

MICRO-MONITOR

An Embedded System Boot Platform

Application Notes

Revision date: November 15, 2002

MicroMonitor Application Notes

Table of Contents

This document covers miscellaneous topics involving the monitor and its use.

SYSTEM STARTUP	4
THE STARTUP SEQUENCE	4
STAGE 1: THE MONRC FILE	4
STAGE 2: A TOP LEVEL STARTUP SCRIPT	5
IF MONRC DOESN'T EXIST.....	7
FACTORY INSTALLED MAC ADDRESS:	7
STARTUP EXAMPLE:.....	9
MEMORY LAYOUT OPTIONS FOR TFS	11
TFS: TINY FILE SYSTEM	14
YET ANOTHER FLASH FILE SYSTEM... WHY?	14
FILE ATTRIBUTES	14
AUTOBOOTABLE FILES	14
WHAT IS AN "IN-PLACE-MODIFIABLE" FILE?.....	15
USER LEVELS	15
FILE DECOMPRESSION.....	15
TIME-OF-DAY.....	16
MULTIPLE STORAGE DEVICES.....	17
FLASH SPACE OVERHEAD REQUIRED BY TFS	17
TFS IN RAM.....	18
FLASH LIFE-EXPECTANCY	18
FIELD UPGRADE OPTIONS	20
UPGRADE OF A MULTI-FILE APPLICATION:	20
REMOTE UPDATE OF MONITOR BINARY:	20
DEBUGGING YOUR APPLICATION	22
SYMBOLIC DEBUGGING	22
RUN-TIME TRACE	23
CONTEXT-SENSITIVE STACK TRACE	23
BUILDING A "SYMTBL" FILE:.....	24
PROFILING YOUR APPLICATION	24
TASK ID STATISTICS.....	24
FUNCTION STATISTICS	25
PC STATISTICS.....	25
EXAMPLE PROFILING SESSION.....	26
PORTING TO A NEW TARGET	28
FIRST THINGS FIRST:	28
DIRECTORY STRUCTURE:	29
RUNTIME EXECUTION, FLASH OR RAM?	29

THE CONFIG.H FILE:	30
THE FLASH DRIVER:	31
CONFIGURING TFS.....	34
MISCELLANEOUS ENTRIES IN CONFIG.H.	35
A FEW GENERAL SUGGESTIONS.....	36
BOOTING LINUX (OR ANY "PROCESS-MODEL OS") THROUGH MICROMONITOR	37

System Startup

THE STARTUP SEQUENCE

There can potentially be multiple stages of startup in a system that uses MicroMonitor as its base platform. The initial stage is for the monitor itself and all subsequent stages are established by the user based on the content of auto-bootable files found in TFS. This page discusses the initial stage and an example stage 2.

STAGE 1: THE MONRC FILE

In this stage, the system is taken from a hard reset state to a point where it is able to communicate with a host to retrieve system configuration information or it may just startup in a standalone mode with all the configuration parameters it needs on-board. After a hard-reset, the monitor firmware runs the lowest levels of initialization, and prior to starting its peripherals, it attempts to execute a script file called "monrc". This file, like any other run-control file, is used to allow the monitor to establish some of its basic configuration at startup. It allows the same monitor program to start up in different ways based on the content of the file. The monrc file is automatically executed by the monitor in the early stages of system startup. It is the only executable file in TFS that is automatically executed at startup despite the fact that it is not flagged as an auto-bootable file. Typically, the only thing that is in this file is a few set commands. They are used to establish things like the IP and MAC addresses (shell variable names: IPADD and ETHERADD respectively); but other things (like a sleep -l, setting pio lines, etc) can be put in the monrc file so that the board boots up with its own personality without the need to recompile a unique monitor per system. It is important to be aware of the fact that when the monitor executes the monrc file it is still in the process of initializing itself. The monitor has not yet turned up its own ethernet interface (the ethernet driver will use the content of shell variables ETHERADD and IPADD to configure itself), so the monrc file should contain only the most basic startup sequences as mentioned above, anything beyond this basic stuff should be done with an auto-bootable script, outside the context of monrc execution.

The value loaded into the IPADD shell variable is typically going to be some IP address, but the monitor also supports automatic boot-up using BOOTP or DHCP (automatic allocation mode only). If the content of IPADD is the string "DHCP", then the monitor will broadcast a DHCP_DISCOVER request. If the content of the IPADD shell variable is the string "BOOTP", then the monitor will initiate a BOOTP broadcast. The monitor can be built to run a DHCP client in several different ways. With either DHCP or BOOTP, the result of the handshake is that some shell variables are loaded and possibly a file has been downloaded and executed. Note that for both the "DHCP" and "BOOTP" string loaded into the IPADD shell variable, if the character 'v' or 'V' is appended to the string (ie.. DHCPv, DHCPV, BOOTPv or BOOTPV) a low (v) or high (V) level of verbosity is enabled during the DHCP/BOOTP handshake. One final note regarding the IPADD shell variable... if it is loaded with "0.0.0.0", this tells the monitor to not startup the ethernet driver at all. It can then be restarted later in a script by re-assigning the IPADD shell variable and issuing the "ether on" and (optionally) the "dhcp" commands.

During development, we usually don't need (or want) the BOOTP/DHCP server to complicate things, so the monrc file would be loaded with a real IP address (along with all of the other environment variables that are typically established by DHCP or BOOTP) and the board doesn't use DHCP. *Note that this is all configurable based on the content of this small ASCII file resident on the target, there is no recompilation of the monitor or application to acquire the per-board uniqueness.* Following are a few example monrc files with shell variable settings that apply to specific applications...

Monrc file example #1

This example establishes the information needed by the Packet Phone Adapter for configuration without the requirement of a DHCP server (note that the IPADD setting is an IP address, not the DHCP string).

```
set ETHERADD 00:60:1D:02:09:9B
set IPADD 135.3.131.87
set GIPADD 135.3.130.1
set NETMASK 255.255.254.0
set PROXY_IP 208.3.248.167
set LINE0_NUMBER 2072221113
set CODER PCM
set CODERCTL 0000
set MIN_NET_DELAY 30
set MAX_NET_DELAY 150
```

Monrc file example #2

This example establishes the basic bootup information needed by the target so that it retrieves its configuration information from the DHCP server. Note that the DHCP client code that comes with the monitor is in two parts: generic code and application specific hooks. For the PPA, there are hooks added to support the retrieval of several different vendor-specific options unique to the PPA application. The mechanism used by the PPA, although specific to that application, is very re-usable for other sets of vendor-specific options, and this code is available for reference. The current DHCP client code is set up so that in most cases there is no need to modify the monitor source code; however, if source code modification is necessary, then hooks have been installed into the common code (dhcpboot.c) that allow the user to add target-specific extensions (dhcp_00.c) without touching the common source code.

```
set ETHERADD 00:60:1D:02:09:9B
set IPADD DHCP
```

STAGE 2: A TOP LEVEL STARTUP SCRIPT

Once the monitor has run the monrc file, it will then complete its own initialization based on the presence (or lack of) shell variables. The monitor will then run through the files in TFS looking for any other auto-bootable, executable file. In the case of the PPA, the only auto-bootable file will be the PPAGo script. Using the scripting capabilities of the monitor, the PPAGo file is the top-level application file resident on the PPA. The actual PPA application code is not even auto-bootable; rather, it is started up through this auto-bootable PPAGo script. The script is responsible for completion of the DHCP handshake, getting the current ICMP timestamp, determining if the current application resident on the PPA is up to date; and if not, kicking off a TFTP transfer to get the latest one.

```
# Packet Phone Adapter (PPA) Startup Script
# Poll the IPADD shell variable until DHCP has completed.
# If after 8 seconds, DHCP has not completed, then reset.

set go
set retry 0
set APPNAME ppaapp.str

# DO_DHCP:
sleep 1
if $retry gt 8 goto DHCP_FAIL
```

```

set -i retry
if $IPADD seq DHCP goto DO_DHCP
if $IPADD seq DHCPV goto DO_DHCP
if $IPADD seq DHCPv goto DO_DHCP
# Uncomment these next 2 lines for ICMP Timestamp...
#set TIMESTAMP
#icmp -d -5 -fd -v TIMESTAMP time $PROXY_IP
goto DO_TFTP

# DHCP_FAIL:
echo DHCP Failed, trying again...
# RESET:
reset

# DO_TFTP:
# If the APPINFO variable is set, then assume the
# DHCP server is supplying enough information for us to
# decide whether or not we want to update the current
# application...
if $APPINFO seq \$APPINFO goto DO_BOOT
TFS info $APPNAME INFO
if $INFO seq $APPINFO goto DO_BOOT
TFS rm $APPNAME
TFS rm errlog
TFS clean
TFTP -feA -F $APPNAME -i $APPINFO $TFTPSRVR get $BOOTFILE 0x10000
TFS info $APPNAME INFO
if $INFO sne \$INFO goto DO_BOOT
echo TFTP Failed, trying again...
reset

# DO_BOOT:
# Setup the BOOT LINE used by VxWorks:
fm 0x10000 0x10100 0
pm -s 0x10000 "mbc(0,0) e=" $IPADD

# Query user to kick off the application...
# If query times out, then start ppa...
echo Start PPA with IP $IPADD ?\c
read -t1000 go
if $go sne \$go goto END

# Start application:
# If application does not exist, restart.
set go
set retry
TFS info $APPNAME INFO
if $INFO seq \$INFO goto RESET
echo
echo Turning control over to $APPNAME ...
ether off
$APPNAME # (This will not return.)

```

END

IF MONRC DOESN'T EXIST...

As discussed above, the monitor will look to the monrc file for boot-up parameters. If the monitor boots up and does not find a monrc file, it will make one additional attempt at it by looking for "monrc.bak". This is done to provide protection from the cases where a system is updating its monrc file... While updating it, copy the old version to monrc.bak, then if somehow the target crashes in the middle of the update, the monitor will still have the monrc.bak file to boot from.

What happens if both monrc and monrc.bak do not exist? The flash used by the file system is not necessarily protected; hence, there is the possibility of the monitor booting up and TFS being empty or the monrc file just not being there. Assuming the target has an ethernet interface, the two most important parameters that the target must get from somewhere are the MAC address and IP address. For the MAC address, there will probably be some non-volatile storage device (typically serial EEPROM) that will be loaded with the MAC address, so assuming the hardware is functional, even if the monrc file exists, in a final product, the MAC address is retrieved from some peripheral, not the monrc file (see MAC storage discussion below). If however, there is no MAC address stored, then the last-resort MAC address can be a hard-coded value that is either a MAC address or a string indicating that the target is to startup with ethernet disabled (see DEFAULT_ETHERADD in config.h). For the IP address, the monitor can be configured to run with a hard-coded (see DEFAULT_IPADD in config.h) IP address, BOOTP or DHCP. Following is a discussion of each of these modes...

Fixed IP Address Startup

The monitor can be configured to simply boot up with a fixed IP address as a last resort. If this is the case, then there is no server interaction at boot up, the monitor simply starts up with the default MAC and IP addresses. The DEFAULT_IPADD macro in config.h must be set to the appropriate IP address as a string.

Ethernet Disabled at Startup

The monitor can be configured to simply boot up the ethernet interface disabled. The bootup will post a warning at the console and all ethernet activity is disabled. This is done by setting the DEFAULT_ETHERADD macro in config.h to "ETHER_OFF".

