and know that your embedded system does not need to support the hardware associated with a specific MTD, undefine the associated constant. This excludes that MTD from `sysTffs.o`. For example, if the only flash device included on your target was an 8-bit flash array implemented on AMD 29F040 flash, you could safely undefine all the MTD defines other than `INCLUDE_MTD_AMD`. This guarantees that `sysTffs.o` includes only that MTD.

**NOTE:** If you write your own MTDs, put them in `src/drv/tffs`. You also need to modify the `src/drv/tffs/Makefile` so that `make` knows how to generate objects for the sources you add. Likewise, you need to modify your BSP’s `sysTffs.c` as well as the `mtdTable[]` in `src/drv/tffs/tffsConfig.c`.

**Including Support for Flash-Resident Boot Images**

By default, `sysTffs.c` defines `INCLUDE_TFFS_BOOT_IMAGE`. Defining this constant automatically includes `tffsBootImagePut()` in your `sysTffs.o`.

Using `tffsBootImagePut()`, you can bypass TrueFFS (and its translation layer) and write directly into any location in flash memory. Although the translation services of TrueFFS provide many advantages for managing the data associated with a file system, those same services also complicate the use of flash memory as a boot device. The only practical solution to this problem is to exclude TrueFFS from the boot area of flash.

Using `tffsDevFormat()`, it is possible to format flash memory so that the TrueFFS-managed area starts at an offset. This creates a fallow area within flash that is totally outside the reach of TrueFFS. To write a boot image into this fallow area, use `tffsBootImagePut()`.

**WARNING:** Because `tffsBootImagePut()` lets you write directly to any area of flash, it is possible to accidentally overwrite and corrupt the TrueFFS-managed area of flash. For more on this and other dangers associated with this powerful utility, see the reference entry for `tffsBootImagePut()`.

**Including Support for Flash Show Functionality**

Defining `INCLUDE_SHOW_ROUTINES` automatically includes `tffsShow()` and `tffsShowAll()` in `sysTffs.o`. The `tffsShow()` routine prints device information for a specified socket interface. It is particularly useful when trying to determine the number of erase units required to write a boot image. The `tffsShowAll()` routine provides the same information for all socket interfaces registered with VxWorks. If you want to exclude these routines from TrueFFS, undefine them in your BSP’s
**Configuring Socket Layer Features**

Although define statements in `sysTffs.c` control the inclusion of translation layer modules and MTDs and other flash-related utilities, the bulk of the file is dedicated to defining the functions that comprise the socket component driver. These functions are a standardized API that bridges the gap between the device hardware and VxWorks.¹

The socket layer code in your BSP’s `sysTffs.c` must provide reasonable definitions for all the functions in the standardized socket layer API. What constitutes a reasonable function definition depends largely on the flash hardware you have chosen. TrueFFS for Tornado supports three general categories of flash hardware:

- PC flash cards (removable flash media)
- DiskOnChip 2000 devices
- board-resident flash arrays

Both the DiskOnChip 2000 and the board-resident flash arrays are non-removable flash media. Their socket interface needs are relatively simple. As a result, simple “do little or nothing” routines are reasonable implementations for many of the standard socket API functions. All BSPs that support TrueFFS for Tornado contain socket code that can handle the simple socket interface needs of these board-resident flash media.

However, the PC flash cards are a removable flash medium. As a result, their socket interface needs are much more demanding. The interface provided with most BSPs is robust but rather simple. It assumes the socket contains a flash card, and it does not support hot swapping. Currently, there is a big-endian and a little-endian version of this simple PCIC driver. The big-endian version is implemented in the `sysTffs.c` that ships with the ads860 BSP. The little-endian version is available in the pc386 and the pc486 BSPs.

Associated with the PCIC driver are two defines, `INCLUDE_SOCKET_PCIC0` and `INCLUDE_SOCKET_PCIC1`. Edit your BSP’s `sysTffs.c` and define or undefine these constants according to your needs:

- **INCLUDE_SOCKET_PCIC0**
  - Creates a socket interface driver for slot 0.

---

¹ For more information the standardized socket layer API, see 3. Writing Socket Component Drivers and MTDs.
INCLUDE_SOCKET_PCIC

Creates a socket interface driver for slot 1.

For the pc386 and pc486 BSPs, you have the option of including a more sophisticated socket interface driver than the simple PCIC driver. For more information on this interface, see 2.4 Socket Layer Options for the pc386 and pc486 BSPs, p.28.

2.2.2 Formatting Flash

The *tffsDevFormat()* function formats flash memory for use with TrueFFS. During the formatting process, this function erases the flash medium and then writes the TrueFFS data-management structures into a header at the start of each erase unit.

As input, *tffsDevFormat()* expects a drive number (socket component number) and a pointer to a *FormatParams* structure. The drive number identifies the flash medium to be formatted and is determined by the order in which the socket components were registered. The *FormatParams* structure passes in values that specify how you want the flash to be formatted.

To facilitate calling *tffsDevFormat()* from a target shell, the second parameter accepts a 0 (the value zero) instead of a *FormatParams* pointer. This value tells *tffsDevFormat()* to use the default *FormatParams* structure defined and initialized in *dosformt.h*. The values defined in this default structure are good enough for most purposes.

However, if you need to share the flash medium between TrueFFS and a boot image, the default values in *dosformt.h* are inadequate. This is because you need to format the flash so that the TrueFFS segment starts at an offset. To do this, you must submit a *FormatParams* structure whose *bootImageLen* member value is at least as large as the boot image.

When *tffsDevFormat()* formats flash, it notes the offset, then erases and formats all erase units with starting addresses higher than the offset. The erase unit containing the offset address (and all previous erase units) are left completely untouched. This preserves any data stored before the offset address.

*Setting the Cluster Size*

When formatting a flash device for use with TrueFFS, each sector on the medium is assigned to a cluster. The number of sectors assigned to each cluster is determined by *flMinClusterSize*, an int global variable defined in *dosFormat.c*. Valid values for this global variable are non-negative integer powers of two (1, 2, 4, 8,...). The default value is 4.
For a finer granularity in describing the flash storage space, you can change the value of `flMinClusterSize` to something smaller than 4. However, because the number of FAT entries is limited, the addressable space on the flash medium is reduced if `flMinClusterSize` is reduced. For example, 16 megabytes is the largest drive addressable if `flMinClusterSize` is set to one.

Thus, choosing an `flMinClusterSize` value means making a compromise between a small address space and small inter-cluster dead spaces. For most file systems, a default `flMinClusterSize` of 4 is a reasonable compromise.

**NOTE:** If you change `flMinClusterSize`, the value remains constant until you explicitly reset it or reboot. The value of `flMinClusterSize` does not revert to the default after rebooting, but not at the end of a format session. If a target system includes multiple flash drives that require different `flMinClusterSize` values, make sure you set `flMinClusterSize` correctly prior to formatting each drive.

---

**About `sysTffsFormat()`**

In some of the configuration examples that finish up this chapter, you are asked to call `sysTffsFormat()`. Internally, `sysTffsFormat()` calls `tffsDevFormat()`. As input to `tffsDevFormat()`, the `sysTffsFormat()` function supplies a pointer to a `FormatParams` structure whose `bootImageLen` member specifies the offset after which to format the flash medium for use with TrueFFS. The area below this offset is excluded from TrueFFS. Typically, you would use such an area to store a boot image. For more information, see the BSP-specific reference entry for the `sysTffsFormat()` defined in your BSP’s `sysTffs.c` file.

**Problems Associated with Expanding the Region Assigned to Boot Images**

As mentioned above, when `tffsDevFormat()` formats flash above an offset, it does not touch the flash below that offset. Normally, this is a good thing. However, problems can arise if any of the erase units below the offset had previously been formatted for use by TrueFFS. These erase units would contain TrueFFS formatting headers written by an earlier call to `tffsDevFormat()`.

When TrueFFS mounts a flash device, it scans the entire flash medium for TrueFFS formatting headers. It uses these headers to construct a map of the regions of flash memory that fall under its control. If the flash below the current offset happens to contain TrueFFS headers left over from a previous `tffsDevFormat()` call, the TrueFFS mount can still see these headers and will fail as a result.

To overcome this problem, you can use `tffsRawio()` to do a physical erase of the problem erase units. However, this is a very powerful and dangerous utility. If misused, it can permanently damage flash memory. During the development
phase of your embedded system, you can probably trust yourself to use it carefully. However, if you need to make the functionality of this utility available in the deployed version of your product, you should surround that functionality with protections appropriate to your application and its users.

### 2.2.3 Creating TrueFFS Block Devices

Before you can create a logical TrueFFS block device, you need to have already run `tffsDrv()`. If you configured VxWorks correctly, this is handled for you automatically at boot time. Running `tffsDrv()` initializes TrueFFS for Tornado, which includes setting up the mutual exclusion semaphore, global variables, and
data structures needed to manage TrueFFS. This initialization also includes the registration of socket component drivers for all the flash devices on the target.

Registering a socket component driver with TrueFFS starts with getting an \texttt{FLSocket} structure from a preallocated 5-element TrueFFS-internal array of \texttt{FLSocket} structures. The next step is to update that \texttt{FLSocket} structure to contain data and pointers to functions that handle the basic hardware interface to the flash device.

When TrueFFS needs to interact with the socket hardware for a particular flash device, it uses the drive number (0 through 4) as an index into its array of \texttt{FLSocket} structures. TrueFFS then uses the functions referenced in that structure to handle its hardware interface needs. Although these socket interface functions do not quite provide a block device interface, they do provide an interface that is good enough for utilities such as \texttt{tffsDevFormat()}, a function you can use to format the flash medium for use with TrueFFS. This ability at this stage is important as it is not possible to create a TrueFFS block device for a flash medium that has not been previously formatted for use with TrueFFS.

After registering a socket component driver for a flash device, it is possible to create a TrueFFS block device on top of the socket component driver. To do this, call \texttt{tffsDevCreate()}. As input, you must specify a number (0 through 4, inclusive) that identifies the socket component driver on top of which to construct the TrueFFS block device. The \texttt{tffsDevCreate()} call uses this number as an index into the array of \texttt{FLSocket} structures. This number is visible later to dosFs as the driver number.

After creating the TrueFFS block device, you must mount dosFs onto the device. To do this, you must call \texttt{dosFsDevInit()}. After mounting dosFs, you can read and write from flash memory just as you would from a standard disk drive. In the examples at the end of this chapter, there are no calls to \texttt{tffsDevCreate()} or \texttt{dosFsDevInit()}. Instead, there is a call to \texttt{usrTffsConfig()}. Internally, this function handles the call to \texttt{tffsDevCreate()}, \texttt{dosFsDevInit()}, and the other functions necessary to create a TrueFFS block device and mount dosFs on that device.\textsuperscript{2}

\textsuperscript{2} To see the source code for \texttt{usrTffsConfig()}, look in \texttt{target/src/config/usrTffs.c}.
2.3  BSP-Specific Examples of Using TrueFFS

The following subsections give examples of how to set up TrueFFS for Tornado on each of the non-x86 BSPs that now ship with a WRS-supplied sysTffs.c.³

The examples given are meant to be illustrative, but they do not show all the configurations possible for each BSP. For example, only the ads860, the iq960rp, and the pid7t examples use a TrueFFS format offset (to leave room on the medium for a boot image). However, all BSPs support the use of this formatting feature.

NOTE: The instructions below do not modify the define statements in sysTffs.c that are used to determine which MTDs are included in TrueFFS. As a result, all MTDs are included. In your deployed system, you might want to trim the list of included MTDs. For more information on how to do this, see Choosing MTDs, p.16.

2.3.1  ads860 with a Board-Resident Flash Array and a PCMCIA Slot

The following instructions show you how to setup two TrueFFS devices for a target using the ads860 BSP. The first flash device, the board-resident flash array, is formatted using sysTffsFormat(). This leaves room for a boot image. The second device, the flash card in the PCMCIA slot, is dedicated entirely to TrueFFS and so is formatted using tffsDevFormat().

NOTE: Instead of editing config.h in the following steps, it may be to your advantage to set these constants by using the Tornado project facility. See the Tornado User’s Guide: Projects.