BOOTP/DHCP Startup

In this case, the monitor has been built with the DEFAULT_IPADD macro loaded with "BOOTP" or "DHCP" and will automatically broadcast the appropriate request on the subnet it is connected. For a detailed discussion of this startup sequence, refer to the dhcp command reference.

FACTORY INSTALLED MAC ADDRESS:

The convenient (during development) mechanism used to assign MAC addresses to each target is through the monrc file using the ETHERADD shell variable. This is adequate during field trials and application development but it is impractical when a large number of targets are deployed. As mentioned above, the hardware will typically have some non-volatile storage device that allows the MAC address to be stored away. Following is a discussion on some different techniques that have been considered for a factory-installed MAC (and possibly IP) address.

DALLAS Semiconductor UNIQUEWARE (DS2502/5/6)

Dallas Semiconductor makes a device that can be programmed in the factory (by Dallas) for an additional charge of about \$.25 per part. There are a few different versions of the part that have varying amounts of programmable space available. According to Dallas technical support, a small quantity of parts

can be ordered to verify that the data is what you want, then a larger quantity of parts can be ordered (note that large quantity does not necessarily mean tens of thousands, although it could). The part would then contain a unique 48-bit ID number followed by the data that you tell Dallas to put in the parts programmable memory. It will be read-only; so, unless the device is physically removed from the circuit board, it provides each board with a programmed MAC address and a guaranteed unique ID number that can be used as its tracking number for maintenance records. Refer to Application Note 99 at Dallas Semiconductor's web-site for more details.

PROs:

- The data will stay with the hardware, so each circuit pack is guaranteed to have some unique ID and MAC address.
- The part is small (3-pin package).
- Allows a factory run of targets to be built with basically no concern for any uniqueness per board, the installation of this part makes it happen automatically.

CONs:

- The part has limited storage space.

Serial EEPROM

Several vendors have various kinds of serial EEPROM ranging in storage size from bytes to Kbytes. A device like this could be used and programmed by the factory to contain MAC address and other project-specific information. Access to the device is not directly through a memory address, so it is unlikely that the device would ever be "accidentally" modified as a result of some code bug. There are probably arrangements (similar to those provided by Dallas) that can be made with the manufacturers of these parts as well.

MAC (and optionally IP) in read-only section of flash

The monitor is set up to be provisioned with the MAC and IP address stored in locations of the boot flash that are conveniently accessible by factory personell that are programming the devices. This means that each flash could be loaded with a unique MAC and/or IP address and the monitor will deal with that. The sectors used by the monitor could be made read-only when programmed, so this would provide a mechanism that does not require any additional hardware, but each flash device would have to be uniquely programmed and the monitor would not be remotely updateable.

PROs:

- No additional parts involved.
- Monitor already supports this.

CONs:

- If the monitor is made read-only so that the MAC cannot be overwritten, then the monitor also cannot be modified. An exception to this is that the flash boot sector could contain enough code to jump out of the read-only sector to the remainder of the monitor that would not be read-only.
- Each FLASH device would have to be uniquely programmed.

First sector(s) of TFS made read-only

The initial sector of flash allocated to TFS could be made read-only after programming of certain configuration files has been done. This supports a variety of site-dependent data that can be stored away as a file in TFS and accessible by the application (read-only).

If some mechanism of making the flash read-only is used, then optionally, some flash devices provide a mechanism that allows a sector to be temporarily un-protected. This means that a jumper could

be installed into the hardware to allow the read-only sectors to be writeable if necessary. In the case of the serial EEPROM, a layer of software protection can be added if access to the EEPROM is only made through the command line of the monitor... One command can be used to open a protection window, and a second command can be used to actually make the EEPROM modifications only if the protection window is opened. This eliminates any one leg of code from being accidentally called to modify EEPROM content.

The point of this whole discussion is that the monc file can provide a convenient means of storage for the MAC address during development, but it is not a practical means for product deployment (unless it is made read-only), so plan ahead!

STARTUP EXAMPLE:

This section discusses another example of a bootup scenerio that doesn't use BOOTP or DHCP but still has the ability to intelligently re-download its application and even a new monitor if necessary. The first script, called Go, is an autobootable script that simply retrieves a second script (called SysStart) and executes it. The second script is where the smarts are. Note that this example assumes that the final application (called SysApp) is simply an executable script (flags = 'e'). For a real application, the flags are likely to be different.

The idea here is that all targets are loaded with the "Go" script, then as the application and/or monitor changes, the content of the "SysStart" script is modified on the TFTP server to cause a new download of the appropriate file (monitor and/or application). Note that this does cause a TFTP file transfer to be done on every reset; but, it does not cause any excessive wear on the flash because TFS is smart enough to NOT touch the flash if the file being added is identical to what is already there.

The "Go" script:

```
#
# Go:
# Startup script for system...
# Retrieve the SysStart script from the TFTP server and run it.
# The logic in that script will do the rest. Note that if the
# shell variable BOOTSRVR is populated (possibly from earlier
# BOOTP or DHCP transaction) then use it; else load a default.
#
if $BOOTSRVR sne \ $BOOTSRVR goto GETFILE
set BOOTSRVR 135.3.94.76

# GETFILE:
tftp -F SysStart -fe $BOOTSRVR get SysStart

SysStart
```

The "SysStart" script:

```
#
# SysStart:
# This script is part of a 2-phase bootup in which the system
# retrieves this file from a tftp server and uses the logic in
# this script to determine if the monitor and/or the application
# needs to be downloaded. The content of the following three
# shell variables must be adjusted on the server to reflect the
# content of the monitor.bin and SysApp files also on the server...
#
set INFO 10/14/99 # Date of application build on server
```

```

set MONBUILD 10/13/99@09:56:38 # Date of monitor build on server
set MONSIZE 129356             # Size of monitor binary on server

# If the current build date of the monitor ($MONITORBUILT) does
# not match the build date of the monitor binary that is available
# on the server ($MONBUILD), then download a new monitor...
# Note1: the "flash ewrite" command executes out of RAM, so it is legal to
# modify the boot flash with this command.
# Note2: after the "ewrite" command executes, a new monitor
# is installed and the target resets. The next pass through this
# script (as a result of the reset) would miss this step
# because the content of MONBUILD will match MONITORBUILT.
#
if $MONBUILD seq $MONITORBUILT goto GOTMONITOR
tftp $BOOTSRVR get monitor.bin
if $TFTPGET ne $MONSIZE goto GOTMONITOR
echo Updating monitor, size = $TFTPGET...
flash opw
flash ewrite $BOOTROMBASE $APPRAMBASE $TFTPGET

# GOTMONITOR:
# If the info field of the current application does not match
# the info field of the application available on the server
# ($INFO), then download a new application...
#
tfs info SysApp APPINFO
if $APPINFO seq $INFO goto GO_APP
tftp -f e -F SysApp -i $INFO $BOOTSRVR get SysApp

# GO_APP:
# Kick off the application...
SysApp

# Should not get here!
reset

```

Memory Layout Options for TFS

The monitor itself is a program. As a program, it doesn't require much of the system that it runs on... An RS-232 port, maybe an ethernet port, some RAM and some flash. That's about it, and if left at this point, chances are likely that the hardware will support the needs of the monitor. On the other hand, the application that the monitor will eventually be turning control over to may have requirements that justify a little bit of up-front thinking with regard to the memory layout. There are a few things to consider when configuring a memory design for use with a system that will run with MicroMonitor. A wide range of flash devices are now available. For the sake of this discussion, there are a few major distinctions between the various devices available: the size of the device and the flexibility it provides with regard to sector protection...

- The size of the device is pretty obvious. You must be sure the application you plan to run, plus the monitor, will fit in the device.
- The sector protection flexibility is a bit less obvious but must be considered if there is any need for some portion of the device to be unprogrammable while other portions are programmable. There are a lot of options here. Some devices can be programmed with an external device programmer and then certain sectors can be made uneraseable. Once put into the system those uneraseable sectors are safe from any "devious" firmware that may accidentally attempt to corrupt that space. This is good. On the other hand, if those sectors ever need to be reprogrammed, then the device will have to be removed from the target system and reprogrammed by the external device programmer. Some of the newer parts support "temporary sector un-protect" and there are varying levels of support for this. Some of the parts go into an "unprotect" mode if some higher voltage is applied to a pin, while others simply need to have one of their pins set to a high or low state (via some physical jumper). Lots of choices here, so think about what you need before you just assume "flash is flash".

Following is a list of considerations that start with a system that needs minimal memory support, up through a system that should consider additional hardware support for TFS. Each question is followed by a brief discussion and a device suggestion in italics.

Will the system ever be updated after initial factory installation?

If the answer to this question is no, then you have the simplest of systems. The monitor and application files can be burned into the memory and as long as the code doesn't try to modify anything, everything will work fine. Since this system does not require any in-system changes, it would be adequate to use EPROM, but EPROM is quickly becoming "old" technology. Careful consideration should be taken if EPROM is being considered. Much of the flexibility of the monitor is lost without the ability to write to the file system.

Is it ok for the monitor binary to be in flash space that is erasable?

If the answer to this question is yes, then the monitor and application files can all exist in the same flash device and there is no need to consider any hardware-assisted flash protection, just leave all the flash in its normal write/erase mode. If the answer to this is no, then either the sectors of the flash device that contain the monitor must be protected, or the boot monitor must be in a different device than TFS. This system would be adequately served by an AMD 29F010-type device. It's simple, and is probably at the lower end of the price range for flash. Note that if the monitor binary is in space that is not erasable, then it cannot be updated in the field.

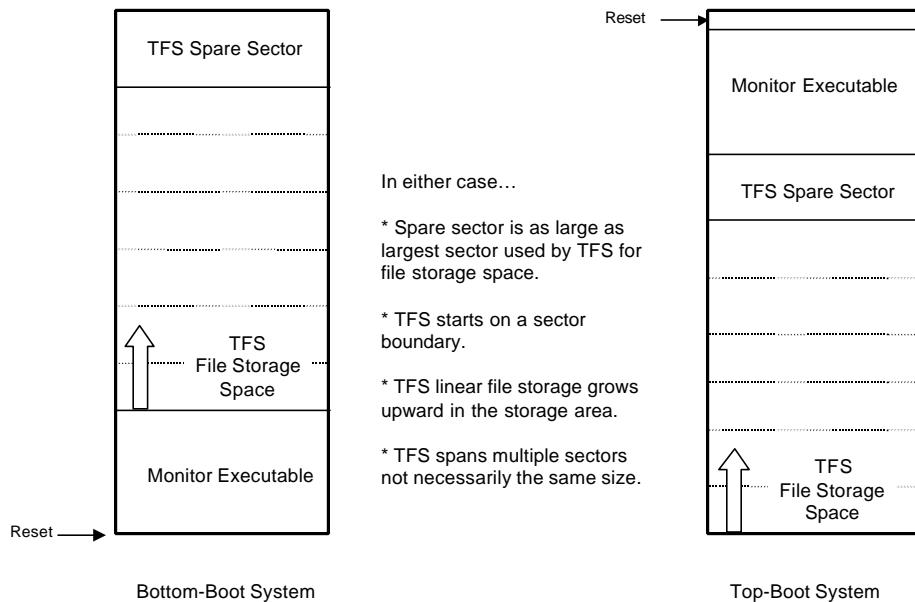
Is there a need for some files in TFS to be un-erasable and others to be erasable?