1. To config.h, add the lines:

```c
#define INCLUDE_TFFS
#define INCLUDE_SHOW_ROUTINES /* optional */
#define INCLUDE_DOSFS
```

2. In config.h, change the value of ROM_SIZE from 1MB to 2MB:

```c
#define ROM_SIZE 0x00200000 /* 2M ROM space */
```

3. Define the following addresses in ads860.h:

3. The pc386 and pc486 BSPs support a socket interface driver not available to other BSPs. For more information, see 2.4 Socket Layer Options for the pc386 and pc486 BSPs, p.28.
4. In `sysLib.c`, edit `sysPhysMemDesc[]` to add the following two entries:

```
{ (void *) PC_BASE_ADRS_0,  
  (void *) PC_BASE_ADRS_0,  
  PC_SIZE_0,  /* 1 m - PCMCIA window 0 */  
  VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,  
  VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}

and

```

```
{ (void *) PC_BASE_ADRS_1,  
  (void *) PC_BASE_ADRS_1,  
  PC_SIZE_1,  /* 32 m - PCMCIA window 1 */  
  VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,  
  VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

5. Change the Makefile as specified in Changing the Project, p.15.

6. Remake VxWorks.

7. Reboot the target.

8. Insert a flash card in the PCMCIA socket.

9. At the target shell prompt, enter the following commands:

```
-> sysTffsFormat
-> tffsDevFormat 1,0
-> usrTffsConfig 0,0,„/RFA/”
-> usrTffsConfig 1,1,„/PCMCIA1/”
```

RFA stands for “resident flash array.”
2.3.2 hkbaja47 a Board-Resident Flash Array

The following instructions show you how to setup and instantiate a single TrueFFS device for a target using the hkbaja47 BSP. This flash device, a board-resident flash array, is dedicated entirely to TrueFFS and so is formatted using `tffsDevFormat()`.

1. To `config.h`, add the lines:

```c
#define INCLUDE_TFFS
#define INCLUDE_SHOW_ROUTINES       /* optional */
#define INCLUDE_DOSFS
```

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set these constants by using the Tornado project facility. See the *Tornado User’s Guide: Projects*.

2. Change the `Makefile` as specified in *Changing the Project*, p.15.
3. Remake VxWorks.
4. Reboot the target.
5. At the target shell prompt, enter the following commands:

   ```
   -> tffsDevFormat 0, 0
   -> usrTffsConfig 0,0,"/RFA/
   ```

2.3.3 iq960rp with a Board-Resident Flash Array

The following instructions show you how to setup and instantiate a single TrueFFS device for a target using the iq960rp BSP. This flash device, a board-resident flash array, is formatted using `sysTffsFormat()` to leave room for a boot image.

**NOTE:** The iq960rp BSP was designed to support the iq960rp board. However, if you apply a patch available from WRS, this BSP can support the iq960rd board as well as TrueFFS for Tornado.

1. To `config.h`, add the lines:

```c
#define INCLUDE_TFFS
#define INCLUDE_SHOW_ROUTINES       /* optional */
#define INCLUDE_DOSFS
```
NOTE: Instead of editing config.h, it may be to your advantage to set these constants by using the Tornado project facility. See the Tornado User's Guide: Projects.

2. Change the Makefile as specified in Changing the Project, p.15.
3. Remake VxWorks.
4. Reboot the target.
5. At the target shell prompt, enter the following commands:

   -> sysTffsFormat
   -> usrTffsConfig 0,0,"/RFA/"

2.3.4 mv177 with a Board-Resident Flash Array

The following instructions show you how to setup and instantiate a single TrueFFS device for a target using the mv177 BSP. The board-resident flash array is dedicated entirely to TrueFFS and so is formatted using tffsDevFormat().

1. To config.h, add the lines:

   #define INCLUDE_TFFS
   #define INCLUDE_SHOW_ROUTINES /* optional */
   #define INCLUDE_DOSFS

   NOTE: Instead of editing config.h, it may be to your advantage to set these constants by using the Tornado project facility. See the Tornado User's Guide: Projects.

2. In sysLib.c, change the VM_STATE of the ROM region from WRITABLE_NOT to WRITEABLE:

   /* ROM */
   {
   (void *) ROM_BASE_ADRS,  
   (void *) ROM_BASE_ADRS,  
   0x400000,  
   VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,  
   VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT  
   },

3. Change the Makefile as specified in Changing the Project, p.15.
4. Remake VxWorks.
5. Reboot the target.
6. Insert a flash card in each PCMCIA socket.
7. At the target shell prompt, enter the following commands:

   -> tffsDevFormat 0,0
   -> usrTffsConfig 0,0,"/RFA/"

2.3.5 pid7t or pid7t_t with a Board-Resident Flash Array

The following instructions show you how to setup and instantiate a TrueFFS
device for a target using either the pid7t or pid7t_t BSP. The flash device, a board-
resident flash array, is formatted using \texttt{sysTffsFormat()}. This leaves room for a
boot image.

1. To config.h, add the lines:

   \begin{verbatim}
   #define INCLUDE_TFFS
   #define INCLUDE_SHOW_ROUTINES /* optional */
   #define INCLUDE_DOSFS
   \end{verbatim}

   \textbf{NOTE:} Instead of editing config.h, it may be to your advantage to set these
   constants by using the Tornado project facility. See the \textit{Tornado User's Guide:}
   Projects.

2. Change the \texttt{Makefile} as specified in Changing the Project, p.15.
3. Remake VxWorks.
4. Reboot the target.
5. At the target shell prompt, enter the following commands:

   -> sysTffsFormat
   -> usrTffsConfig 0,0,"/RFA/"

2.3.6 ss5 with a Board-Resident Flash Array

The following instructions show you how to setup and instantiate a TrueFFS
device for a target using the ss5 BSP. This flash device, a board-resident flash array,
is dedicated entirely to TrueFFS and so is formatted using \texttt{tffsDevFormat()}. In
addition to the usual changes to config.h, the instructions below require that
you undefine \texttt{INCLUDE_D_CACHE_ENABLE}. TrueFFS is incompatible with data
 caching. Additionally, you must define \texttt{INCLUDE_FLASH}. This configures \texttt{sysLib.c}
to include special functions necessary to TrueFFS on the ss5 BSP.
1. To *config.h*, add the lines:

```c
#define INCLUDE_TFFS
#define INCLUDE_SHOW_ROUTINES  /* optional */
#define INCLUDE_DOSFS
#undef  INCLUDE_D_CACHE_ENABLE  /* disable data cache */
#define INCLUDE_FLASH  /* make flash writable */
```

**NOTE:** Instead of editing *config.h*, it may be to your advantage to set these constants by using the Tornado project facility. See the *Tornado User’s Guide: Projects*.

2. Change the *Makefile* as specified in *Changing the Project*, p.15.
3. Remake VxWorks.
4. Reboot the target.
5. At the target shell prompt, enter the following commands:

```bash
-> tffsDevFormat 0, 0
-> usrTffsConfig 0,0,"/RFA/"
```

### 2.4 Socket Layer Options for the pc386 and pc486 BSPs

For the pc386 and pc486 BSPs, you have the option of including a fully functional PCMCIA interface driver that TrueFFS can use to handle its socket layer interface. You can use this PCMCIA driver instead of the PCIC driver. The PCMCIA interface driver is designed to support a wide variety of PC card devices, including flash memory card devices. It supports hot swapping and is careful to check the type of the device in the PCMCIA socket before it sends any commands to the device.4

Of course, there are situations in which you need some support for PC flash cards but could do without all the services provided by the PCMCIA driver. In such cases, you should use the simple PCIC driver with *pc386sysTffs.c*. The PCIC driver does not support hot swapping, nor does it check the device type before sending commands to the socket. However, the PCIC code is quite small.

---

4. Support for “hot swapping” means registering a function to handle the interrupt generated when the user removes a device from the PCMCIA socket.
NOTE: It is not necessary to define INCLUDE_PCMCIA in order for TrueFFS to use the PCMCIA sockets. The PCIC driver can provide TrueFFS with an interface to that hardware. Likewise, the PCMCIA package does not require TrueFFS (unless you intend to support TrueFFS-managed flash cards in those sockets).

2.4.1 Configuring the pc386 and pc486 for TrueFFS and the PCMCIA Driver

The changes described in Changing the Project, p.15, and Changing sysTffs.c, p.15, apply to both the pc386 and pc486 BSPs. However, the changes you must make to config.h and syslib.c differ as described below.

Changing config.h

If you want to use the simple PCIC driver, follow the instructions in Changing the configuration, p.15. To use the PCMCIA driver you will need to modify config.h as described below.

- Including TrueFFS Support with the PCMCIA Driver

The PCMCIA interface driver is a product offering distinct from TrueFFS for Tornado. To include it in VxWorks, you must define INCLUDE_PCMCIA in your config.h file. Because this PCMCIA interface driver must support TrueFFS for Tornado, you must also add INCLUDE_TFFS to the list of devices that PCMCIA includes. Thus, in addition to defining INCLUDE_PCMCIA, your config.h make sure that your config.h also defines the following:

```c
#ifndef INCLUDE_PCMCIA
#define INCLUDE_TFFS    /* include TFFS driver */
#define INCLUDE_ATA      /* include ATA driver */
#define INCLUDE_SRAM     /* include SRAM driver */
#ifdef INCLUDE_NETWORK
#define INCLUDE_ELT    /* include 3COM EtherLink III driver */
#endif
#endif
#endif
```

NOTE: Instead of editing config.h, it may be to your advantage to set these constants by using the Tornado project facility. See the Tornado User’s Guide: Projects.
Avoiding Address Conflicts with the DiskOnChip 2000

If you are using the M-System DiskOnChip 2000 product, you might run into address conflicts with the PCMCIA product. To resolve conflict, redefine the CIS macros as follows:

```c
#ifdef INCLUDE_TFFS
#undef CIS_MEM_START
#undef CIS_MEM_STOP
#undef CIS_REG_START
#undef CIS_REG_STOP
#define CIS_MEM_START   0xc8000 /* mapping addr for CIS tuple */
#define CIS_MEM_STOP    0xcbfff
#define CIS_REG_START   0xcc000 /* mapping addr for config reg */
#define CIS_REG_STOP    0xccfff
#endif /* INCLUDE_TFFS */
```

NOTE: Instead of editing config.h, it may be to your advantage to set these constants by using the Tornado project facility. See the Tornado User’s Guide: Projects.

Changing syslib.c

If your C: drive is a DiskOnChip 2000 device, and, if you boot from it instead of floppy disk, set the sysWarmType global variable to 3. This specifies a warm start from the DiskOnChip 2000 device. To enable rebooting from the TrueFFS flash device, you must change sysLib.c as follows:

- Define sysWarmTffsDrive

After the declaration of the sysWarmAtaDrive variable, add the following:

```c
int sysWarmTffsDrive = SYS_WARM_TFFS_DRIVE; /* TFFS drive 0 (DOC) */
```

- Changes to sysToMonitor()

In the function sysToMonitor(), find the clause that defines the reboot process for the reboot device ATA. After the #endif statement for INCLUDE_ATA, add the following:

```c
#ifdef INCLUDE_TFFS
if (sysWarmType == 3)
{
  IMPORT int dosFsDrvNum;
  tffsDrv (); /* initialize TFFS */
  if (dosFsDrvNum == ERROR)/* initialize DOS-FS */
    dosFsInit (NUM_DOSFS_FILES);

```
if (usrTffsConfig (sysWarmTffsDrive, FALSE, "/vxboot/" == ERROR) {
    printErr ("usrTffsConfig failed.\n");
    return (ERROR);
}
#endif /* INCLUDE_TFFS */

Search for the lines:

#if (defined (INCLUDE_FD) || defined (INCLUDE_ATA))
    if ((sysWarmType == 1) || (sysWarmType == 2))
#endif /* defined (INCLUDE_FD) || defined (INCLUDE_ATA) */

Replace them with the following:

#if (defined(INCLUDE_FD) || defined(INCLUDE_ATA) || defined(INCLUDE_TFFS))
    if ((sysWarmType == 1) || (sysWarmType == 2) || (sysWarmType == 3))
#endif /* (INCLUDE_FD) || (INCLUDE_ATA) || (INCLUDE_TFFS) */

2.4.2 x86 with DiskOnChip 2000 and Two PCMCIA Slots Using INCLUDE_PCMCIA

The following instructions show you how to setup and instantiate three TrueFFS devices (a DiskOnChip 2000 and two flash cards in the PCMCIA sockets) for a target using the pc386 or pc486 BSP. All three devices are dedicated entirely to TrueFFS and so each is formatted using \texttt{tffsDevFormat}(). Notice also that the instructions do not include setting INCLUDE_TFFS or INCLUDE_DOS. That is because defining INCLUDE_PCMCIA implicitly sets those defines for you. 5

1. Edit \texttt{config.h} as described in Changing config.h, p.29.
2. Edit \texttt{sysLib.c} as described in Changing syslib.c, p.30.
3. Change the \texttt{Makefile} as described in Changing the Project, p.15.
4. Remake VxWorks.
5. Reboot the target.
6. Insert a flash card in each PCMCIA socket.

5. The DiskOnChip 2000 is a “disk on chip” device supplied by M-Systems.
7. At the target shell prompt, enter the following commands:

```bash
-> tffsDevFormat 0,0
-> tffsDevFormat 1,0
-> tffsDevFormat 2,0
-> usrTffsConfig 0,0,"/DOC/"
-> usrTffsConfig 1,1,"/PCMCIA1/"
-> usrTffsConfig 2,1,"/PCMCIA2/"
```

Adding Boot Code to the DiskOnChip 2000

The DiskOnChip 2000 uses BIOS extensions to make itself appear to the BIOS as a hard disk. This section outlines the procedures you must follow to make sure that the DiskOnChip 2000 device correctly presents itself to the BIOS as a hard disk. VxWorks can then use the DiskOnChip 2000 as a boot device (just as if it were a floppy driver or a conventional hard disk).

Preparation

1. Build your VxWorks boot image (usually `bootrom_uncmp`).
2. Boot DOS on your target.
   
The `dformat` utility is an M-Systems tool that you should have received with your DiskOnChip 2000 flash device. Running the `dformat` utility clears the flash medium.
4. Using the `/s` switch, run the DOS format utility on the flash device. This guarantees that the flash medium has system tracks on it. The VxWorks boot strap installer assumes that you have done this and proceeds to modify the contents of the boot sector.
Installing the VxWorks Boot Code

You may install the VxWorks boot code using either DOS or the commands available from a VxWorks target shell.

Installing from DOS. To use this method, you must be able to connect a floppy drive to your target. You must also be able to boot DOS on the target.

1. Make a bootable floppy disk to which you must then copy the following files:
   - From host\x86-win32\bin:
     mkboot.bat
     vxcopy.exe
     vxsys.exe
   - From target\config\pc386 or target\config\pc486:
     bootrom_uncmp
   - From the your DOS distribution:
     chkdsk.exe

2. Put the floppy in the drive connected to your target.

3. Boot the target from the floppy.

4. Run mkboot from the floppy:

   mkboot flash_disk_drive_letter: bootrom_uncmp

   For more information on mkboot, see the mkboot reference entry. 6

   The 8.3 version of bootrom_uncmp usually lists as bootro~1. It causes no problems for mkboot if you specify bootrom_uncmp as bootro~1.

5. Remove floppy and reboot the target. The target should now boot VxWorks from the DiskOnChip 2000.

---

6. There are two versions of mkboot; one that runs on DOS, and one that runs on VxWorks. The purpose of mkboot is to install the VxWorks boot code on the boot disk. Both versions expect that the system tracks have been installed from DOS’s format utility using the /s switch. For more information, see the x86 release notes.
Installing from VxWorks. To use this method, you need an Ethernet connection to your target. After installing TrueFFS for Tornado and building VxWorks with TrueFFS included, reboot your target.

1. Build `mkboot.o` in the BSP directory.
2. Open a target shell and enter the following commands:

   ```
   -> ld < path_to_target_BSP_directory/mkboot.o
   -> mkbootTffs 0,0,"path_to_target_BSP_directory/bootro~1"
   ```

   The first zero in the `mkbootTffs` command shown above is the drive number. The second zero indicates that the flash device is not removable (that is, not a PC flash card).

X-86 Pitfalls—CMOS Setup

1. Shadowing must be turned off for the memory region in which DiskOnChip 2000 or any other flash device is located.
2. Caching must be turned off (disabled) in the flash disk region.
   
   In some BIOSs, you can specifically disable caching in the following regions: A0000 to F0000.

3. Some BIOSs handle caching in a way that is incompatible with TrueFFS. For these BIOSs, you must disable the external (L2) cache. You can leave the internal cache enabled.
4. Some PnP BIOSs do not load the DiskOnChip 2000 BIOS extension correctly. There is no way to test for this except to plug a DiskOnChip 2000 into the x86 board. The issue is being addressed with the BIOS vendors. There is a special debug version of TFFS BIOS for testing the system. You can get this version from M-Systems.
3

Writing Socket Component Drivers and MTDs

3.1 Introduction

The purpose of this chapter is to provide you with a description of the interface between your flash memory hardware, TrueFFS for Tornado, and VxWorks. Thus, this chapter describes all the standard functions and structures essential to a socket component driver and an MTD. The two structures that are your main concern are the FFlash structure and the FSocket structure.

Internally, TrueFFS for Tornado allocates an array containing five FFlash structures, one for each possible flash device. TrueFFS uses the FFlash structure to store the data and function pointers it needs to manage the flash device. For example, it is the FFlash structure that contains the MTD function pointers that TrueFFS uses to handle the low-level details of reading and writing the flash medium. These function pointers are installed by running an MTD identify routine.

The FFlash structure also contains a pointer to an FSocket structure. TrueFFS uses this FSocket structure to store the data and function pointers it uses to handle the hardware (socket) interface to the flash device. These functions are installed by calls from your sysTffsInit() to the xxxRegister() routines.

NOTE: TrueFFS for Tornado ships with a set of MTDs that supports a wide variety of flash devices. Thus, it is likely that you do not need to write your own MTD, although you are always free to do so. For a list of the flash devices currently supported, see 3.5 MTD-Supported Flash Devices, p. 57.
Registering Your Socket Component Driver with TrueFFS

Including TrueFFS in VxWorks configures `usrRoot()` to call `tffsDrv()`. This starts the chain of function calls shown in Figure 3-1. One of the goals of these function calls is to register the functions of your socket component driver with TrueFFS.

For the most part, the bulk of the work takes place in `xxxRegister()`, a function defined in `sysTffs.c`. The `xxxRegister()` function updates the `FLSocket` structure that TrueFFS has assigned to a drive number (volume) with references to the socket services available in the socket component driver. TrueFFS calls the functions referenced in the `FLSocket` structure to handle the hardware interface to the flash device.

**Figure 3-1  Registering a Socket Component Driver**

![Diagram](注册高速缓存驱动程序)

Identifying an MTD for the Flash Technology

To create a TrueFFS block device, you must call `tffsDevCreate()`. Internally, the call to `tffsDevCreate()` gives rise to the chain of function calls shown in Figure 3-2. One of the goals of these function calls is the identification of the MTD appropriate to the flash technology. This identification takes place in `flIdentifyFlash()`. This function steps through a table of `xxxIdentify()` routines, executing each routine until one succeeds. 1

---

1. The same MTD could be simultaneously active on multiple flash volumes.
Upon success, the identify routine updates the **FLFlash** structure with data and pointers to the MTDs routines for reading, writing, erasing, and mapping the flash media. Identification also completes the initialization of the **FLSocket** structure referenced within this **FLFlash** structure.

**Figure 3-2 Identifying an MTD**

```
    tffsDevCreate()
    
    flMountVolume()
    
    flMount()
    
    flIdentifyFlash()
    
    aaaIdentify()
    bbbIdentify()
    cccIdentify()
    ...
```

**NOTE:** Included with TrueFFS for Tornado are sources for several MTDs and socket component drivers. The MTDs are in `target/src/drv/tffs`. The socket component drivers are defined in the `sysTffs.c` files provided in the `target/config/bspname` directory for each BSP that supports TrueFFS.

### 3.2 About the **FLFlash** and **FLSocket** Structures

TrueFFS for Tornado is designed to handle a maximum of five TrueFFS block devices. Internally, TrueFFS allocates an **FLFlash** structure and an **FLSocket** structure for each of the five possible flash devices. The initialization of these
structures starts with the registration of your socket component driver functions with TrueFFS.

For the most part, this registration updates the FLSocket structure referenced by the socket member of FLFlash. The initialization of the FLFlash structure is completed by running an MTD identify routine. Because this identify routine depends on functions referenced in the FLSocket structure, you must install your socket component driver before you can run an MTD identify routine.

### 3.2.1 FLFlash

Almost all members of the FLFlash structure are set by the MTD identify routine, which gets most of the data it needs by probing the flash hardware. The only exception is the socket member, which is set by functions internal to TrueFFS. The FLFlash structure is defined in h/tffs/flflash.h as follows:

```c
typedef struct tFlash FLFlash; /* forward definition */
struct tFlash {
    FlashType    type; /* flash device type (JEDEC id) */
    long int    erasableBlockSize; /* smallest erasable area */
    long int    chipSize; /* chip size */
    int         noOfChips; /* no. of chips in array */
    int         interleaving; /* chip interleaving */
    unsigned    flags; /* special options */
    void *      mtdVars; /* MTD private area for socket */
    FLSocket *  socket; /* FLSocket for this drive */
    /* MTD-supplied flash map function */
    void FAR0 * (*map)(FLFlash *, CardAddress, int);
    /* MTD-supplied flash read function */
    FLStatus (*read)(FLFlash *, CardAddress, void FAR1 *, int, int);
    /* MTD-supplied flash write function */
    FLStatus (*write)(FLFlash *, CardAddress, const void FAR1 *, int, int);
    /* MTD-supplied flash erase function */
    FLStatus (*erase)(FLFlash *, int, int);
    /* callback to execute after power up */
    void (*setPowerOnCallback)(FLFlash *");
};

typedef struct tFlash FLFlash; /* forward definition */
struct tFlash {
    FlashType    type; /* flash device type (JEDEC id) */
    long int    erasableBlockSize; /* smallest erasable area */
    long int    chipSize; /* chip size */
    int         noOfChips; /* no. of chips in array */
    int         interleaving; /* chip interleaving */
    unsigned    flags; /* special options */
    void *      mtdVars; /* MTD private area for socket */
    FLSocket *  socket; /* FLSocket for this drive */
    /* MTD-supplied flash map function */
    void FAR0 * (*map)(FLFlash *, CardAddress, int);
    /* MTD-supplied flash read function */
    FLStatus (*read)(FLFlash *, CardAddress, void FAR1 *, int, int);
    /* MTD-supplied flash write function */
    FLStatus (*write)(FLFlash *, CardAddress, const void FAR1 *, int, int);
    /* MTD-supplied flash erase function */
    FLStatus (*erase)(FLFlash *, int, int);
    /* callback to execute after power up */
    void (*setPowerOnCallback)(FLFlash *");
};

The JEDEC ID for the flash memory hardware. This member is set by the MTD’s identification routine.

2. If you are not writing an MTD, you can ignore the member descriptions provided for the FLFlash structure and skip ahead to the description of the FLSocket structure.
erasableBlockSize
The size, in bytes, of an erase block for the attached flash memory hardware. This value takes interleaving into account. Thus, when setting this value in an MTD, the code is often of the form:

\[
\text{vol.erasableBlockSize} = aValue \times \text{vol.interleaving};
\]

Where \(aValue\) is the erasable block size of a flash chip that is not interleaved with another.

chipSize
The size (storage capacity), in bytes, of one of the flash memory chips used to construct the flash memory array. Set by the MTD using your \text{flFitInSocketWindow()} global routine.

noOfChips
The number of flash memory chips used to construct the flash memory array.

interleaving
The interleaving factor of the flash memory array. This must be a power of 2 (for example, 1, 2, 4). The interleaving is defined as the address difference on the media of two consecutive bytes on a chip. For most PCMCIA cards the interleaving is 2.

flags
Bits 0-7 are reserved for the use of TrueFFS (it uses these flags to track things such as the volume mount state). Bits 8-15 are reserved for the use of the MTDs.

mtdVars
This field, if used by the MTD, is initialized by the MTD identification routine to point to a private storage area. For example, the WRS-supplied MTD for the 16-bit AMD devices uses this member to store a pointer to an area containing AMD-specific flash parameters.

socket
A pointer to the \text{FLSocket} structure for this hardware device. This structure contains data and pointers to the socket layer functions that TrueFFS needs to manage the board interface for the flash memory hardware. The functions referenced in this structure are installed when you register your socket component driver.

Further, because TrueFFS uses these socket component driver functions to access the flash memory hardware, you must register your socket component driver before you try to run the MTD identify routine that initializes the bulk of this structure.

map
A pointer to the flash memory map function, the function that maps flash into an area of memory. Internally, TrueFFS initializes this member to point to a default map function appropriate for all NOR (linear) flash memory types. This default routine maps flash
memory through simple socket mapping. NAND or other type Flash should replace this pointer to the default routine with a reference to a routine that uses map-through-copy emulation. For more information on this function, see 3.4.2 Writing a Map Function for an MTD, p.53.

**read**  
A pointer to the flash memory read function. On entry to the MTD identification routine, this member has already been initialized to point to a default read function that is appropriate for all NOR (linear) flash memory types. This routine reads from flash memory by copying from a mapped window. If this is appropriate for your flash device, leave read unchanged. Otherwise, the MTD identify routine should update this member to point to a more appropriate function. For more information on this function, see 3.4.3 Writing a Read Function for an MTD, p.54.

**write**  
A pointer to the flash memory write function. Because of the dangers associated with an inappropriate write function, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member. For more information on this function, see 3.4.4 Writing Write and Erase Functions for an MTD, p.55.

**erase**  
A pointer to the flash memory erase function. Because of the dangers associated with an inappropriate erase function, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member. For more information on this function, see 3.4.4 Writing Write and Erase Functions for an MTD, p.55.

**setPowerOnCallback**  
A pointer to the function TrueFFS should execute after the flash hardware device powers up. TrueFFS calls this routine when it tries to mount a flash device. Do not confuse this member of FLFlash with the powerOnCallback member of the FLSocket structure. For many flash memory devices, no such function is necessary. However, this member is used by the MTD defined in target/src/drv/tffs/nfdc2048.c.

### 3.2.2 FLSocket

As a writer of a socket component driver, your primary concern is the initialization of an FLSocket structure. This structure provides TrueFFS with pointers to the
Writing Socket Component Drivers and MTDs

functions that you have written to handle the interface with the flash hardware. For the most part, the specifics of how you implement these functions are rather hardware specific.

When relevant, the member descriptions point out WRS-supplied BSPs that you can use as examples. However, your greatest asset in this situation is your intimate knowledge of your particular hardware. The **FLSocket** structure is defined in `h/tffs/flsocket.h` as follows:

```c
typedef struct tSocket FLSocket; /* forward definition */

struct tSocket {
    unsigned volNo;  /* volume no. of socket */
    unsigned serialNo;  /* serial no. of socket on controller */
    FLBoolean cardChanged;  /* need media change notification */
    int VccUsers;  /* no. of current VCC users */
    int VppUsers;  /* No. of current VPP users */
    PowerState VccState;  /* actual VCC state */
    PowerState VppState;  /* actual VPP state */
    FLBoolean remapped;  /* set to TRUE if the socket window is moved */
    void (*powerOnCallback)(void *flash); /* notification routine for Vcc on */
    void * flash;  /* flash object for callback */
    struct {  /* window state */
        unsigned int baseAddress;  /* physical base as a 4K page */
        unsigned int currentPage;  /* our current window page mapping */
        void FAR0 * base;  /* pointer to window base */
        long int size;  /* window size (must by power of 2) */
        unsigned speed;  /* in nsec. */
        unsigned busWidth;  /* 8 or 16 bits */
    } window;
    FLBoolean (*cardDetected)(FLSocket vol);
    void (*VccOn)(FLSocket vol);
    void (*VccOff)(FLSocket vol);
#ifdef SOCKET_12_VOLTS
    FLStatus (*VppOn)(FLSocket vol);
    void (*VppOff)(FLSocket vol);
#endif /* SOCKET_12_VOLTS */
    #ifdef EXIT
    void (*freeSocket)(FLSocket vol);
    #endif
} FLSocket;
```

**NOTE:** TrueFFS for Tornado always defines both `SOCKET_12_VOLTS` and `EXIT`. Thus, all the members shown above are always included in the TrueFFS for Tornado version of this structure.
volNo
For internal use. Do not change the value of this member except through an MTD identification routine. This member stores the volume number (drive number) associated with this socket component driver. Each socket component is assigned a driver number based on its order of registration. Thus, the first driver to register itself is assigned a 0, the second is assigned a 1, and so on up to 4. If you are using PCMCIA slots to add flash memory cards to your system, TrueFFS for Tornado creates a unique driver for each slot. Thus, each slot has its own volNo.

serialNo
A free-use member. This member is not used by any function internal to TrueFFS and is typically set to zero. Its name implies that it was intended as a convenient place to store the serial number associated with the flash device. Thus, you are free, for the most part, to use this member as you see fit. For example, the PCIC driver supplied with some WRS BSPs use this member to store slot numbers (0 and 1).

cardChanged
The intended purpose of this member is to track whether there was a change of flash card. To detect a change of card, TrueFFS polls the card every 100 milliseconds by calling the function you specify in the cardDetected member of this structure. If that function returns FALSE, TrueFFS sets this member to TRUE. This member is also set by the function referenced in getAndClearCardChangeIndicator.

For non-removable flash media, this member should always be set to FALSE. If this member is TRUE, when TrueFFS tries to access flash, it triggers a remount for the flash device.

VccUsers
For internal use. Do not change the value of this member. Each call to flNeedVcc() increments this counter internally in TrueFFS. The 100 millisecond timer routine uses this counter to verify that it is safe to call the VccOff() routine in the socket driver.

VppUsers
For internal use. Do not change the value of this member. Each call to flNeedVpp() increments this counter internally in TrueFFS. The 100 millisecond timer routine uses this counter to verify that it is safe to call the VppOff() routine in the socket driver.

VccState
For internal use. Do not change the value of this member outside the function supplied in FLSocket.VccOn. This member reports whether Vcc is in a PowerOn, a PowerGoingOff, or a PowerOff state.
VppState  For internal use. Do not change the value of this member outside the function supplied in FLSocket.VccOn. This member reports whether Vcc is in a PowerOn, a PowerGoingOff, or a PowerOff state.

remapped  For internal use. Do not change the value of this member except from within an MTD map function. TrueFFS uses this member to track whether or not the window was just remapped (moved). If you write your own MTD map function, that function should set this member before returning.

powerOnCallback
Set by your xxxRegister() function. TrueFFS calls the function referenced here after calling the function referenced in the VccOn member of this structure. If you have no need of this feature, set this member to a null pointer.

flash  For internal use. Do not change the value of this member. This member contains a pointer to the FLSocket structure associated with this socket component driver. It is needed to support removable flash media and is set dynamically by TrueFFS.

window.baseAddress  Set by your xxxRegister() function. In this context, a window is a part of host system memory through which a part of the media memory is directly visible and addressable. The window.baseAddress member stores the memory address of this host-system memory in terms of 4K pages (that is, the base address of the window divided by 4K). The value is stored in this way in order to guarantee that the base address (when reconstructed) is page aligned.

window.currentPage  For internal use. Do not change the value of this member. Stores the currently mapped window page (in 4KB units).

window.base
Set by the function you supply in the setWindow member of this structure. TrueFFS uses this member to store the base address of the memory window on flash memory.

window.size
Set by the function you supply in the setWindow member of this structure. TrueFFS uses this member to store the size of the memory window on flash memory.
**window.speed**

For internal use. You should not have to change the value of this member from your socket component driver. TrueFFS uses this member to store the length of time needed to complete an interaction with the flash device. Initially, TrueFFS sets this to a default value of 250 nano seconds. However, all WRS-supplied MTDs reset it to 120 nano seconds. If you write your own MTD, be sure to reset this member value appropriately in its mapping function.

**window.busWidth**

For internal use. Do not change the value of this member from your socket component driver. TrueFFS uses this member to record whether the flash device is 8 or 16 bits wide. Initially, TrueFFS sets this to a default value of 16. However, some WRS-supplied MTDs reset it to 8. If you write your own MTD, be sure to reset this member value appropriately in your mapping function.

**cardDetected**

Initialized by your `xxxRegister()` function. This member points to a function that reports whether there is a flash memory card in the PCMCIA slot associated with this device. For non-removable media, this routine should always return TRUE. Internally, TrueFFS for Tornado calls this function every 100 milliseconds to check that flash media is still there. If this function returns FALSE, TrueFFS sets `cardChanged` to TRUE.

**VccOn**

Set by your `xxxRegister()` function. This member contains a pointer to a function that TrueFFS can call to turn on the operating voltage (Vcc, usually 5 or 3.3 Volts) for the flash memory hardware. When the media is idle, TrueFFS conserves power by turning Vcc off. Prior to making a call that accesses flash memory, TrueFFS uses this function to turn the power back on again.

When switching Vcc on, the `VccOn` function must not return until Vcc has stabilized at the proper operating voltage. If necessary, your `VccOn` function should delay execution with an idle loop or with a call to the `flDelayMsec()` routine, until the Vcc has stabilized.

Prior to accessing the flash device, TrueFFS turns on Vcc. To conserve power, TrueFFS switches Vcc off at the completion of the operation. However, when socket polling is active, a delayed Vcc-off mechanism is used, in which Vcc is turned off only after at least one interval has passed. If several flash-accessing operations are executed in rapid sequence, Vcc remains on during the sequence, and is turned off only when TrueFFS goes into a relatively idle state.
VccOff  Set by your xxxRegister(). This member contains a pointer to a function that TrueFFS can call to turn off the operating voltage for the flash memory hardware. When the media is idle, TrueFFS conserves power by turning Vcc off. However, when socket polling is active, Vcc is turned off only after a delay. Thus, if several flags accessing operations are executed in rapid sequence, Vcc is left on during the sequence. Vcc is turned off only when TrueFFS goes into a relatively idle state.

VppOn   Set by your xxxRegister(). TrueFFS calls this routine to apply a programming voltage (Vpp, usually 12 Volts) to the flash chip. Because not all flash chips require this voltage, the member is included only if SOCKET_12_VOLTS is defined.

When switching Vpp on, the VppOn function must not return until Vpp has stabilized at the proper voltage. If necessary, your VppOn function should delay execution with an idle loop or with a call to the flDelayMsec() routine, until the Vpp has stabilized.

VppOff  Set by your xxxRegister(). TrueFFS calls this routine to turn off a programming voltage (Vpp, usually 12 Volts) to the flash chip. Because not all flash chips require this voltage, the member is included only if SOCKET_12_VOLTS is defined.

initSocket Set by your xxxRegister(). TrueFFS calls the function referenced here before it tries to access the socket. TrueFFS uses this function to handle any initialization that is necessary before accessing the socket, especially if that initialization was not possible at socket registration time. For example, if you did no hardware detection at socket registration time or if the flash memory medium is removable, this function should detect the flash memory medium and respond appropriately (including setting cardDetected to FALSE) if it is missing.

setWindow Set by your xxxRegister(). TrueFFS calls the function referenced in this member to update key members of the window structure included in this FLSocket structure. When writing a setWindow function for most hardware, you must do the following:

1. Set window.baseAddress to the base address in terms of 4K pages, that is, to the base address of the window divided by 4K. This is to guarantee that the base address is page aligned.
TrueFFS assumes that it has exclusive access to the window. That is, after it sets one of these window characteristics, it does not expect your application to directly change any of them, and could crash if you do. An exception to this is the mapping register. Because TrueFFS always reestablishes this register when it accesses flash memory, your application may map the window for purposes other than TrueFFS. However, you must not do this from an interrupt routine.

On systems with multiple socket drivers (to handle multiple flash devices), make sure that the window base address is different for each socket. In addition, you must take window size into account and verify that the windows do not overlap.