If the answer to this is yes, then the hardware must be able to selectively protect (or not) multiple sectors of flash space. Note that this does not necessarily mean any additional hardware, it can be addressed by selecting a more versatile flash device. *The Intel TE28F320 device (family) is very versatile for this kind of stuff.*

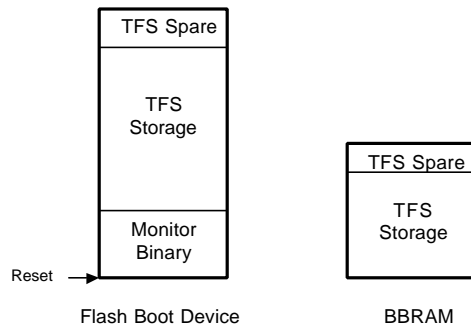
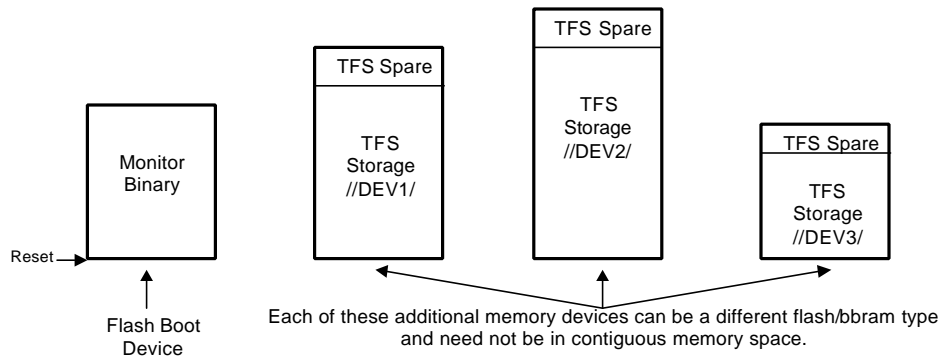
Is there a need for some files to be very frequently written and rewritten?

If the answer to this is yes, then another question must be asked: **Do these frequently modified files need to be in memory that will survive a power hit?** If no, then a portion of TFS can be configured to overlay standard RAM or DRAM space (and it will be cleared after a reset or power hit). If yes, then the best solution is to add some battery backed ram to the system and overlay TFS onto that memory range. Note that we are talking about tens-of-thousands of rewrites here, not just a few. Dallas Semiconductor makes BBRAM parts that have the battery built into the device and tout a 10-year life. They also have devices that allow you to turn a normal SRAM into a battery-backed RAM.

Once the answers to these questions are known, then it's necessary to decide what hardware support to add and if not added, what adjustments need to be made to the requirements. For the bottom boot system, the monitor would reside just above the reset vector address of the CPU. For the top boot system, the monitor would reside just below the reset vector address of the CPU. Following are a few diagrams that should help...



Single-device flash layout options with TFS



Multiple-device flash layout options with TFS

Note that there is additional discussion on flash in the **Error! Reference source not found.** command and the boot block safety appnote.

TFS: Tiny File System

Yet another Flash File System... Why?

There are flash file systems (FFS) available as extensions to most of the RTOSes out there. Each has its own set of capabilities and each one has to be hooked into the RTOS in some non-trivial way. Plus, an FFS typically costs an additional fee to use. Then, if you use a different OS, or possibly none at all, you have to determine whether or not the FFS you are using is compatible with the new environment; and, even if it is, you have to re-do the "hook-up" of the FFS to the RTOS.

The model for TFS (and the servers that use TFS) is to keep things simple, but still provide the embedded system with a wide range of facilities. TFS is a linear file system that gives a typical embedded system project all of the "file-system-like" capabilities it will ever need (most of which would not come with an off-the-shelf FFS)... A complete programming API that looks very similar to the standard open/close/read/write/etc... system calls that you would use on a Unix platform. Also a command line interface that allows a user to list, display, add, delete, load and edit files. Its loader supports executables as COFF, ELF, AOUT or ascii scripts. File decompression, user levels, a file change-log and automatic file system cleanup (defragmentation) are also supported with TFS. Its implementation is independent of the OS used (doesn't even need one) and it is easily hooked into the application.

Also, typical application-level servers that interface to a file system are already written and available. HTTP, FTP and TFTP servers can easily be incorporated into any RTOS/TCP-IP package that conforms to the standard sockets API. Note that these servers are not full-blown servers (no, the HTTP server is not Apache compatible!); but they do support the majority of the typical needs of an embedded system.

File Attributes

TFS supports file attributes. The attribute simply describes the file to TFS, so that when TFS does some automatic stuff, it knows how to do it. An attribute in the file header is simply a bit setting. At the command line, each attribute is assigned a letter which is used to display or create (in a non-verbose mode) the files in TFS. Following is a list of all the file attributes, including a brief description of each...

Attribute	Abbreviation	Description
executable script	e	executable, not simply data
auto-boot	b	file is to be run at boot time
auto-boot with query	B	file is to be run at boot time, after querying user
executable binary	E	COFF, ELF, AOUT, etc...
compressed	c	file is compressed
in-place-modifiable	l	file is in-place-modifiable
unreadable	u	file not readable when monitor is below required user level
user_level	0-3	4 different user levels

Autobootable Files

At some point in the startup of the monitor, it looks to TFS for auto-bootable files. There are actually three different types of autobootable files: two types established by the attributes assigned to the file and one special case (the monrc file). For the monrc file to automatically run, it must exist and be marked as executable. It will be run prior to any other autobootable file in TFS. Actually, it is run prior to the monitor having completed its own initialization (this is done so that the execution of monrc can be used to configure the monitor as it starts up). The remaining autobootable files are run after the monitor has

completed initialization. They will be run in alphabetical order, so the order in which they are placed in TFS doesn't matter. The order in which they are listed via "tfs ls" is the order in which they will be executed...

Name	Size	Location	Flags
bkgd.jpg	5976	0x80047a6c	
cardtilt.gif	6099	0x80040c1c	
boot_diag	109358	0x8004921c	BeE
construction.gif	20222	0x8004243c	
form.html	466	0x8004004c	
ias_app	792080	0x80155a7c	BeE
index.html	442	0x8004026c	
info1.html	1053	0x8004047c	
info2.html	734	0x800408ec	
lucentlogo.gif	1680	0x8004738c	
monrc	823	0x80154d2c	e

For the above list (output of "tfs ls"), the order of autoboot execution would be: 1. monrc; 2. boot_diag; 3. ias_app. All other files listed are simply data files used by the application.

The two autobootable attributes supported are "B" and "b", both of which will run at startup; but the "B" type will query the user at the console port, providing an opportunity to abort the autoboot of that file. Both scripts and binary executables can be configured with autoboot flags enabled; but there are limitations. Consider the list of autobootable files to be a script that contains those executables, each of which will be run in the order shown by "tfs ls", then refer to the section on script nesting for further discussion.

What is an "in-place-modifiable" file?

Typically, when a file is modified, the original file is marked as deleted, and the new version of the file is appended to the end of the list of files currently stored in TFS. This can involve a relatively large amount of overhead if the modification to be made is trivial. As an alternative, a file can be created as an "in-place-modifiable" file which means that the API provides a means by which a file can be modified without the typical deletion/re-creation step mentioned above. This is done by creating the file as in-place-modifiable and specifying the file to be of some size. The space is then allocated in TFS for this file, but the flash is all left in a writable state. This usually means that the bytes in the flash are all 0xff (usually, bits in flash can be cleared on a byte-by-byte basis, but to reset them, an entire sector must be passed). All subsequent writes to this file, then, are done directly to the currently allocated flash instead of to a new block of flash. Obviously this puts some responsibility back on the programmer, but it can potentially save quite a bit of overhead if necessary. When a file is created as in-place-modifiable, then the TFS API function tfsipmod() should be used instead of the standard open-modify-close model.

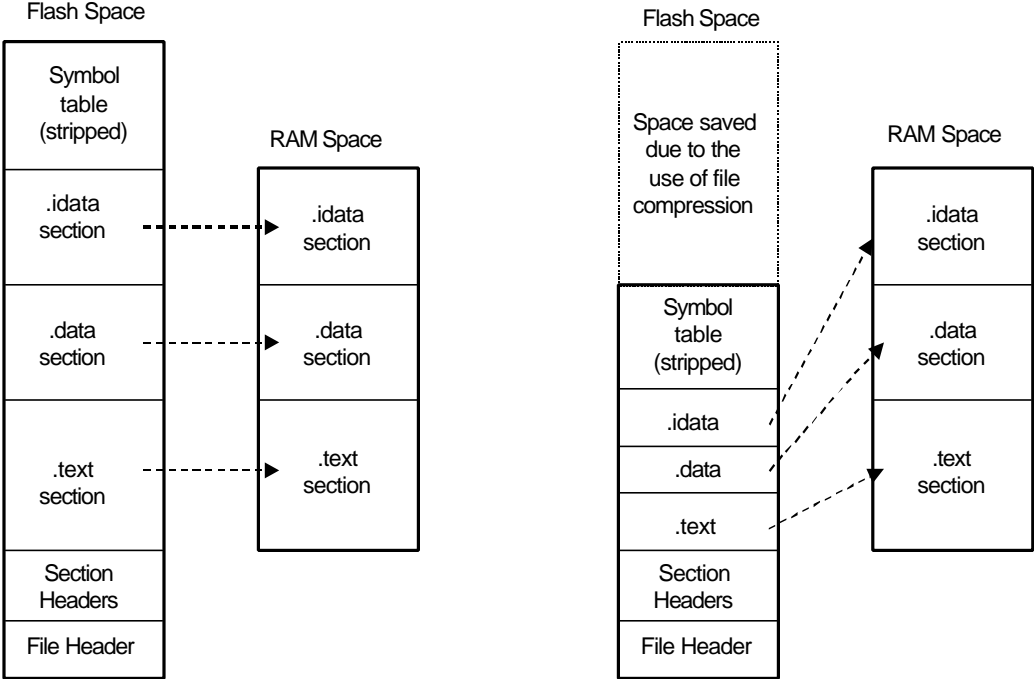
User Levels

The monitor supports the concept of user levels. At any given time, there is an active user level. TFS supports the ability to store a file at a particular user level, then limit access of that file based on the user level at the time of the access. The access can be limited to read-only or not-even-readable. Refer to the discussion on user levels for complete details, but in short, this means that certain files (and executables) can be configured to be accessible only at certain user levels. Since each user level is attainable only via password, a system can be built at user level 3, then lowered to user level 2,1, or 0 and provide a certain degree of protection from unauthorized access.

File Decompression

A typical application file will be either COFF, ELF or AOUT. Each of these file formats has multiple sections of text and data that must be transferred out of the flash space the file resides in, and into some

RAM/DRAM space that the application was built to run in. Decompression of these file types is done by decompressing each of the sections within the file from flash, directly into the RAM/DRAM space the section is destined for. This "section-at-a-time" decompression eliminates the need to decompress the entire file into some block of memory, then load from that block into the actual memory space the image was built for.