2. Call \texttt{flSetWindowSize()} and specify the window size in 4K units (\texttt{window.baseAddress}). Internally, the call to \texttt{flSetWindowSize()} sets \texttt{window.size} and \texttt{window.base} for you.

setMappingContext

Set by your \texttt{xxxRegister()} function. This member contains a pointer to a function that TrueFFS can call to set the window mapping register. Because board-resident flash arrays usually map the entire flash into RAM, they do not need this function. The known exception to is the flash array associated with the ss5 BSP. Flash cards in the PCMCIA slot can use this function to access a mapping register that moves the effective flash address into the host’s memory window (see the pc386 and pc486 BSPs).

getAndClearCardChangeIndicator

Set by your \texttt{xxxRegister()} function. This function reads the hardware card-change indication and clears it. It serves as a basis for detecting media-change events. If you have no such hardware capability, set this function pointer to NULL.

writeProtected

Set by your \texttt{xxxRegister()} function. This member contains a pointer to a function that TrueFFS can call to get the current state of the media’s write-protect switch (if available).

freeSocket

Set by your \texttt{xxxRegister()} function. This member contains a pointer to a function that TrueFFS can call to free the resources reserved internally to the socket driver.
3.3 Writing a Socket Component Driver

At start up, TrueFFS for Tornado creates a socket component driver for each flash device you want to support on your BSP. As shown in Figure 3-1, a call to \textit{tffsDrv()} eventually results in a call to \textit{sysTffsInit()}. This routine must do all that is necessary to register a socket component driver for each flash device attached to your embedded system.

For the most part, this means appropriating an \textit{FLSocket} structure (previously allocated internally to TrueFFS for Tornado) and installing pointers to functions in your socket component driver. The details of how you handle this are up to you. Strictly speaking, there is no organization imposed on the internals of \textit{sysTffsInit()}. However, to allow for scalability, the details specific to each flash device are typically isolated to \textit{xxxRegister()} functions, one for each flash device on your system.

Finally, your socket component driver must provide two global functions: \textit{flFitInSocketWindow()} and \textit{flDelayLoop()}. These functions are global in the sense that they must be externally callable (and thus must not be declared as \textit{LOCAL} or any other equivalent to \textit{static}).

The \textit{flFitInSocketWindow()} routine returns the size (in bytes) of an individual chip in the flash array. TrueFFS uses this value to determine whether its socket window is big enough. The \textit{flDelayLoop()} routine gives TrueFFS a hardware-appropriate way to wait for a specific number of milliseconds.

3.3.1 Writing a \textit{sysTffsInit()} Routine

The \textit{sysTffsInit()} function takes no arguments and returns void. Internally, it must, at a minimum, call the socket registration routine for each flash device you want registered. For an example of the simplest possible \textit{sysTffsInit()}, consider the \textit{sysTffsInit()} defined in the \textit{sysTffs.c} for the \textit{mv177} BSP:

\begin{verbatim}
LOCAL void sysTffsInit (void)
{
    rfaRegister ();
}
\end{verbatim}

This BSP supports only one flash device. Thus, its \textit{sysTffsInit()} contains only a call to \textit{rfaRegister()}, a registration routine for a resident flash array, an on-board non-removable flash device. Other BSPs define slightly more complicated \textit{sysTffsInit()}
routines. For example, the `sysTffsInit()` routine in the pc486 BSP makes three `xxxRegister()` calls:

```c
#ifdef INCLUDE_SOCKET_DOC
(void) docRegister (); /* Disk On Chip */
#endif /* INCLUDE_SOCKET_DOC */

#ifdef INCLUDE_SOCKET_PCIC0
(void) pcRegister (0, PC_BASE_ADRS_0); /* flash card on socket 0 */
#endif /* INCLUDE_SOCKET_PCIC0 */

#ifdef INCLUDE_SOCKET_PCIC1
(void) pcRegister (1, PC_BASE_ADRS_1); /* flash card on socket 1 */
#endif /* INCLUDE_SOCKET_PCIC1 */
```

Note that each `xxxRegister()` call is encapsulated in pre-processor conditional statements. The constants that control these statements are defined in the BSP’s `config.h`. Using these constants, you can selectively control which calls are included in `sysTffsInit()` at compile time.

### 3.3.2 Writing an `xxxRegister()` Routine

Internally, TrueFFS allocates an array of `FLSocket` structures. TrueFFS uses these structures to find the socket component driver functions (and other information) that it needs to interact with the flash hardware. Within your `xxxRegister()` routine, you must appropriate one of these `FLSocket` structures and initialize its members appropriately. To appropriate an `FLSocket` structure from TrueFFS, call `flSocketOf()`:

```c
FLSocket *flSocketOf(unsigned volNo)
```

As input, this function expects a drive number (valid values range from 0 to 4, inclusive). Upon completion, this function returns a pointer to an `FLSocket` structure as its value. Consider the following code fragment:

```c
FLSocket vol = flSocketOf (noOfDrives);
if (noOfDrives >= DRIVES) return (flTooManyComponents);
tffsSocket[noOfDrives] = "RFA";
noOfDrives++;
```

Both `noOfDrives` and `tffsSocket[]` are global. `DRIVES` is a symbolic constant set to the maximum number of drivers (currently 5, do not attempt to change this value). In your call to `flSocketOf()`, you must use `noOfDrives` the global variable for the `volNo` parameter, and you must update it after successfully appropriating an `FLSocket` structure. This global variable is used internally by TrueFFS for Tornado to keep count of the flash memory logical devices it has created.
Likewise, you should update the `noOfDrives` element of the `tffsSocket[]` global array to contain a label for the drive you just created (this label is used in the output from the `tffsShow()` and `tffsShowAll()` routines).

After retrieving this `FLSocket` pointer, your `xxxRegister()` routine must set the values of the following members:

- `window.baseAddress`
  See `window.baseAddress`, p.43.
- `cardDetected`
  See `cardDetected`, p.44.
- `VccOn`
  See `VccOn`, p.44.
- `VccOff`
  See `VccOff`, p.45.
- `VppOn`
  See `VppOn`, p.45.
- `VppOff`
  See `VppOff`, p.45.
- `initSocket`
  See `initSocket`, p.45.
- `setWindow`
  See `setWindow`, p.45.
- `setMappingContext`
  See `setMappingContext`, p.46.
- `getAndClearCardChangeIndicator`
  See `getAndClearCardChangeIndicator`, p.46.
- `writeProtected`
  See `writeProtected`, p.46.
- `freeSocket`
  See `freeSocket`, p.46.
3.3.3 Writing an flDelayLoop() Routine

TrueFFS uses flDelayLoop() to handle the timing needs of interacting with the flash hardware. The purpose of this function is simply to wait for the number of cycles requested. This function must be of the form:

```c
void flDelayLoop(int)
```

For the mv177 BSP, flDelayLoop() is defined as followed:

```c
void flDelayLoop
{
    int cycles /* loop count to be consumed */
}
{
    while (--cycles);
}
```

3.3.4 Writing an flFitInSocketWindow() Routine

TrueFFS uses flFitInSocketWindow() to make sure that chipSize is never larger than windowSize. Your flFitInSocketWindow() must be of the form:

```c
long int flFitInSocketWindow
{
    long int chipSize, /* size of single physical chip in bytes */
    int interleaving, /* flash chip interleaving (1,2,4 etc) */
    long int windowSize /* socket window size in bytes */
}
```

If your BSP uses a sliding window or if it maps all the flash into the window, your definition for this function can simply return chipSize as its function value.

If your BSP uses a fixed window that is not large enough to map the entire flash medium into host memory, your flFitInSocketWindow() function must compare chipSize and the windowSize. If chipSize is smaller than or equal to windowSize, this function can return chipSize as its function value. If chipSize is larger than windowSize, this function should return a size that fits within windowSize.

The MTD mapping function uses the returned value of the flFitInSocketWindow() function to adjust the value of FLSocket.chipSize. For an example of a BSP that defines a non-trivial flFitInSocketWindow() function, look at the mv177 BSP.
3.4 Writing an MTD

An MTD is a software module that provides the basic read, write, erase, and map functionality that TrueFFS needs in order to program flash memory. This module also provides an identification utility, a routine that probes a flash device for its type. If the flash type is appropriate to the MTD, the MTD provides TrueFFS with data and pointers to the functions (map, read, write, and erase) that it needs to program that flash device.

In the process of creating a logical block device for a flash memory array, TrueFFS tries to match an MTD to the flash device. To do this, TrueFFS calls the identification routine from each MTD until one reports a match. The first reported match is the one taken. If no MTD reports a match, TrueFFS falls back on a default read-only MTD that reads from the flash device by copying from the socket window.

Assumptions You Can Make within an MTD Function

When any MTD function is called, assume the following:

- There is a card in the socket.
- Vcc is on.
- Vpp is off (until you turn it on).
- The FLSocket passed into the function belongs to the currently active drive.

3.4.1 Writing an MTD Identification Routine

The MTD identification routine is guaranteed to be called prior to any other routine in the MTD. An MTD identification routine is of the following format:

`FLStatus xxxIdentify(FLFlash vol)`

Within an MTD identify routine, you must probe the device to determine its type. How you do this depends on the hardware. If the type is not appropriate to this MTD, return failure. Otherwise, set the members of the FLFlash structure listed below:

- `type` See `type`, p.38.
- `erasableBlockSize` See `erasableBlockSize`, p.39.
- `chipSize` See `chipSize`, p.39.
TrueFFS for Tornado
Programmer's Guide

**noOfChips**

See noOfChips, p.39.

See also the description of **flIntelSize()** given below.

**interleaving**

See interleaving, p.39.

See also the description of **flIntelIdentify()** given below.

**write**

See 3.4.4 Writing Write and Erase Functions for an MTD, p.55, and write, p.40.

**erase**

See 3.4.4 Writing Write and Erase Functions for an MTD, p.55, and erase, p.40.

**map**

See 3.4.2 Writing a Map Function for an MTD, p.53, and map, p.39.

**read**

See 3.4.3 Writing a Read Function for an MTD, p.54, and read, p.40.

After setting the members listed above, this function should return flOK.

---

**Special Utilities for Use with NOR Flash Arrays**

For most NOR flash arrays, you can use the standard routines **flIntelIdentify()** and **flIntelSize()** to automatically detect the JEDEC id, the interleaving, and the size of the flash array. Call the **flIntelIdentify()** routine first. Its interface is:

```c
FLStatus flIntelIdentify(
    FLFlash vol,
    void (*amdCmdRoutine) (FLFlash *, long int, unsigned char),
    long int chipOffset)
```

**flIntelIdentify()** uses the chip’s READ_ID command to detect the device and retrieve its JEDEC id and chip interleaving. Set **amdCmdRoutine** to NULL for Intel flash devices. For AMD & Fujitsu flash devices, set **amdCmdRoutine** to the address of a routine that writes a command to the flash device through the special “unlock sequence” that is necessary for this kind of flash. Use **chipOffset** to pass in the offset on the chip that is used for reading the chip id. A value of 0 (offset zero) is almost always appropriate.

If **flIntelIdentify()** fails, your MTD identification routine should immediately return failure. If **flIntelIdentify()** succeeds, it sets **vol.type** and **vol.interleaving**.

Your MTD identification routine should then check the JEDEC id stored in

---

3. Both **flIntelIdentify()** and **flIntelSize()** assume 8-bit access to the flash array. If this is inappropriate for your particular flash device, you could directly set the values of the relevant members of the **vol** structure.
vol.type. If your MTD can handle this type of flash, it should set vol.chipSize to its correct value, and then call **flIntelSize()**, defined as:

```c
FLStatus flIntelSize(
    FLFlash vol,
    void (*amdCmdRoutine) (FLFlash *, long int, unsigned char),
    long int chipOffset)
```

This function is able to detect the array size correctly even when an address wraparound exists at the end of the flash array, although all chips in the flash array must be of the same type. If this function succeeds, it sets vol.noOfChips correctly. If this routine fails, your MTD identification routine must return failure.

**Reading the CIS through flMap()**

Often an MTD needs to identify a PCMCIA flash card through its CIS or Attribute Memory. To access Attribute Memory, use the **flMap()** function, as described in 3.4.2 **Writing a Map Function for an MTD**, p.53. The address parameter is the Attribute Memory address plus 128 M. For example:

```c
void far *cis = flMap(vol, 0x8000000l);
```

Note that both the socket hardware and the card must recognize a separate Attribute Memory Space. If one of them does not, you will find yourself mapped to the equivalent address in common memory.

### 3.4.2 Writing a Map Function for an MTD

On many systems, the flash array is connected to the system bus through an intelligent controller (PCMCIA, for example). From the programmer’s perspective, this connection makes it appear as if there is a host address space window that is sliding along the flash array. This feature is used extensively by TrueFFS to optimize read operations on NOR flash array when TrueFFS needs to access system meta data.