Transfer of one file from TFS to DRAM without file compression.

Transfer of one file from TFS to DRAM with file compression.

Generic Comparison of File With and File Without Compression
 (actual amount of compression depends on data being compressed)

Time-of-Day

Since the basic model of the monitor is to run without the need of any interrupts from the host processor, how is it that TFS can keep track of time of day? Actually, it doesn't. It depends on the application code to provide it with two functions that will support this: getLtime() and getAtime(). The first one, (long)getLtime(void), must return a long that is stored in the header of the TFS file when it is created. The second one, (char *)getAtime((long *)tval, (char *)buf, (int)buflen) can be used to simply return an ascii string representing the current time (if tval is 0) or it can return an ascii string representing the value stored in tval. The value in tval will typically be the value that was previously returned from getLtime(). With this interface, TFS really doesn't have a clue about time-of-day, but it uses the capabilities given to it by the application to make it look like it does.

Note that this is a feature that is used by TFS to populate an entry in the header of the file being written at the time. If the two above functions are not supplied to TFS, then the header entry is left blank, and the file simply has no recollection of its time of creation.

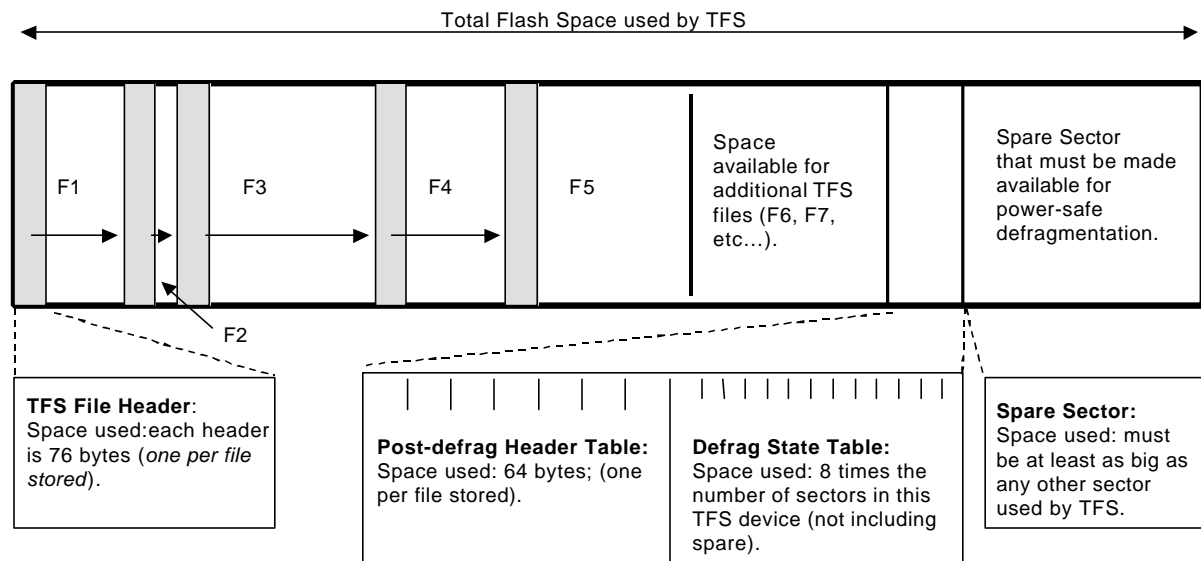
Multiple Storage Devices

In some hardware designs there may be more than one device that could be used for file storage. TFS supports this. A default system will have a boot monitor in the base of the flash, then all remaining flash in that device is used by TFS and that's it. A more complicated system may contain battery backed ram, a boot flash device and a secondary storage flash device, etc... TFS supports multiple devices that are not necessarily in contiguous address space. Each device appears to the user as a directory, so any file can be stored in any device (limited by the size of the device, of course), but a file cannot span across multiple non-contiguous devices. For each device, the same power-safe defragmentation method is used; hence, if battery-backed RAM was on-board, it could be used to eliminate the problem of flash-life expectancy (see below) if there is a need to modify files at a high frequency.

To "steer" a file to a particular device, each device has a unique prefix that, when made part of the file name, tells TFS that the file is destined for that device. If the prefix is omitted from the filename, then the default device is used for storage. Similarly, the file-system maintenance commands (tfs check, tfs clean, tfs freemem, etc...) can also be pointed to a particular device by specifying the device prefix.

Flash Space Overhead Required by TFS

Overlaying TFS onto a FLASH device is not free. There is a certain amount of space that is required by TFS; some of that space is fixed; other space is based on the number of files stored. The following diagram illustrates this...



The amount of space that is actually available for file data storage depends on the number of files stored...

$$\text{Overhead} = (\text{FTOT} * (76+64)) + \text{SPARESIZE} + (\text{SECTORCOUNT} * 8) + 16$$

where...

FTOT is # of files to be stored

Size of TFS header = 76 bytes, size of defragmentation header = 64 bytes, TFS name size = 23 bytes.

SPARESIZE is the size of the spare sector

SECTORCOUNT is the number of sectors allocated to TFS (*not including spare*)

TFS Flash Space Overhead (per device)

(note that header sizes may change as new versions of TFS become available)

Note1: a file that is marked as deleted in TFS requires less overhead than a file that is "living". This is because when a file is dead, there is no need to allocate a defragmentation header to that file. This means that removal of a file (even though it is not actually erasing the flash) still frees up some memory space for new file storage.

Note2: the spare sector must be at least as large as any other sector allocated to a particular flash device within TFS; also, the sector that is just before the spare sector must be large enough to contain all of the defrag header tables. Files can never be stored in the spare sector. Files can be stored in the sector that will contain the defrag overhead, but TFS will not allow the file storage in that sector to exceed the space that is needed for defragmentation overhead; hence, ideally that sector should be as large as the spare..

Note3: when a file is copied into TFS flash space, TFS first looks to see if this file with the same name already exists. If it does, and the data in the new file is identical to that of the current file, then there is no action taken. If on the other hand, the new file is different, then the old file is marked "stale" while the new one is being transferred into flash. This "install-the-new then remove-the-old" mechanism is a protection against the case of the new file transfer process being interrupted. If during the transfer of the new file something causes the target to restart, then TFS can recover by clearing out the partially transferred "new" file and re-installing the stale one. The user must be aware of this because it requires additional flash overhead... There must be space for the old and the new in one device. Note that the user can remove the old one, defragment and then install the new one to avoid this overhead, but then the protection described above is lost.

TFS In RAM

In some cases it is very practical to consider having file storage in RAM space. For non-volatile file storage, this implies battery-backed RAM (or some other power-safe type), and this is supported. Taking it one step further, TFS also supports the ability to have a block of volatile RAM space designated as TFS storage space. This has the disadvantage of the content being lost on a power up (similar to /tmp on UNIX) but has the advantage of unlimited write-erase cycles. Refer to the TFS-In-RAM appnote for details on how to configure this.

Flash Life-Expectancy

It is important to be aware of the fact that the underlying technology (flash) has a limited number of erase cycles. Current flash devices typically support 100,000 to 1,000,000 erases per sector. Applications will use TFS in different ways, so it is impossible to draw any conclusions here with regard to how long the flash will last in a system using TFS. This section will; however, discuss the way TFS uses the flash so that the user can determine flash life based on their application's intended use of the file system.

First of all, be aware that TFS makes no attempt to do "wear-leveling". If the file system you need is to be used heavily enough to require wear-leveling, then buy one! Wear leveling adds a great deal of complexity to the underlying implementation and is beyond the scope of TFS. In every embedded system project I've been on, the need for clean interface to flash by name (instead of by address) has been there. On the other hand, the frequency of file modification has never deemed it necessary to even consider wear-leveling; especially with the trend by most flash manufacturers to support 1,000,000 erases per sector.

As mentioned above, the TFS API presents itself to the programmer in much the same way a standard OS's file system would be seen. This may mislead application developers into thinking that the file system can be thought of as a disk that provides the user with the freedom to write/erase at any frequency. This is not the case! As is the case with any flash file system, the user must be aware of the fact that the flash has a limited number of erase cycles that can be applied to a given sector.

Excluding in-place-modify files, when a file is modified in ANY way, it causes TFS to delete the old and create a new. The file is deleted by simply clearing a bit in the file's header to indicate that it is deleted. When there is no room left at the end of the list of files in flash space, then TFS will automatically start up a defragmentation. How this defragmentation "wears out" the sectors depends on what is being

done. The defragmentation process tries to be as "sector friendly" as possible, but doesn't do anything very sophisticated to do this. The spare sector is usually going to be the sector that is hit the hardest, but even this depends on the organization of the files and the frequency of deletion. If, at the time of defragmentation, the space taken up by deleted (or "dead") files exceeds the size of a sector in TFS, then the spare sector will not be needed for defrag.

Since its tough to make a general statement about how long you can expect your flash to "live", lets take an example of worst case just so that the numbers are put into perspective... Assume you have a 12 sector TFS flash space and every defrag requires every sector to be copied to the spare prior to updating the sector. This means that the spare sector is erased about 15 times per defragmentation. Assuming a defragmentation is done every day (very unlikely), and assuming you have a flash device with a life expectancy of 100,000 erase cycles (many support 1,000,000 now), then your device will last 18 years before it reaches the 100,000 erase count. All of the criteria in this example is very conservative, so in this case, 18 years is an absolute minimum life span.

The bottom line is this: be aware of the fact that the underlying technology is flash and in most cases the life expectancy is not an issue. If TFS is being used as a convenient means of flash maintenance and system upgrade, there is absolutely no need to be concerned with flash life expectancy. If on the other hand, your system needs to be able to write to flash quite frequently, then make sure flash is the correct medium for your storage needs and use a commercially available product!

Field Upgrade Options

There are several ways an application (as well as the monitor itself) can be installed/upgraded on a system running MicroMonitor. One basic assumption is that the target system already has some version of the monitor installed and running on the target. With that as a given, the monitor's XMODEM and TFTP facilities provide mechanisms for upgrades through a serial port or over ethernet. The monitor itself has a TFTP client and server. As a client, TFTP can be used in conjunction with DHCP or BOOTP to automatically pull down an application from some server. Also, the TFTP client can be used with some other startup script based on specific application needs (refer to system startup for a few examples). The TFTP server is always running when the monitor is in control, so pushing a new application down to the target is also an option. Finally, even without an ethernet connection, XMODEM can be used to transfer files to TFS or to install a new monitor. The bottom line is that without any consideration for field upgrade in your application, use of the monitor provides several basic alternatives right from the start.

The set of monitor-provided options do assume that the system is taken out of service during the download process. This may not be an option, so the running application may have to download the new application while in service. Since the files in TFS are copied from flash to DRAM, and then executed out of DRAM space, the update of the application stored in the FLASH is relatively straight-forward. The application must have its own TFTP or FTP server (or some proprietary interface) to support the ability to download a new application file from a host, but that's it. The application can do whatever it wants to TFS. This means that the application (now in DRAM space) can install a new copy of itself in TFS. Then at some point in the future, the system can be reset and the new application will automatically boot. If during the overwrite process something causes the system to reset, then TFS has facilities built in so that the old copy of the application file is re-installed and it would simply start back up. It is important to realize that TFS has this fallback mechanism in place because although it provides the security of knowing that a file can be updated safely, it does require that the old file not be removed from flash space until the new one is 100% installed in flash; hence, this overhead space must be considered.