Because system meta data is distributed evenly across the region of the flash array, TrueFFS needs to issue many calls to the MTD read routine in order to collect the data. Thus it is more efficient to map an entire region of the flash array into the host address space (using the MTD map routine) and then read directly from the memory window.

Initially, TrueFFS assigns **FLFlash.map** to point to a default mapping function called **flashMap()**. This default function is appropriate for flash devices that can
be mapped directly into host memory. Internally, this function makes only one call, a call to \texttt{flMap}:

\begin{verbatim}
void FAR0 *flMap ( FLSocket vol, CardAddress address )
\end{verbatim}

This function maps the flash window (described in the submitted \texttt{FLSocket} structure) to a specified card address and returns a pointer to that location, which is at some offset within the window.

After calling \texttt{flMap()}, the specified part of the flash array is mapped to the socket window and is effectively a part of the local host memory. Thus, it is accessible through normal addressing. To address subsequent bytes on the card, increment the \texttt{address} pointer—but only so far as the socket window extends in host memory. At some offset you are no longer addressing memory on the card. Because the minimum window size is 4 Kilobytes, a safe practice is to never cross a 4 KByte boundary when adding offsets to a window location.

Of course, if the entire flash array is visible in the host address space, the map routine need do nothing more than return a pointer to the specified address in flash memory.

Unfortunately, not all flash devices allow this direct mapping of flash memory into host memory. For some devices, it is necessary to map through a buffer. For example, the MTD defined in \texttt{target/src/drv/tffs/nfdc2048.c} takes this approach. The identification routine of this MTD assigns \texttt{FLFlash.map} to \texttt{cdsnMap}, a pointer to \texttt{cdsnMap()}, which is defined as follows:

\begin{verbatim}
static void FAR0 * cdsnMap ( FLFlash vol, CardAddress address, int length )
{
  cdsnRead(&vol,address,thisBuffer,length, 0);
  vol.socket->remapped = TRUE;
  return (void FAR0 *)thisBuffer;
}
\end{verbatim}

There is a common case of RFA fully visible in the host address space. In this marginal case, the only thing the map routine needs to do is to return a pointer to the specified address in flash memory.

\subsection{3.4.3 Writing a Read Function for an MTD}

If the flash device can be mapped directly into flash memory, it is generally a simple matter to read from it. TrueFFS supplies a default function that does a
remap and simple memory copy to retrieve the data from the specified area. If the mapping is done through a buffer, you must provide your own read routine.

Typically, you should write your read routine as generically as possible. That is, it should read only a character or a word or a long word at a time, and that it should be able to handle an unaligned read as well as a read that crosses chip boundaries.

For an example of an MTD that provides its own read routine, see the `cdsnRead()` function defined in `target/src/drv/tffs/nfdc2048.c`.

### 3.4.4 Writing Write and Erase Functions for an MTD

Your write and erase functions should be as generic as possible. That is, the write should write only a character or a word or a long word at a time, and it should be able to handle an unaligned write as well as a write that crosses chip boundaries. Similarly, the erase function should be able to handle an erase that crosses chip boundaries. As input, the erase can expect a block number. Use the value of the `erasableBlockSize` member of the `FLFlash` structure to translate this block number to the offset within the flash array.

When writing these functions, you probably want to use the MTD helper functions `flNeedVpp()`, `flDontNeedVpp()`, and `flWriteProtected()`. The interfaces for these routines are as follows:

```c
FLStatus flNeedVpp(FLSocket vol)
void flDontNeedVpp(FLSocket vol)
FLBoolean flWriteProtected(FLSocket vol)
```

Use `flNeedVpp()` if you need to turn on the Vpp (programming voltage) for the chip. Internally, `flNeedVpp()` bumps a counter, `FLSocket.VppUsers`, and then calls the function referenced in `FLSocket.VppOn`. After calling `flNeedVpp()`, check its return status to verify that it succeeded in turning on Vpp.4

When done with the write or erase that required Vpp, call `flDontNeedVpp()` to decrement the `FLSocket.VppUsers` counter. This `FLSocket.VppUsers` counter is part of a delayed-off system. While the chip is busy, TrueFFS keeps the chip continuously powered. When the chip is idle, TrueFFS turns off the voltage to conserve power.

Use `flWriteProtected()` to test that the flash device is not write protected. MTD write and MTD erase routines must not do any flash programming before checking

---

4. An MTD does not need to touch Vcc. TrueFFS turns Vcc on before calling an MTD function.
that writing to the card is allowed. The Boolean function `flWriteProtected()` returns TRUE if the card is write-protected and FALSE otherwise.

### 3.4.5 Registering Your MTD with TrueFFS

When TrueFFS tries to match an MTD to a flash device, it executes the functions referenced in `mtdTable[]`, which is defined in `target/src/drv/tffs/tffsConfig.c` as follows:

```c
MTDidentifyRoutine mtdTable[] = /* MTD tables */
{
    #ifdef INCLUDE_MTD_I28F016
        i28f016Identify,
    #endif /* INCLUDE_MTD_I28F016 */

    #ifdef INCLUDE_MTD_I28F008
        i28f008Identify,
    #endif /* INCLUDE_MTD_I28F008 */

    #ifdef INCLUDE_MTD_AMD
        amdMTDIdentify,
    #endif /* INCLUDE_MTD_AMD */

    #ifdef INCLUDE_MTD_CDSN
        cdsnIdentify,
    #endif /* INCLUDE_MTD_CDSN */

    #ifdef INCLUDE_MTD_DOC2
        doc2Identify,
    #endif /* INCLUDE_MTD_DOC2 */

    #ifdef INCLUDE_MTD_CFISCS
        cfiscsIdentify,
    #endif /* INCLUDE_MTD_CFISCS */
};
```

If you write a new MTD and want to make it available to TrueFFS, you must add its registration routine to this table. When doing so, you should probably surround the function name with conditional include statements. The symbolic constants that control these conditional includes are defined in the BSP's `config.h`. Using these constants, your end users can conditionally include specific MTDs. When you add your MTDs identification routine to this table, you should also add a new constant to the BSP's `config.h`.

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set these constants by using the Tornado project facility. See the *Tornado User's Guide: Projects*. 

---

56
3.5 MTD-Supported Flash Devices

This section tells you which flash devices are supported by which shipped MTD.

Intel 28F008 Flash Support

The MTD defined in `I28F008.c` supports the Intel 28F008SA, Intel 28F008SC and Intel 28F016SA/SV (in 8-mbit compatibility mode) flash components. Any flash array or card based on these chips is recognized and supported by this MTD. However, the WORD-mode of 28F016SA/SV is not supported (BYTE-mode only). This MTD also supports all interleaving factors (1, 2, 4, ...). Interleaving of more than 4 is recognized, although the MTD does not access more than 4 flash parts simultaneously. The list of supported flash media includes the following:

- M-Systems D-Series PC Cards
- M-Systems S-Series PC Cards
- Intel Series-2 (8-mbit family) PC Cards
- Intel Series-2+ (16-mbit family) PC Cards
- Intel Value Series 100 PC Cards
- Intel Miniature cards
- M-Systems PC-FD, PC-104-FD, Tiny-FD Flash disks

Define `INCLUDE_MTD_I28F008` in your BSP’s `config.h` to include this MTD in TrueFFS for Tornado.

NOTE: Instead of editing `config.h`, it may be to your advantage to set this constant by using the Tornado project facility. See the Tornado User’s Guide: Projects.

Intel 28F016 Flash Support

The MTD defined in `I28F016.c` supports Intel 28F016SA and Intel 28F008SV flash components. Any flash array or card based on these chips is recognized and supported by this MTD. This MTD also supports interleaving factors of 2 and 4 for BYTE-mode 28F016 component access. For WORD-mode component access, only non-interleaved (interleave 1) mode is supported. The list of supported flash media includes the following:

- Intel Series-2+ PC Cards
- M-Systems Series-2+ PC Cards
Define `INCLUDE_MTD_128F016` in your BSP’s `config.h` to include this MTD in TrueFFS for Tornado.

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set this constant by using the Tornado project facility. See the *Tornado User’s Guide: Projects*.

### CFI/SCS Flash Support

The MTD defined in `cfiscs.c` supports flash components that follow the CFI/SCS specification. CFI stands for Common Flash Interface. SCS stands for Scalable Command Set. CFI is a standard method for querying flash components for their characteristics. SCS is a second layer built on the CFI specification. This lets a single MTD handle all CFI/SCS Flash technology in a common manner.

The joint CFI/SCS specification is currently adopted by Intel Corporation and Sharp Corporation for all new flash components starting 1997. The CFI and SCS specifications can be obtained from Intel Corporation at the following address:

Kurt Robinson  
Intel Corporation  
1900 Prairie City Road, Mailstop FM3-121  
Folsom, Ca. 95630-9598

Define `INCLUDE_MTD_CFISCSCS` in your BSP’s `config.h` to include this MTD in TrueFFS for Tornado.

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set this constant by using the Tornado project facility. See the *Tornado User’s Guide: Projects*.

### AMD/Fujitsu Flash Support

The MTDs defined in `amdmtd.c` (8-bit) supports AMD flash components of the AMD Series-C and Series-D flash technology family, as well as the equivalent Fujitsu flash components. The flash types supported are:

- Am29F040  (JEDEC id’s 01a4h, 04a4h)
- Am29F080  (JEDEC id’s 01d5h, 04d5h)
- Am29LV080  (JEDEC id’s 0138h, 0438h)
- Am29F008  (JEDEC id’s 0137h, 0437h)
- Am29F016  (JEDEC id’s 01adh, 04adh)
- Am29F016C  (JEDEC id’s 013dh, 043dh)
Any flash array or card based on these chips is recognized and supported by this MTD. The MTD supports interleaving factors of 1, 2 and 4. The list of supported flash media includes the following:

- AMD & Fujitsu Series-C PC cards
- AMD & Fujitsu Series-D PC cards
- AMD & Fujitsu miniature cards

Define `INCLUDE_MTD_AMD` in your BSP’s `config.h` to include the 8-bit MTD in TrueFFS for Tornado.

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set this constant by using the Tornado project facility. See the *Tornado User's Guide: Projects*.

### NAND Flash Support (NFDC2048)

The MTD defined in `nfdc2048.c` supports Toshiba and Samsung NAND flash. The MTD interfaces to the flash components through M-Systems’ NFDC2048 ASIC controller, or through CDSN (Common Design Standard for NAND) on which the NFDC2048 controller is based. The flash components supported are:

- Toshiba TC5816 16-mbit NAND Flash
- Toshiba TC5832 32-mbit NAND Flash
- Toshiba TC5864 64-mbit NAND Flash
- Samsung KM29N16000 16-mbit NAND Flash
- Samsung KM29N32000 32-mbit NAND Flash
- Samsung KM29N64000 64-mbit NAND Flash
- National Semiconductor NM29N16 16-mbit NAND Flash
- National Semiconductor NM29N32 32-mbit NAND Flash
- National Semiconductor NM29N64 64-mbit NAND Flash

This MTD supports M-Systems Series-2000 PC Cards.

Define `INCLUDE_MTD_CDSN` in your BSP’s `config.h` to include this MTD in TrueFFS for Tornado.

**NOTE:** Instead of editing `config.h`, it may be to your advantage to set this constant by using the Tornado project facility. See the *Tornado User’s Guide: Projects*. 
Disk-On-Chip 2000 Support (NFDC2148)

The MTD defined in nfdc2148.c supports Toshiba and Samsung NAND flash. The MTD interfaces to the flash components through M-Systems’ NFDC2148 ASIC controller. The flash components supported are:

- Toshiba TC5816 16-mbit NAND Flash
- Toshiba TC5832 32-mbit NAND Flash
- Toshiba TC5864 64-mbit NAND Flash
- Samsung KM29N16000 16-mbit NAND Flash
- Samsung KM29N32000 32-mbit NAND Flash
- Samsung KM29N64000 64-mbit NAND Flash
- National Semiconductor NM29N16 16-mbit NAND Flash
- National Semiconductor NM29N32 32-mbit NAND Flash
- National Semiconductor NM29N64 64-mbit NAND Flash

This MTD supports M-Systems DiskOnChip-2000.