Upgrade of a multi-file application:

The above scenario is acceptable if the application consists of one file, but what about the case where the application is made up of several files? Say our application consists of files: app, data1, data2, data3. TFS provides the security needed if a reset occurs in the middle of any one file transfer; but what if a reset occurs in the middle of a group of files? Now we have (for example) a new "app" and "data1", but the older "data2" and "data3" files are still installed.

This becomes the responsibility of the application, the file system cannot help out here because it has no idea what files constitute a full set of application files. The application must be able to determine if the other files within TFS are the files that it wants, so an easy way of doing this is to include some kind of "version_sanity" file that contains the name and CRCs of each of the files in a given application build. The application code would then look to this file and make sure that all of the files exist and have the correct CRC. If something fails, the application must deal with this by notifying an administrator or kicking off a TFTP; or perhaps just knowing about it is enough for it to deal with it internally somehow. This gets a bit application specific. The point is that a multi-file application can easily be dealt with using TFS.

Remote update of monitor binary:

Update of the monitor binary itself is a bit different. This is because the monitor is not a file in TFS (since the monitor contains the code that is TFS); so a different set of steps must be followed. Note that in the case of a monitor update, there is a small window of time during which the target's reset vector is erased; hence, during that time if the target is reset or power-cycled, it will not reboot. I mention this warning because it is a possibility although I have never had it happen in a field situation; also, refer to

boot-block safety appnote for an optional mechanism that can be part of the monitor that will protect against the "can't reboot" case. The update is a multi-step process that is typically handled with a host-resident tool (newmon)...

- Reset the target and don't restart the application.
- Optional step (see note below)
- Download the new image of the monitor into RAM.
- Erase the old monitor image.
- Burn the new monitor image into the boot flash.
- Reset the target and allow it to autoboot the applications that were in TFS prior to the monitor upgrade.

Depending on what has been updated in the monitor, this is usually all it takes. If; however, the new monitor includes a modification to the header structure of TFS, then additional steps are necessary for bootup. This is because the monitor-upgrade process does not affect the files stored in TFS flash space; so if the monitor code expects the files to have one header structure and the files have a different header structure; the monitor will claim that TFS space is corrupt and no files will be visible. At this point, the file system would have to be re-initialized (tfs init) and all files re-installed. TFS header updates are rare.

Note: If the monitor does include a TFS header update, then the binary image that newmon transfers to the target could be configured to include the monitor and all the files currently resident on the target. This would require that all files be transferred up to the host and the f2mem utility be used to build an image that includes these files and the new monitor image.

Debugging Your Application

This section will discuss how MicroMonitor can help a developer debug an application. Portions of these capabilities are CPU and compiler independent, but some portions do require code in the monitor that is specific to the target hardware and the compiler used. That being the case, sections of the following discussion that are not CPU and compiler independent may not be available on all target systems. For additional debugging tips, refer to stack trace and profiling pages.

Symbolic Debugging

Definition: the ability to look at variables of various types (integer, short, struct, etc...) in the target system without the need to be aware of their actual physical address.

The monitor's ability to do symbolic debugging comes from the command line interface's (CLI) ability to replace "symbols" on the command line with "replacement strings" from the symbol table. The symbol table for the monitor is a simple file (called symtbl) in TFS. Each line in the file is assumed to contain a symbol name followed by a string to replace the symbol name. In all cases of its usage so far, the string that replaces the symbol name is a hex address, but this is not a requirement. Following is an example that uses symtbl to replace a string with a hex address. Assume we have the following lines in our symtbl file in TFS...

```
IpAddr_binary      0xf0041008
MacAddr_binary     0xf0042000
myVariable         0xf0042040
ipHdr              0xf0048010
testFunction       0xf0100480
```

With that, we can now type in the following command on the monitor console...

```
dm -4d %myVariable 1
```

and it will be converted by the monitor's CLI to...

```
dm -4d 0xf0042040 1
```

The command interpreter then processes the converted line. The result is that whatever was stored in myVariable as a 4-byte integer will be displayed in decimal format to the user (see dm command for details on output formats); with the user unaware of the physical location of the variable. All commands within the monitor can take advantage of this symbolic replacement by the CLI, but certain commands are more applicable to symbolic debugging.

To modify a location in memory, use the "pm" command (similar in syntax to dm). The following line...

```
pm -4 %myVariable 87
```

will place decimal 87 into the location 0xf0042040, assuming it is a 4-byte integer.

To execute a function within the system, the "call" command can be used. The following line...

```
call %testFunction
```

will execute the function "testFunction()" assumed to reside in memory at 0xf0100480. The return value will be displayed and arguments could be passed to it.

Finally, and probably most useful is the "cast" command. The following line...

```
cast ip %ipHdr
```

will display memory starting at location 0xf0048010 as a structure of type "ip" (see cast for more details).

Run-time Trace

Definition: the ability to analyze the behaviour of a system as it is carrying out the tasks it was designed to carry out, without severely affecting the performance and without modifying the code.

Note, first that this is "run-time" not "real-time" trace; although, it is about as close to real time as you are going to get without the help of external equipment. One of the commands in the monitor is called "at". This is an attempt to provide the ability to do some fairly sophisticated run-time state analysis by setting a variety of breakpoints, or hooks, in the code that allow the user to keep track of events. For example, as a simple case, you could set up a counter that would keep track of the number of times a particular function is called. Or you could cause a breakpoint to occur ONLY after some other function has occurred 3 times. There is a difference between a breakpoint and a hook. A breakpoint causes the application to stop and turn over control to the monitor; while a hook simply causes the application to branch into some record-keeping code in the monitor and then continue from the point of the branch.

Note1: An important limitation of the monitor regarding breakpoints is that once a breakpoint (not a hook) has been taken, there is no turning back. The monitor does not support the ability to resume execution of the application. This limitation keeps the monitor simple, because a true breakpoint would have to shutdown all periphery until control was returned to the application. This can be quite tricky and is heavily target-system dependent, so keeping simplicity in mind, it is not a feature of this monitor's debug capability.

Note2: This capability, although implemented for some earlier monitor ports, has not been used much lately; hence, it needs work to be compatible with some of the current CPUs.

Context-Sensitive Stack Trace

Definition: the ability to determine how the system reached a certain point in the code (function nesting) for both single and multi-threaded applications.

What do you do if you hit a piece of code (or take an exception) and want to know how you got there? A stack trace capability is invaluable in cases like this. MicroMonitor has a stack trace capability that allows the user to step back through the function nesting to see each of the functions that have been called to reach the current context. Having this capability allows the user to insert a "man-made" exception into an error leg of the code and upon taking that exception the user can quickly determine how the code got there. Refer to the strace command for more details. Following is an example of a system exception followed by the monitor's strace output...

```
EXCEPTION (offset 0xc00) occurred near 0xf003204c

uMON> strace
0xf003204c: systemError() + 0x4
0xf00265d0: funcA() + 0xe4
0xf00a17fc: TaskMain() + 0x34
uMON>
```

Building a "syntbl" File:

The steps taken to build the syntbl file vary because it is dependent on the way the compiler/linker toolset builds the symbol file (some derivative of 'nm' usually). It is usually some type of tabulated output that can be easily parsed with a few shell commands like awk & grep. If this is not available, there is a tool called monsym that will take an input file and rearrange the columns to be in the format needed for syntbl. Refer to monsym for details. The syntax of each line in the monitor's syntbl file is simply...

```
"<input_string> SP <replacement_string>"
```

two whitespace delimited strings per line.

Profiling Your Application

Ideally, the functions that are most heavily used by the application should be the ones that are most heavily optimized. The purpose behind "profiling" an application is to determine what functions (or tasks) are active the most. This provides the developer with pointers to the code that should be worked on hardest. This section discusses how MicroMonitor can help a developer profile an application.

The "prof" command is used to configure the runtime portion of the profiler and to dump the statistics gathered by the profiler. The runtime portion of this is handled through a call to mon_profiler(). This monitor API function is called with a pointer to a monprof structure...

```
struct monprof {
    int type;
    int pc;
    int tid;
}
```

Refer to monprof.h for the latest information. Currently there are three different modes of operation (bitfields in the 'type' member) supported by this profiler: MONPROF_FUNCLOG, MONPROF_TIDLOG and MONPROF_PCLOG. Each of them have their value and depend on the CPU and facilities available in the target hardware. The application simply includes monprof.h as a header in the file that contains some high-priority interrupt (preferably the system tick) that can call mon_profiler(). It must load the 'pc' entry of the monprof structure with the address that was interrupted by this interrupt handler, and the 'tid' entry with the task id of the running task.

Note that MONPROF_FUNCLOG and MONPROF_PCLOG need the 'pc' entry and MONPROF_TIDLOG needs the 'tid' entry; hence, both the pc and tid entries may not be required information.

Task ID Statistics

Definition: the ability to determine what tasks are statistically the most active.

The prof command is used to initialize the profiler so that it knows how many different TID values to keep track of. Each time mon_profiler is called the TID value is compared to all tid values already logged (binary search), if a match is found, then that TID count is incremented; if no match is found then the new TID value is inserted into the list. The list is kept sorted so that the TID search is kept efficient.

Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

```
struct monprof mp;

mp.type = MONPROF_TIDLOG;
mp.tid = getCurrentTaskId(); /* this is application/RTOS specific */
```

```
mon_profiler(&mp);
```

Function Statistics

Definition: the ability to determine what functions are statistically the most active.

The prof command is used to initialize the profiler based on the content of the "symtbl" file. This file is assumed to have its entries in ascending address order, so the profiler uses the addresses as symbolic ranges into which each interrupted PC will fall into. For example, if the content of symtbl was:

```
main 0x123000
func 0x123804
func1 0x124008
```

then any interrupted PC value between 0x123000 and 0x123803 would be considered a hit to the function main(). This profiling method uses the content of symtbl, so it is wise to remove all but the function addresses from symtbl for this profiling method. Optimizing symtbl provides two benefits: less ram is needed to store this data and less time is taken in the interrupt handler to determine which symbolic range the PC has fallen within. At runtime, each call to mon_profiler() results in a binary search through the list of function symbol addresses looking for a match between the incoming PC and one of the symbol address ranges.

Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

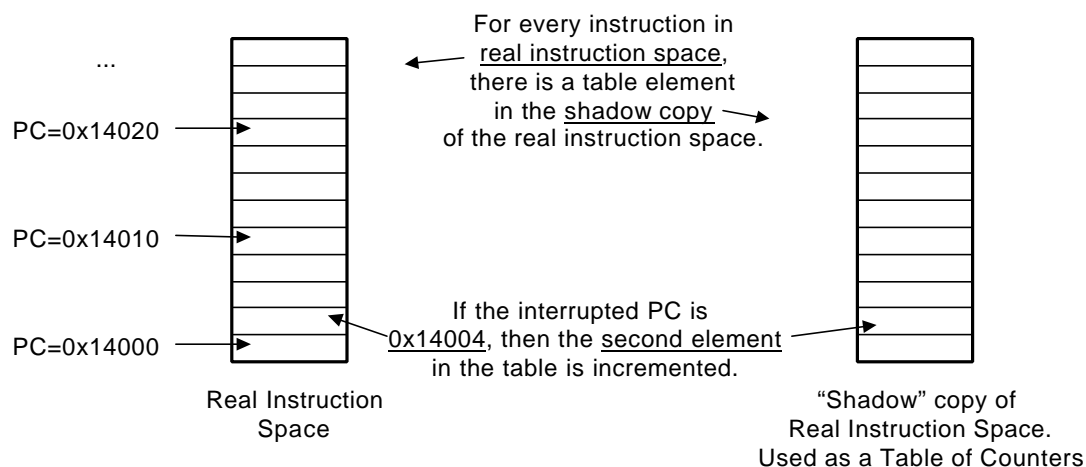
```
struct monprof mp;

mp.type = MONPROF_FUNCLOG;
mp.pc = getInterruptedPC(); /* this is application/RTOS specific */
mon_profiler(&mp);
```

PC Statistics

Definition: the ability to determine what lines of code are statistically the most active.

The prof command is used to initialize the profiler based on the size of the application's instruction space (size of .text section). The runtime profiler assumes that all instructions are of fixed size and that an array of space equal to the size of the instruction space is dedicated to this profiling. At the call to mon_profiler() this mechanism simply takes the interrupted PC and computes an offset into the secondary text array based on the PC value and the start of the real .text space. The location of the offset is incremented; hence, this provides the ability to log each instruction that is interrupted in runtime. If the memory space is available and the CPU does have a fixed size for each instruction, then this is a very efficient profiling alternative, the run-time profiler is simply incrementing a location in a table based on the incoming PC and some already computed offset between the actual text space and the "shadow" text space used to log the data. Refer to the diagram...



As the system tick interrupts various locations in real instruction space, the call to `mon_profiler()` with that address will increment the corresponding location in the "shadow" copy of instruction space.

PC-Level Statistical Profiling

Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

```
struct monprof mp;

mp.type = MONPROF_PCLOG;
mp.pc = getInterruptedPC(); /* this is application/RTOS specific */
mon_profiler(&mp);
```

Example Profiling Session

Let's assume we have an application called 'app'. It is an RTOS-based program with three tasks, and a memory map as is shown by the output of "tfs -v ld app" below. The function "sysTick()" is the system tick handler (high level periodic interrupt), the function `getInterruptedPC()` will return the address of the instruction that was running just as the interrupt occurred and the function `getCurrentTid()` will return the ID of the task that was running prior to the interrupt occurring.

```
uMON>tfs -v ld app
.text      : copy    14200 bytes from 0xf01b308c to 0x00030000
.data      : copy     264 bytes from 0xf01b6804 to 0x00033778
.rodata    : copy     208 bytes from 0xf01b690c to 0x00033880
.got       : copy     16 bytes from 0xf01b69dc to 0x00033950
.bss       : set     204 bytes at 0x00033960 to 0x00
.sbss      : set     12 bytes at 0x00033a2c to 0x00
```

We are running on a CPU that has fixed size instructions (4 bytes per) and enough spare memory to allocate a block of RAM equal in size to the `.text` section of our application (14200 bytes). In the application we insert the following code into the `sysTick()` function:

```
struct monprof mp;
```

```
mp.type = MONPROF_PCLOG | MONPROF_TIDLOG;
mp.pc = getInterruptedPC();
mp.tid = getCurrentTid();
mon_profiler(&mp);
```

Prior to running the application, we must configure the profiler code in the monitor. This is done with the "prof" command. There are several steps:

- Initialize (clear) the profiling statistics and control structures

```
prof init
```

- Configure the profiling mechanisms to be used. In this case we are using the PC and TID logging. For this example we are running an application with 3 task ids and a text section of size 14200 bytes starting at 0x30000...

```
prof tidcfg 3
prof pccfg 4 0x30000 14200
```

- Enabling profiling. Without this, the mon_profiler() function would simply return with no logging performed.

```
prof on
```

At this point, the application can be started and statistics will be gathered. At completion of the application profiling run, the prof command is used to dump the results of the statistics...

```
prof show
```

The output of prof show is something like this...

```
FuncCount Cfg: tbl: 0x00000000, size: 0x0
TidCount  Cfg: tbl: 0x00022000, size: 0x3
PcCount   Cfg: tbl: 0x00022018, size: 0x3778

TID_PROF stats:
00000002   : 1
01114000   : 3
01114800   : 2

PC_PROF stats:
00030804   : 1
00031214   : 1

4 pc out-of-range hits
6 total profiler calls
```

Porting to a New Target

One of the design goals of this monitor package was to make it simple to port to new processors and target hardware. There are a few things in the monitor (timing in particular) which suffer a bit just to keep the porting effort to a minimum. The two most significant points are...

- **The monitor does not use any interrupts from the CPU**

From the porting point of view this is a real luxury. It makes it much easier to get a port to a new processor up and running. You don't have to worry about how a processor's interrupt handler mechanism is set up and you don't have to worry about making sure you save all the registers appropriately. Additionally, the serial port and ethernet drivers are much less complex because everything is polled. Not having interrupts means that it is a bit more difficult (and somewhat inaccurate) to deal with elapsed time. If you have no clock reference, then you don't really know how much time has passed; however, to get around this a bit, there is a calibration that is made by the "sleep" command that allows you to tweak this timing a bit. Still, it is admittedly inaccurate, but worth it. Keep in mind that the drivers in the monitor are not intended to be reused by the application anyway. If the overlaying application needs a real driver, then that application needs to provide it.

- **The monitor code assumes that initialized data is not writeable**

In most ports, the monitor code is running (fetching instructions) directly out of flash (see note1 below). The monitor does not copy any .data space to ram, so it too is accessed in runtime from flash. This means that the monitor's initialized data is not writeable. Yep, this is something to be aware of; but, once again, it makes the port much easier because now there is no need to figure out how the compiler toolset allows the data section to be remapped to ram space and there is no need to copy the data section into ram space (see note2 below). On the other hand, the monitor DOES assume that on startup the .bss section is initialized to zero (see start.c). This is ok because it is easy to clear a block of memory regardless of the toolset.

Note1: the functions that perform write/erase operations on the flash are executed out of RAM. This is necessary because most flash devices do not allow the device to be modified if instructions are being fetched from it at the same time.

Note2: initialized data in an embedded system must exist in non-volatile memory, and prior to application startup, if the data is to be writeable, it must be copied from non-volatile space to ram. This is done so that at startup the data is properly initialized and at runtime the application has the option to change it. This means that some compiler-specific directives must be used to make sure that the data space in flash is copied from flash to some space in RAM where the application code will be able to access/modify it. This complexity is avoided in the monitor by carefully making sure that no initialized data is modified in the monitor code.

Note3: in some cases the monitor is specifically built so that it boots out of flash, but copies itself to RAM. If this is the case, then the above discussion on non-writeable initialized data is not applicable. Also, in this case, since the entire monitor runtime is in RAM space, there is no need to write flash drivers that copy themselves to RAM.

First things first:

Before beginning your port, make sure you are aware of what the monitor needs and provides. It assumes a CPU with linear address space (no bank switching, and if x86, then use a memory model that eliminates the need for C-code to deal with segmentation). To use TFS, an obvious assumption is flash. TFS assumes that the flash is accessible directly on the address/data bus, there is no IDE or I2C (or any other) interface between the memory allocated to TFS and the CPU (future versions of TFS may support these interfaces). Make sure you are aware of the flash life expectancy issues with TFS and based on the features that you decide to use within the monitor platform, make sure you have the memory space to allocate for it.

There are 4 sections of code within the monitor that need to be modified when porting to a new target. Based on the ports already available, some of these sections may be re-usable from other ports that have already been made, so it is wise to check with the author prior to starting this work.

- Code to handle CPU reset, basic exception handling and getting up to 'C' level (see `reset.s` and `start.c`). With this in place, you know you have a grasp on the whole build process. Ideally, your hardware will have some means of output (like an LED) that is easily accessible so that you can be sure your code is running.
- Code for a polled serial port driver (see `cpuio.c`). With the serial driver working, you immediately inherit the ability to download applications into ram. One thing to make sure at this point is that your definition of `LOOPS_PER_SECOND` in `config.h` is close. This is important because many portions of the code use this count value for a delay loop; so if it isn't even close, facilities (like XMODEM) may not work efficiently.
- Code for a flash device driver. Once the flash device functions are installed, TFS will come up quickly.
- Code for a polled ethernet driver (see `etherdev.c` and start with `etherdev_template.c`). After this, TFTP, DHCP&BOOTP all just work! (really!).

Directory structure:

There are currently two top-level directories that contain the code... the common directory and the target-specific directory. There are a few sub-directories under common and in general, the code here should be left untouched. This common code has been designed to interface to the target-specific code found under the `targets/targetname/monitor` (where `targetname` is the name of some target platform) directory.

When starting a port, retrieve the latest tar file from the author and copy it to some workspace. If you're lucky, there may already be a port to a platform that is similar to your platform. Create a new target-specific directory. Use a directory name that contains an acronym for your target followed by a string representing the CPU (ie.. `abcsh2` is the ABC board with the Hitachi SH2 processor). Also, under the common directory, if you will be writing a new flash device driver, add a new directory that will eventually contain the flash code for the device you are going to be using. For the sake of this discussion, let's assume the target-specific directory name is `ABC960` and the common/flash directory is `29XX32`. Into each of these new directories copy the content of one of the supplied target-specific directories that is closest in architecture to your design.

RunTime Execution, FLASH or RAM?

The monitor can be built to run out of flash at runtime, or it can be built to copy itself from boot flash to RAM and run from RAM at runtime. Refer to different targets to see the different implementations. In general, the decision is based on the needs of the system. In some cases, RAM is limited, so space cannot be allocated to the monitor, so it is left in flash. In some cases the boot flash is a slow 8-bit wide device, so fetching from that device at runtime is inefficient, so the monitor is copied to RAM for speed.

If running from flash, then the monitor is built as a single image (refer to the SH2 evaluation platform for an example of this). If running from RAM, then the monitor is built as two images. The first image is the code that must start off in boot flash. The second image is converted to an array that is included in the first image. The first image then simply copies that array to some pre-defined location in RAM and branches to that point. Refer to the file `cstart.c` for the code that does the copy, and look to any of the Cogent Computer Systems ports (target directory names that start with `csb`). Note that eventually, this dual-image mode will also support the ability to decompress the array out of flash and into RAM for execution (not there yet!).

There are several reasons why one solution is better than the other, it just depends on what you're building and what constraints you have to deal with, so just pick the one that works best for your system.