Define INCLUDE_MTD_DOC2 in your BSP’s config.h to include this MTD in TrueFFS for Tornado.

NOTE: Instead of editing config.h, it may be to your advantage to set this constant by using the Tornado project facility. See the Tornado User’s Guide: Projects.
### Libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysTffs</td>
<td>BSP-specific TrueFFS library</td>
<td>62</td>
</tr>
<tr>
<td>tffsConfig</td>
<td>TrueFFS configuration file for VxWorks</td>
<td>64</td>
</tr>
<tr>
<td>tffsDrv</td>
<td>TrueFFS interface for VxWorks</td>
<td>65</td>
</tr>
</tbody>
</table>
sysTffs

NAME
sysTffs - BSP-specific TrueFFS library

SYNOPSIS
flFitInSocketWindow() - check if flash array fits in socket window
flDelayLoop() - consume the specified time

long int flFitInSocketWindow
   (long int chipSize, int interleaving, long int windowSize)

void flDelayLoop
   (int cycles)

DESCRIPTION
This library provides board-specific hardware access routines for TrueFFS. In effect, these
routines comprise the socket component driver (or drivers) for your flash device
hardware. At socket registration time, TrueFFS stores pointers to the functions of this
socket component driver in an FLSocket structure. When TrueFFS needs to access the
flash device, it uses these functions.

Because this file is, for the most part, a device driver that exports its functionality by
registering function pointers with TrueFFS, very few of the functions defined here are
externally callable. For the record, these external functions are flFitInSocketWindow() and
flDelayLoop(). You should never have any need to call these functions.

However, one of the most important functions defined in this file is neither referenced in an
FLSocket structure, nor is it externally callable. This function is sysTffsInit(). TrueFFS
calls this function at initialization time to register socket component drivers for all the
flash devices attached to your target. It is this call to sysTffs() that results in assigning
drive numbers to the flash devices on your target hardware. Drive numbers are assigned
by the order in which the socket component drivers are registered. The first to be
registered is drive 0, the second is drive 1, and so on up to four. As shipped, TrueFFS
supports up to five flash devices.

After registering socket component drivers for a flash device, you may format the flash
medium even though there is not yet a block device driver associated with the flash (see
the reference entry for the tffsDevCreate() routine). To format the flash medium for use
with TrueFFS, call tffsDevFormat() or, for some BSPs, sysTffsFormat().

The sysTffsFormat() routine is an optional but BSP-specific externally callable helper
function. Internally, it calls tffsDevFormat() with a pointer to a FormatParams structure
initialized to values that leave a space on the flash device for a boot image. This space is
outside the region managed by TrueFFS. This special region is necessary for boot images
because the normal translation and wear-leveling services of TrueFFS are incompatible
with the needs of the boot program and the boot image it relies upon. To write a boot
image (or any other data) into this area, use tffsBootImagePut().
Finally, this file also contains define statements for symbolic constants that determine which MTDs, translation layer modules, and other utilities are ultimately included in TrueFFS. These defines are as follows:

**INCLUDE_TL_NFTL**
- To include the NAND-based translation layer module.

**INCLUDE_TL_FTL**
- To include the NOR-based translation layer module.

**INCLUDE_TL_SSFDC**
- To include the SSFDC-appropriate translation layer module.

**INCLUDE_MTD_128F016**
- For Intel 28F016 flash devices.

**INCLUDE_MTD_128F008**
- For Intel 28F008 flash devices.

**INCLUDE_MTD_128F008_BAJA**
- For Intel 28F008 flash devices on the Heurikon Baja 4700.

**INCLUDE_MTD_AMD**
- For AMD, Fujitsu: 29F0{40,80,16} 8-bit flash devices.

**INCLUDE_MTD_CDSN**
- For Toshiba, Samsung: NAND CDSN flash devices.

**INCLUDE_MTD_DOC2**
- For Toshiba, Samsung: NAND DOC flash devices.

**INCLUDE_MTD_CFI/SCS**
- For CFI/SCS flash devices.

**INCLUDE_MTD_WAMD**
- For AMD, Fujitsu 29F0{40,80,16} 16-bit flash devices.

**INCLUDE_TFFS_BOOT_IMAGE**
- To include `tffsBootImagePut()` in TrueFFS for Tornado.

To exclude any of the modules mentioned above, edit `sysTffs.c` and undefine its associated symbolic constant.

**INCLUDE FILES**
- `flsocket.h`
- `tffsDrv.h`

**SEE ALSO**
- `tffsDrv`
- `tffsConfig`
tffsConfig

NAME
tffsConfig - TrueFFS configuration file for VxWorks

SYNOPSIS

tffsShowAll() - show device information on all socket interfaces
tffsShow() - show device information on a specific socket interface
tffsBootImagePut() - write to the boot-image region of the flash device

void tffsShowAll
  (void)

void tffsShow
  (int driveNo)

STATUS tffsBootImagePut
  (int driveNo, int offset, char * filename)

DESCRIPTION

This source file, with the help of sysTffs.c, configures TrueFFS for VxWorks. The
functions defined here are generic to all BSPs. To include these functions in the BSP-
specific module, the BSP’s sysTffs.c file includes this file. Within the sysTffs.c file, define
statements determine which functions from the tffsConfig.c file are ultimately included in
TrueFFS.

The only externally callable routines defined in this file are tffsShow(), tffsShowAll(),
and tffsBootImagePut(). You can exclude the show utilities if you edit config.h and
undefine INCLUDE_SHOW_ROUTINES. You can exclude tffsBootImagePut() if you edit
sysTffs.c and undefine INCLUDE_TFFS_BOOT_IMAGE. (If you find these utilities
are missing and you want them included, edit config.h and define
INCLUDE_SHOW_ROUTINES and INCLUDE_TFFS_BOOT_IMAGE.)

However, for the most part, these externally callable routines are only a small part of the
TrueFFS configuration needs handled by this file. The routines internal to this file make
calls into the MTDs and translation layer modules of TrueFFS. At link time, resolving the
symbols associated with these calls pulls MTD and translation layer modules into
VxWorks.

However, each of these calls to the MTDs and the translation layer modules is only
conditionally included. The constants that control the includes are defined in sysTffs.c. To
exclude an MTD or translation layer module, you edit sysTffs.c, undefine the appropriate
constant, and rebuild sysTffs.o. These constants are described in the reference entry for
sysTffs.

INCLUDE FILES
stdcomp.h
**tffsDrv**

**NAME**
tffsDrv - TrueFFS interface for VxWorks

**SYNOPSIS**
tffsDrv() - initialize the TrueFFS system
tffsDevCreate() - create a TrueFFS block device suitable for use with dosFs
tffsDevFormat() - format a flash device for use with TrueFFS
tffsRawio() - low level I/O access to flash components

**STATUS** tffsDrv
(void)

BLK_DEV * tffsDevCreate
(int tffsDriveNo, int removableMediaFlag)

**STATUS** tffsDevFormat
(int tffsDriveNo, int arg)

**STATUS** tffsRawio
(int tffsDriveNo, int functionNo, int arg0, int arg1, int arg2)

**DESCRIPTION**
This module defines the routines that VxWorks uses to create a TrueFFS block device. Using this block device, dosFs can access a board-resident flash memory array or a flash memory card (in the PCMCIA slot) just as if it was a standard disk drive. Also defined in this file are functions that you can use to format the flash medium, as well as functions that handle low-level I/O to the device.

To include TrueFFS for Tornado in a VxWorks image, you must edit your BSP’s config.h and define INCLUDE_TFFS, or, for some hardware, INCLUDE_PCMCIA. This configures the VxWorks initialization code to call tffsDrv(). This call sets up the structures, the global variables, and the mutual exclusion semaphore needed to manage TrueFFS. This call to tffsDrv() also registers socket component drivers for each flash device found attached to the target.

These socket component drivers are not quite block devices, but they are an essential layer within TrueFFS. Their function is to manage the hardware interface to the flash device, and they are intelligent enough to handle formatting and raw I/O requests to the flash device. The other two layers within TrueFFS are known as the translation layer and the MTD (the Memory Technology Driver). The translation layer of TrueFFS implements the error recover and wear-leveling features of TrueFFS. The MTD implements the low-level programming (map, read, write, and erase) of the flash medium.

To implement the socket layer, each BSP that supports TrueFFS includes a sysTffs.c file. This file contains the code that defines the socket component driver. This file also contains a set of defines that you can use to configure which translation layer modules and MTDs are included in TrueFFS. Which translation layer modules and MTDs you should include depends on which types of flash devices you need to support. Currently, there are three
basic flash memory technologies, NAND-based, NOR-based, and SSFDC. Within sysTffs.c, define:

**INCLUDE_TL_NFTL**
- To include the NAND-based translation layer module.

**INCLUDE_TL_FTL**
- To include the NOR-based translation layer module.

**INCLUDE_TL_SSFDC**
- To include the SSFDC-appropriate translation layer module.

To support these different technologies, TrueFFS ships with three different implementations of the translation layer. Optionally, TrueFFS can include all three modules. TrueFFS later binds the appropriate translation layer module to the flash device when it registers a socket component driver for the device.

Within these three basic flash device categories there are still other differences (largely manufacturer-specific). These differences have no impact on the translation layer. However, they do make a difference for the MTD. Thus, TrueFFS ships with a variety of different MTDs that can support a wide range of flash devices from Intel, Sharp, Samsung, National, Toshiba, AMD, and Fujitsu. Within sysTffs.c, define:

**INCLUDE_MTD_I28F016**
- For Intel 28f016 flash devices.

**INCLUDE_MTD_I28F008**
- For Intel 28f008 flash devices.

**INCLUDE_MTD_I28F008_BAJA**
- For Intel 28f008 flash devices on the Heurikon Baja 4000.

**INCLUDE_MTD_AMD**
- For AMD, Fujitsu: 29F0{40,80,16} 8-bit flash devices.

**INCLUDE_MTD_CDSN**
- For Toshiba, Samsung: NAND CDSN flash devices.

**INCLUDE_MTD_DOC2**
- For Toshiba, Samsung: NAND DOC flash devices.

**INCLUDE_MTD_CFISCs**
- For CFI/SCS flash devices.

**INCLUDE_MTD_WAMD**
- For AMD, Fujitsu 29F0{40,80,16} 16-bit flash devices.

The socket component driver and the MTDs are provided in source form. If you need to write your own socket driver or MTD, use these working drivers as a model for your own.
EXTERNALLY CALLABLE ROUTINES

Most of the routines defined in this file are accessible through the I/O system only. However, four routines are callable externally. These are: tffsDrv(), tffsDevCreate(), tffsDevFormat(), and tffsRawio().

The first routine called from this library must be tffsDrv(). Call this routine exactly once. Normally, this is handled automatically for you from within usrRoot(), if INCLUDE_TFFS is defined, or from within pccardTffsEnabler(), if INCLUDE_PCMCIA is defined.

Internally, this call to tffsDrv() registers socket component drivers for all the flash devices connected to your system. After registering a socket component driver for the device, TrueFFS can support calls to tffsDevFormat() or tffsRawio(). However, before you can mount dosFs on the flash device, you must call tffsDevCreate(). This call creates a block device on top of the socket component driver, but does not mount dosFs on the device. Because mounting dosFs on the device is what you will want to do most of the time, the sysTffs.c file defines a helper function, usrTffsConfig(). Internally, this function calls tffsDevCreate() and then does everything necessary (such as calling the dosFsDevInit() routine) to mount dosFs on the resulting block device.

LOW LEVEL I/O

Normally, you should handle your I/O to the flash device using dosFs, which, in turn, uses TrueFFS and its component drivers to read and write from the flash device. However, there are situations in which this level of indirection can cause problems. To handle such situations, this library defines tffsRawio(). Using this function, you can bypass both dosFs and the TrueFFS translation services to program the flash medium directly.

However, you should not try to program the flash device directly unless you are intimately familiar with the physical limits of your flash device as well as with how TrueFFS formats the flash medium. Otherwise you risk not only corrupting the medium entirely but permanently damaging the flash device.

If all you need to do is write a boot image to the flash device, use the tffsBootImagePut() utility instead of tffsRawio(). This function provides safer access to the flash medium.

IOCTL

This driver responds to all ioctl codes by setting a global error flag. Do not attempt to format a flash drive using ioctl calls.

INCLUDE FILES

- tffsDrv.h
- fatlite.h
Subroutines

flDelayLoop() - consume the specified time .................................................. 70
flFitInSocketWindow() - check whether the flash array fits in a socket window .... 70
mkbootTffs() - make the specified flash device a boot device ....................... 71
tffsBootImagePut() - write to the boot-image region of the flash device ............ 71
tffsDevCreate() - create a TrueFFS block device suitable for use with dosFs.... 73
tffsDevFormat() - format a flash device for use with TrueFFS .................... 73
tffsDrv() - initialize the TrueFFS system ................................................. 74
tffsRawio() - low level I/O access to flash components.............................. 75
tffsShow() - show device information on a specific socket interface ............. 76
tffsShowAll() - show device information on all socket interfaces .................. 77
**flDelayLoop()**

**NAME**

`flDelayLoop()` - consume the specified time

**SYNOPSIS**

```c
void flDelayLoop(
    int cycles /* loop count to be consumed */
)
```

**DESCRIPTION**

This routine consumes the specified time.

**SEE ALSO**

`sysTffs`

---

**flFitInSocketWindow()**

**NAME**

`flFitInSocketWindow()` - check whether the flash array fits in a socket window

**SYNOPSIS**

```c
long int flFitInSocketWindow(
    long int chipSize, /* size of single physical chip in bytes */
    int  interleaving, /* flash chip interleaving (1,2,4 etc) */
    long int windowSize /* socket window size in bytes */
)
```

**DESCRIPTION**

TrueFFS uses this routine to make sure that the specified chip fits in the socket window.

**RETURNS**

A chip size guaranteed to fit in the socket window.

**SEE ALSO**

`sysTffs`
**mkbootTffs()**

**NAME**

`mkbootTffs()` - make the specified flash device a boot device

**SYNOPSIS**

```c
STATUS mkbootTffs
{
    int  drive,  /* drive number: (0 - TFFS_MAX_DRIVES - 1) */
    int  removable,  /* removable or not: (TRUE - FALSE) */
    char *in  /* name of file to read: "bootrom_uncmp" */
}
```

**DESCRIPTION**

This command sets up the flash device, `drive`, as a boot device. The `removable` parameter expects a 1 if the flash device is removable or a 0 if it is not. The `in` parameter specifies the name of the boot file.

**EXAMPLES**

In the following example, the first zero identifies drive zero as the flash device to be made into a boot device. The second zero indicates that the flash is not removable. The `bootrom_uncmp` parameter specifies the name of the boot file.

```c
-> mkbootTffs 0, 0, "bootrom_uncmp"
```

**RETURNS**

OK, or ERROR if there is an error copying from `in` to the disk.

---

**tffsBootImagePut()**

**NAME**

`tffsBootImagePut()` - write to the boot-image region of the flash device

**SYNOPSIS**

```c
STATUS tffsBootImagePut
{
    int   driveNo,  /* TFFS drive number */
    int   offset,  /* offset in the flash chip/card */
    char * filename /* binary format of the bootimage */
}
```

**DESCRIPTION**

This routine writes an input stream to the boot-image region (if any) of a flash memory device. Typically, the input stream contains a boot image, such as the VxWorks boot image, but you are free to use this function to write any data needed. The size of the boot-image region is set by the `tffsDevFormat()` call (or the `sysTffsFormat()` call, a BSP-specific helper function that calls `tffsDevFormat()` internally) that formats the flash device for use with TrueFFS.
If `tffsBootImagePut()` is used to put a VxWorks boot image in flash, you should not use the s-record version of the boot image typically produced by `make`. Instead, you should take the pre-s-record version (usually called `bootrom` instead of `bootrom.hex`), and filter out its loader header information using an `xxxToBin` utility. For example:

```
elfToBin < bootrom > bootrom.bin
```

Use the resulting `bootrom.bin` as input to `tffsBootImagePut()`.

The discussion above assumes that you want only to use the flash device to store a VxWorks image that is retrieved from the flash device and then run out of RAM. However, because it is possible to map many flash devices directly into the target’s memory, it is also possible run the VxWorks image out of flash memory, although there are some restrictions:

1. The flash device must be non-NAND.
2. Only the text segment of the VxWorks image (`vxWorks.res_rom`) may run out of flash memory. The data segment of the image must reside in standard RAM.
3. No part of the flash device may be erased while the VxWorks image is running from flash memory.

Because TrueFFS garbage collection triggers an erase, this last restriction means that you cannot run a VxWorks boot image out of a flash device that must also support a writable file system (although a read-only file system is OK).

This last restriction arises from the way in which flash devices are constructed. The current physical construction of flash memory devices does not allow access to the device while an erase is in progress anywhere on the flash device. As a result, if TrueFFS tries to erase a portion of the flash device, the entire device becomes inaccessible to all other users. If that other user happens to be the VxWorks image looking for its next instruction, the VxWorks image crashes.

**RETURNS**

OK or ERROR

**SEE ALSO**

`tffsConfig`
### tffsDevCreate()

**NAME**  
tffsDevCreate() - create a TrueFFS block device suitable for use with dosFs

**SYNOPSIS**  
```c
BLK_DEV * tffsDevCreate
    (  
        int tffsDriveNo, /* TFFS drive number (0 - DRIVES-1) */
        int removableMediaFlag /* 0 - nonremovable flash media */  
    )
```

**DESCRIPTION**  
This routine creates a TFFS block device on top of a flash device. It takes as arguments a drive number, determined from the order in which the socket components were registered, and a flag integer that indicates whether the medium is removable or not. A zero indicates a non removable medium. A one indicates a removable medium.

It is within this function that a device-appropriate MTD and translation layer module are bound to the socket component driver.

**RETURNS**  
BLK_DEV pointer, or NULL if it failed.

**SEE ALSO**  
tffsDrv usrTffsConfig

---

### tffsDevFormat()

**NAME**  
tffsDevFormat() - format a flash device for use with TrueFFS

**SYNOPSIS**  
```c
STATUS tffsDevFormat
    (  
        int tffsDriveNo, /* TrueFFS drive number (0 - DRIVES-1) */
        int arg /* pointer to tffsDevFormatParams structure */  
    )
```

**DESCRIPTION**  
This routine formats a flash device for use with TrueFFS. It takes two parameters, a drive number and a pointer to a device format structure. This structure describes how the volume should be formatted. The structure is defined in `dosformat.h`. The drive number is assigned in the order that the socket component for the device was registered.

The format process marks each erase unit with an Erase Unit Header (EUH) and creates the physical and virtual Block Allocation Maps (BAM) for the device. The erase units reserved for the “boot-image” are skipped and the first EUH is placed at number (boot-image length - 1). To write to the boot-image region, call `tffsBootImagePut()`.
WARNING
If any of the erase units in the boot-image region contains an erase unit header from a previous format call (this can happen if you reformat a flash device specifying a larger boot region) TrueFFS fails to mount the device. To fix this problem, use tffsRawio() to erase the problem erase units (thus removing the outdated EUH).

The macro TFFS_STD_FORMAT_PARAMS defines the default values used for formatting a flash disk device. If the second argument to this routine is zero, tffsDevFormat() uses these default values.

RETURNS
OK, or ERROR if it failed.

SEE ALSO
tffsDrv

tffsDrv()

NAME
tffsDrv() - initialize the TrueFFS system

SYNOPSIS
STATUS tffsDrv (void)

DESCRIPTION
This routine sets up the structures, the global variables, and the mutual exclusion semaphore needed to manage TrueFFS. This call also registers socket component drivers for the flash devices attached to your target, and, in the process, this function binds an appropriate translation layer module and MTD to the socket component driver.

Because tffsDrv() is the call that initializes the TrueFFS system, this function must be called (exactly once) before calling any other TrueFFS utilities, such as tffsDevFormat() or tffsDevCreatet(). Typically, the call to tffsDrv() is handled for you automatically by the VxWorks start up code if you defined INCLUDE_TFFS or INCLUDE_PCMCIA in your BSP's config.h.

RETURNS
OK, or ERROR if it fails.

SEE ALSO
tffsDrv
tffsRawio()

NAME

*tffsRawio* - low level I/O access to flash components

SYNOPSIS

```c
STATUS tffsRawio
   ( int tffsDriveNo,  /* TrueFFS drive number (0 - DRIVES-1) */
     int functionNo,  /* TrueFFS function code */
     int arg0,  /* argument 0 */
     int arg1,  /* argument 1 */
     int arg2 /* argument 2 */
   )
```

DESCRIPTION

Use the utilities provided by this routine with the utmost care. If you use these routines carelessly, you risk data loss as well as permanent physical damage to the flash device.

This routine is a gateway to a series of utilities (listed below). Functions such as `mkTffsboot()` and `tffsBootImagePut()` use these *tffsRawio()* utilities to write boot sector information. The functions for physical read, write, and erase are made available with the intention that they be used on erase units allocated to the boot-image region by *tffsDevFormat()*(). Using these functions elsewhere could be dangerous.

The *arg0*, *arg1*, and *arg2* parameters to *tffsRawio()* are interpreted differently depending on the function number you specify for *functionNo*. The drive number is determined by the order in which the socket components were registered.

<table>
<thead>
<tr>
<th>functionNo</th>
<th>arg0</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFFS_GET_PHYSICAL_INFO</td>
<td>user buffer</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TFFS_PHYSICAL_READ</td>
<td>address to read</td>
<td>byte count</td>
<td>user buffer address</td>
</tr>
<tr>
<td>TFFS_PHYSICAL_WRITE</td>
<td>address to write</td>
<td>byte count</td>
<td>user buffer address</td>
</tr>
<tr>
<td>TFFS_PHYSICAL_ERASE</td>
<td>first unit</td>
<td>number of units</td>
<td>N/A</td>
</tr>
<tr>
<td>TFFS_ABS_READ</td>
<td>sector number</td>
<td>number of sectors</td>
<td>user buffer address</td>
</tr>
<tr>
<td>TFFS_ABS_WRITE</td>
<td>sector number</td>
<td>number of sectors</td>
<td>user buffer address</td>
</tr>
<tr>
<td>TFFS_ABS_DELETE</td>
<td>sector number</td>
<td>number of sectors</td>
<td>N/A</td>
</tr>
<tr>
<td>TFFS_DEFragment_VOLUME</td>
<td>number of sectors</td>
<td>user buffer address</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*TFFS_GET_PHYSICAL_INFO* writes the flash type, erasable block size, and media size to the user buffer specified in *arg0*.

*TFFS_PHYSICAL_READ* reads *arg1* bytes from *arg0* and writes them to the buffer specified by *arg2*.

*TFFS_PHYSICAL_WRITE* copies *arg1* bytes from the *arg2* buffer and writes them to the flash memory location specified by *arg0*. 

75
TFFS_PHYSICAL_ERASE erases arg1 erase units, starting at the erase unit specified in arg0.

TFFS_ABS_READ reads arg1 sectors, starting at sector arg0, and writes them to the user buffer specified in arg2.

TFFS_ABS_WRITE takes data from the arg2 user buffer and writes arg1 sectors of it to the flash location starting at sector arg0.

TFFS_ABS>Delete deletes arg1 sectors of data starting at sector arg0.

TFFS_DEFRAGMENT_VOLUME calls the defragmentation routine with the minimum number of sectors to be reclaimed, arg0, and writes the actual number reclaimed in the user buffer by arg1. Calling this function through some low priority task will make writes more deterministic. No validation is done of the user specified address fields, so the functions assume they are writable. If the address is invalid, you could see bus errors or segmentation faults.

RETURNS OK, or ERROR if it failed.

SEE ALSO tffsDrv

tffsShow()

NAME tffsShow() - show device information on a specific socket interface

SYNOPSIS void tffsShow

   (             
       int  driveNo /* TFFS drive number */          
    )

description This routine prints device information on the specified socket interface. This information is particularly useful when trying to determine the number of Erase Units required to contain a boot image. The field called unitSize reports the size of an Erase Unit.

If the process of getting physical information fails, an error code is printed. The error codes can be found in flbase.h.

SEE ALSO tffsConfig
# tffsShowAll()

**NAME**

tffsShowAll() - show device information on all socket interfaces

**SYNOPSIS**

```c
void tffsShowAll (void)
```

**DESCRIPTION**

This routine prints device information on all socket interfaces.

**SEE ALSO**

tffsConfig