The config.h file:

At this point we are ready to start modifying code. The first thing to do is to reduce the complexity of the monitor by turning off the majority of the features established by the ABC90/monitor/config.h file. In this file there is a block of INCLUDE_FEATURE macros (where FEATURE is some feature-specific string). Following is what can be used as the minimal configuration...

```
#define FORCE_BSS_INIT          1

#define INCLUDE_MEMCMDS       1
#define INCLUDE_PIO           0
#define INCLUDE_EDIT         0
#define INCLUDE_DEBUG        0
#define INCLUDE_DISASSEMBLER 0
#define INCLUDE_UNPACK       0
#define INCLUDE_UNZIP        0
#define INCLUDE_ETHERNET     0
#define INCLUDE_TFTP         0
#define INCLUDE_DHCPBOOT     0
#define INCLUDE_TFS          0
#define INCLUDE_XMODEM       1
#define INCLUDE_LINEEDIT     0
#define INCLUDE_EE           0
#define INCLUDE_FLASH        0
#define INCLUDE_CRYPT        0
#define INCLUDE_TFSAPI       0
#define INCLUDE_TFSCLEAN     0
#define INCLUDE_TFSSYMTBL   0
#define INCLUDE_TFSSCRIPT    0
#define INCLUDE_TFSCLI       0
#define INCLUDE_GDB          0
#define INCLUDE_STRACE       0
#define INCLUDE_CAST         0
#define INCLUDE_EXCTEST      0
#define INCLUDE_IDEV         0
#define INCLUDE_STRACE       0
#define INCLUDE_CAST         0
#define INCLUDE_REDIRECT     0
#define INCLUDE_QUICKMEMCPY  0
#define INCLUDE_ARGV         0
#define INCLUDE_PROFILER     0
#define INCLUDE_BBC          0
#define INCLUDE_MEMTRACE     0
#define INCLUDE_BOARDINFO    0
```

This configuration provides a very basic monitor... a serial port driver with xmodem for download and memory commands for interfacing to target memory. Refer to the file inc_check.h for the most up-to-date list of these definitions and their meaning. The files that need the work now are reset.s and cpuio.c. Note the definition of FORCE_BSS_INIT. This is used by start.c to unconditionally clear the monitor's BSS space. Later, when things settle down a bit, this definition can be removed so that a soft reset of the monitor does not wipe out all of the environment.

If this is the first attempt at this processor, then spend some time looking for some example code from the CPU manufacturer's website. This is usually not too hard to find. For the very first build attempt, don't even worry about the exception handlers. Once this is working, then add the exception handling stuff found in ABC960/monitor/except.c and ABC960/monitor/vectors.s. Note that for some ports, the exception handlers (and other cpu-specific code) may be found under common/cpu directory because they are generally applicable to multiple platforms. This is also true for driver code (ie serial port and ethernet) that may be written for a peripheral that is built in to a particular processor. With this in mind, be aware that the filenames may be slightly different (except_405.s, uart_SH2.c, etc...).

MicroMonitor
Application Notes

The Flash Driver:

At this point, you have a basic monitor working with a serial port. Next is flash. Start by enabling `INCLUDE_FLASH` in `config.h`. You can use `XMODEM` to download small programs and run them by executing the "call" command. There are three fundamental functions needed for each flash device, and possibly two additional functions, depending on the need of the system:

- `Flash_Init()`: initialize the control structures used by the flash API.
- `Flash_Erase()`: erase a sector, or group of sectors.
- `Flash_Write()`: write any number of bytes starting at any point within a flash bank.
- `Flash_Type()`: retrieve the manufacturer and device code.
- `Flash_Ewrite()`: erase and write in one step.

This text will not dig into any single device driver implementation. It is assumed that the reader has some working knowledge of the flash device. There are plenty of examples to start with, and chances are pretty good that the driver already exists. The goal here is to describe in general terms the philosophy behind MicroMonitor's flash driver strategy. Based on that, following are a few discussions on various topics that the reader should be aware of...

What is a "bank" of flash?

The term "bank" here refers to a device or group of parallel devices that are accessed by a single CPU read or write. This may be one, two or four devices depending on the target configuration. If a single 8-bit boot flash device is in the system, that is a bank. If there is a set of 2 16-bit devices configured in parallel to form one 32-bit wide flash peripheral, that would also be one bank. Similarly, 2 or 4 8-bit devices in parallel could be a bank. Note that within any one bank, only one device type will exist; and when sector erase is performed, it is performed on as many devices as are in parallel.

Flash operations copied to ram?

Most (not all) flash devices do not support simultaneous operations. In other words, if you are fetching from the device, you can't erase (or write to) a sector in that device. Some devices do allow you to fetch from one portion of the device while operating on the other portion; however, for simplicity this is ignored. MicroMonitor simply assumes that the code that is running some flash operation must be in RAM (or in some other device). This is accomplished in one of two ways:

If the monitor is built to fetch from flash in runtime, then the flash operations must be copied to RAM when the monitor is started. If the monitor copies itself to RAM for runtime execution, then by default, the flash operations are in RAM so there is nothing further to do. The tricky part is the first case. If a function is to be copied to a different location, then it must be relocatable or "position independent". This can be tricky, and definitely takes some additional thought on the part of the developer. I don't want to get off on the subject of position independent code; however, the most important thing to realize if there is a need to write a position-independent function, is that you should avoid making any calls to any other functions from within that function. This makes it tough to debug because you can't simply call `printf()`.

The decision regarding whether or not the functions are copied to RAM is dealt with in `FlashInit()` using the function `flashopload()`. The important thing to note here is that if the functions are to be copied to RAM, then I&D caches should be off during this operation (or precaution should be taken to assure that the instructions copied to the ram space are flushed from the data cache).

The flashinfo & sectorinfo structures

For each bank of flash, there is a *flashinfo* structure that must be initialized to provide the driver with basic information about the flash device. This structure contains the high-level information needed by the driver... The address range occupied by the device, the device id, pointers to the functions that operate on the device, and a pointer to a table of *sectorinfo* structures. This whole strategy assumes that the flash is broken up into sectors, so the table of *sectorinfo* structures contain sector-specific information. This

includes the size, beginning and end address, the sector number (as it exists in the system, not the single device) and the protection status of the sector¹. There is one flashinfo structure for each bank and one sectorinfo structure for each sector in each bank. Note that if the bank is made up of a parallel set of devices, then the sector information must account for this. In other words, if the bank consists of two 29f040² devices in parallel, there is a total of 8 sectors in the bank and each sector is 128K (2x64K). The initialization of the flashinfo and sectorinfo structures is done in the `FlashInit()` function at system startup. Refer to `common/flash.h` for definition of these structures.

As of Sept 2002, there are multiple "styles" that can be used to implement the flash drivers. The intent here is to not eliminate the original method for those who are comfortable there, but to provide a needed improvement in the way the drivers are structured. Note that this is simply a "style" issue. The interfaces are essentially identical, so either approach can be used. The two techniques are distinguished by the location in which the source is stored...

common/flash: This directory has multiple directories below it, with each directory dedicated to a single target hardware flash layout. The assumption is that you are building a driver that will work for your target's flash layout. If you have more than one device, then that directory would contain code for each device. If you want to support a "simulated" flash-in-ram device for use by TFS, that detail would also be in that directory of code. So, for example, if I have one target that has a 29F040 boot flash and a 29LV160 flash for file storage, and another target with a 29F040 boot flash and a 28F128 for file storage, each directory would contain device interface code for their set of flash devices. The secure thing about this approach is that the directory will only be modified when a change is being made to that target; hence, you will have a platform on which you can test the driver changes. The inconvenient thing about this approach is that different target hardware may use the same device (the 29F040 in this example); however, the code must be duplicated for each target. This approach was the original approach for MicroMonitor because I did not want to have to worry about "adjusting" the driver for one target only to find out that the adjustment broke something for some other target.

Within each directory are at least three files. A file containing the code that would have to be relocated (usually contains "pic" in the name, pic=position-independent-code). A file that contains the basic establishment of the flash control structures and a device header file. If more than one device is supported, then more files will exist.

common/flashdev: This directory has no subdirectories. The goal here is to provide a simple, single-device driver for targets that have only one flash device (which is probably the most common); but to also allow the target to include multiple files if the target has multiple flash devices. The point is that each file is specific to a particular device set up with a particular configuration (not a particular target hardware platform). The term "configuration" here refers to the width of the single device and the width of the overall bank and the number of devices put in parallel to form that configuration. The naming convention for the device files is as follows (each device has a header file with the same name)...

```
DEVICENAME_DBW_DIP
```

```
where...
```

```
DEVICENAME    is the device name (duh)
DBW           is the configured device data bus width (in bits)
DIP           is the number of devices in parallel (to form a bank)
```

Many flash devices can be configured in x8 or x16 bus width mode, so for example, the Am29LV160D can have several different source files depending on its configuration in the target system...

¹ MicroMonitor's flash interface provides a basic level of "soft" protection to each sector. If protected, then the sector can only be operated on if the flash protection window is opened. This is done with the "flash opw" command and remains opened for only the next command issued at the monitor CLI.

² The AM29F040 contains 8 sectors, each 64K

```

am29lv160d_08x1.c    -> Am29LV160D in 8-bit mode, 1 device in parallel
am29lv160d_08x2.c    -> Am29LV160D in 8-bit mode, 2 devices in parallel
am29lv160d_08x4.c    -> Am29LV160D in 8-bit mode, 4 devices in parallel
am29lv160d_16x1.c    -> Am29LV160D in 16-bit mode, 1 device in parallel
am29lv160d_16x2.c    -> Am29LV160D in 16-bit mode, 2 devices in parallel

```

Refer to the README file in the common/flashdev directory for more details.

Description of FLASH configuration parameters in config.h...

Note that the use of these definitions depends on how you have configured your flash driver, so actual naming may be slightly different.

FLASHBANKS	Set this to the number of different banks of flash your target system will have. In general, "BANK" means device, except when devices are in parallel. So, if you have a single 16-bit flash device or 2 8-bit flash devices in parallel to form a 16-bit access to flash, both configurations would be considered a single bank. If you have two 16-bit devices back-to-back in memory space, this would be considered 2 banks.
FLASH_BANK0_WIDTH	Set to the width of flash bank zero. This is in 8-bit increments, so a 16-bit bank has a width of 2.
FLASH_BANK0_BASE_ADDR	Base address of flash bank zero.
FLASH_LARGEST_SECTOR	Size of the largest sector in all of the flash banks
FLASH_PROTECT_SIZE	This is a definition used in some of the early flash drivers. This value would be set to the highest location in flash that needs software protection. The assumption was that all sectors occupying the space between 0 and this value would be protected. This doesn't work well with targets that do not have their boot flash at address 0, so it has been replaced with FLASH_PROTECT_RANGE. If you are writing a new flash driver, use FLASH_PROTECT_RANGE instead of this.
FLASH_PROTECT_RANGE	This is some string formatted as individual, comma-delimited or dash-delimited ranges. A few valid choices would be "0", "0-3", "0-3.8". Protected flash simply means that the flash command will require preliminary "opw" request for those banks to be modified. See the flash command for more details. This is a replacement for FLASH_PROTECT_SIZE. If you are writing a new flash driver, use this instead of FLASH_PROTECT_SIZE.
WIDTHX	This macro is only used with flash drivers that are written to be configurable for different widths simply by establishing this macro. Currently, the only device that supports this in the monitor code is the 29F040. Refer to that header file for details.

Note that if there was more than one flash bank in your target system, then there would be an additional set of FLASH_BANKN_XXX macros for each additional bank.

Once the flash driver is operational, the Xmodem -B command can be used to download new monitor binaries without the need for an external programmer.

NOTE: This flash implementation is still under construction. I've been having conversation with a few folks who have some really good ideas on how to improve the driver structure. Not to change the API, but to simply clean up the current implementation for better reuse and code consistency.

Configuring TFS

TFSSPARESIZE	This value should be set to the size of the largest flash sector that is dedicated to TFS
TFSSECTORCOUNT	This should be set to the number of sectors that are to be dedicated to TFS file storage space. This does not include the SPARE sector.
TFS_DEVTOT	Total number of non-contiguous devices that TFS spans. This is usually just 1, but depends on the target configuration.
TFSSTART	Base address of the beginning of TFS.
TFSEND	End address of TFS file storage space that started at TFSSTART. Note that this address should NOT include the space used by the spare sector. Usually, TFSEND is 1 byte less than TFSSPARE because the spare sector lies just after the end of TFS storage space.
TFSSPARE	Base address of the spare sector that is to be used by the TFS storage space starting at TFSSTART.
TFS_EBIN_ELF TFS_EBIN_COFF TFS_EBIN_AOUT	If there is a need to use TFS's loader, then one (and only one) of these three definitions should be set. The choice here depends on the cross compilation toolset being used to build the application code.

There is also `tfsdev.h`. This header file is used to help the `tfs` code with the possibility of there being more than one memory device over which TFS will span. For most targets, TFS will only span one contiguous block of flash; however, TFS does support multiple non-contiguous blocks. The `tfsdev.h` structure establishes this. So for most targets, the structure will look like this...

```
struct tfsdev tfsdevtbl[] = {
    { "//040/", TFSSTART, TFSEND, TFSSPARE, TFSSPARESIZE, TFSSECTORCOUNT, TFS_DEVTYPE_FLASH, },
    { 0, TFSEOT, 0, 0, 0, 0, 0 }
};
```

The first member of the first entry in the table is the name to be associated with the device. See `tfsprivate.h` for details. TFS can be set up so that the device used is somewhat dynamic (within a family). If you have a target that may be configured with a 29F040 or a 29F010 for example, then set the `TFS_DEVTYPE_DYNAMIC` bit and all other entries NULL. This assumes that you have written your flash driver to deal with this. See the 29F040 code for an example.

```
struct tfsdev tfsdevtbl[] = {
    { "//040/", TFSSTART, 0, 0, 0, 0, TFS_DEVTYPE_FLASH | TFS_DEVINFO_DYNAMIC, },
    { 0, TFSEOT, 0, 0, 0, 0, 0 }
};
```

If there is a need to configure a second device, then the parameters for that second entry would be taken from a second set of parameters that would be established for the second device in `config.h` (like `TFSSTART_1`, `TFSEND_1`, `TFSSPARE_1`, etc...). The structure would then simply have a third entry...

```
struct tfsdev tfsdevtbl[] = {
```

```

{ "//040/", TFSSTART, TFSSEND, TFSSPARE, TFSSPARESIZE, TFSSECTORCOUNT, TFS_DEVTYPE_FLASH, },
{ "//NVRAM/", TFSSTART_1, TFSSEND_1, TFSSPARE_1, TFSSPARESIZE_1, TFSSECTORCOUNT_1, TFS_DEVTYPE_RAM, },
{ 0, TFSEOT, 0, 0, 0, 0, 0 }
};

```

If there is a need to configure a portion of RAM to be allocated as a "fake" TFS flash device, then refer to the app-note "Allocating a Section of RAM as a TFS Storage Area" for a detailed walkthrough of that procedure. Keep in mind that these additional TFS devices should be configured in after the port has been completed. For the basic porting process, just configure the simplest TFS configuration.

Miscellaneous entries in config.h.

DEFAULT_ETHERADD	If all else fails, this will be the ethernet address used by the monitor. If not specified in config.h, then ether.h will assign 00:00:00:00:00:00 and the ethernet interface is not turned up at runtime.
DEFAULT_IPADD	If all else fails, this will be the IP address used by the monitor. If not specified in config.h, then ether.h will assign 0.0.0.0.
APPRAMBASE_OVERRIDE	By default, the monitor will automatically establish a base address at which the application code can start assuming ownership of. This is done by simply finding the next 0x1000 modulo address after the end of the monitor's .bss section. This address is loaded into the APPRAMBASE shell variable and used by both the monitor and user for various things. If there is a reason to override this default setting, then put the override value in this definition; otherwise, this can be omitted.
BOOTROMBASE_OVERRIDE	Similar to APPRAMBASE_OVERRIDE, this value is automatically set to what the monitor sees as its base FLASH address. If for some reason this needs to be overridden, then it should be loaded into this macro.
FORCE_BSS_INIT	When the monitor first starts up, if it is a warmstart, then the .bss space of the monitor is not initialized to zero so that certain state variables (and shell variables) will survive a warmstart. If this is not desirable, then set this flag and all of the monitor's restart options will look like a coldstart. For source code use of these three macros, see start.c.
LOOPS_PER_SECOND	Since the monitor does not use any interrupts, all timing is based on loop counts. To get this at-least close to being accurate, this macro should be set to the appropriate value. Use the sleep -c option to determine what the optimum setting is for this macro.
SYMFILE	This macro defines the filename used by the monitor as the symbol-table file.
CPU_NAME	String that contains the name of the CPU. This is displayed as part of the boot-up header.
PLATFORM_NAME	String that contains the name of the target platform. This is displayed as part of the boot-up header
ALLOCSIZE	The amount of memory space that is to be allocated to the monitor's own heap. The heap in the monitor does not simply grow up from the end of .bss. It is a statically allocated array within the monitor's own .bss space. This avoids a conflict with other applications that

	may want to use the memory in some other way.
MON_CMDLIST_HEADER	If specified, this is assumed to be a string (used by the "help" command) that contains text that will be output at the top of the portion of the command list that displays the monitor's built-in commands.
APP_CMDLIST_HEADER	If specified, this is assumed to be a string (used by the "help" command) that contains text that will be output at the top of the portion of the command list that displays the commands that have been installed by the application (using <code>mon_addcommand()</code>).
DONT_CENTER_MONHEADER	If defined, then this simply disables the monitor's default action to center the banner that the monitor dumps to the console after a reset.

A Few General Suggestions

- It is important to **take little steps** when doing a port. The `config.h` `INCLUDE_XXX` macros allow you to do this. Don't try to build the whole platform in one shot. One step at a time will get you there a lot faster!!!
- **Don't even enable cache** if your CPU has it. Just leave it off until you have the port complete and you build up some confidence in what you have.
- **Make sure optimization is turned off.** Eventually, you can enable it and things should be made to work with optimization; however, eliminate that complexity when starting a new port.
- In all of the driver polling loops insert some type of timeout. If the timeout occurs, depending on where it occurs, you can do a few different things: if the timeout occurs in a UART-related poll, then blink an LED, or do SOMETHING to let the outside world know where the code is. If the timeout occurs somewhere after the uart interface is known to be up and running, then call some simple error message that will print out the serial port.
- In the `start()` function, make sure that `ramstart` & `ramend` variables are properly pointing to the beginning and end of the monitor-owned `.bss` space. Also, when first bringing up a new port, force this bss initialization to be done by **defining the `FORCE_BSS_INIT` macro** in `config.h`.
- When writing `warmstart` code (usually in `reset.s`), be aware that it may be called by an actual reset (or power cycle) or it may be called as a result of an exception. If called by the monitor's exception handler, then it is important to flush data cache prior to disabling it (at startup). This is important because the exception handler loads a table with the current context of the CPU (registers), so if that data space is in cacheable space, and it isn't flushed, it will be lost; hence, not retrieveable by the monitor for debugging the cause of the exception.

Booting Linux (or any "process-model OS") through MicroMonitor

The MicroMonitor platform is used as a startup environment for many different embedded system applications. Under the context of this discussion, there are three main types of applications...

- Those that run single-threaded, without any need for an embedded operating system.
- Those that run multi-threaded, using an RTOS that runs with a flat, basically unprotected memory space.
- Those that run multi-processed, using an RTOS that provides memory protection between processes.

For the sake of this discussion, the first two types can be bunched together, leaving us with those applications that run with per-process memory protection and those that do not. This brief appnote is for intended for those that run with per-process memory protection. There are some distinct differences and limitations that apply to this type of environment, and the intent here is to bring these limitations forward.

Note that this discussion is based on the author's **limited** experience with booting Linux, so if you read this and find it in error, please inform the author.

The biggest issue regarding the process model is that once the monitor turns over control to the application, the application (usually) no longer has the ability to execute any code out of the memory space used by the monitor .text. Refer to the Programming API documentation of the monitor, and note that this entire API is not applicable in cases where the application is process based. The biggest hit here is that TFS is not accessible at application runtime; however, in the case of Linux (which is currently the most prevalent process-model OS being used with MicroMonitor) JFFS is usually used and this is not a big loss.

The other loss in this case is the fact that the application is not able to retrieve any environment that was previously established by the monitor because the `mon_getenv()` hook (part of the API) is inaccessible. This too can be overcome quite easily with the following proposal...

Typically Linux is started up by transferring some image into RAM (from either TFTP or TFS) and jumping into the starting point of the image. Usually, the entry point assumes that a few parameters contain pointers to some target-specific data that allows the Linux image to get in sync with the target it is running on top of. In my experience, this interface is slightly different for each target Linux has been booted on. My initial attempt at booting Linux followed this pattern but with a common wrapper (the `lboot` command) that pulled some of the target specific ugliness into a common command interface. The underlying code was still target-specific, and could be messy. Here is an alternative that I think applies to any target running any process-model OS...

At the entry point of the OS, the code expects one argument. That argument is simply a pointer to an ASCII string of "name=value" pairs that can be parsed by the application to determine the environment on which it is booting. This allows the monitor to come up and establish the environment in a clean and target-independent manner, and also provides a common means for Linux to retrieve the underlying environment. This method is 100% independent of the target.

The monitor's "set -e" command supports this. The "set -e" will build a string within monitor's .bss space that contains every environment variable (and its value) currently in the system, and it also populates a new environment variable with the location of this string. So, for example, if my system had the following environment variables...

```
IPADD = 1.2.3.4
```

```
ETHERADD = 11:22:33:44:55:66
NETMASK = 255.255.255.0
GIPADD = 1.2.3.1
```

then the command "set -e ENVIRON" would build a string as follows...

```
IPADD=1.2.3.4\nETHERADD=11:22:33:44:55:66\nNETMASK=255.255.255.0\nGIPADD=1.2.3.4\n\0
```

where...

\n is newline and \0 is NULL

and...

the environment variable ENVIRON would contain the address in memory that the string was placed.

This could be built into an lboot command if needed, or the bootup can simply be a script that looks something like this...

```
tfs ld linux_app          # Load image from TFS elf file
set -e ENVP               # Establish the environment string
call $ENTRYPOINT $ENVP   # Jump into entry point with pointer to environment string.
```

Optionally, the load line could be...

```
tfs cp image IMAGE_ENTRY # Transfer image from TFS flash to RAM address IMAGE_ENTRY
set -e ENVP               # Establish the environment string
call IMAGE_ENTRY $ENVP   # Jump into entrypoint with pointer to environment string.
```

or...

```
tftp 1.2.3.4 get linux_app IMAGE_ENTRY # Transfer image from remote host to RAM
set -e ENVP                             # Establish the environment string
call IMAGE_ENTRY $ENVP                   # Jump into entrypoint with pointer to environment string.
```

All three have the same philosophy, just use different mechanisms to transfer the image to RAM. Notice that steps 2 and 3 are the same for each method.