

MICRO-MONITOR

An Embedded System Boot Platform

Application Programmer's Interface

Revision date: November 11, 2002

MicroMonitor Application Programmer's Interface

Table of Contents

This document focuses on how the application code and the monitor get connected. The monitor is a standalone executable, and for the most part, the application is designed as a standalone executable, so how do the two programs communicate when they are together on a target system? The following information will answer this question and also provide the user with a description of the API that is provided to the application by the monitor once this connection is made.

THE MONITOR/APPLICATION RELATIONSHIP	4
THE MONITOR-TO-APPLICATION CONNECTION	4
APPLICATION-PROVIDED FUNCTIONALITY	5
APPLICATION-PROVIDED MUTUAL EXCLUSION	5
APPLICATION-MONITOR HOOKUP: A SMALL, SINGLE-THREADED EXAMPLE.....	5
APPLICATION-MONITOR HOOKUP: A VXWORKS EXAMPLE.....	6
USE OF APPLICATION-PROVIDED LOCKOUT FOR THE MONITOR API.....	10
APPLICATION INSTALLATION ON A MONITOR BASED SYSTEM.....	14
EXTENDING THE MONITOR'S HEAP	14
AN APPLICATION THAT USES THE MONITOR'S MALLOC:.....	15
USING ZLIB TO DECOMPRESS FILES IN TFS:	15
MICROMONITOR API FUNCTION DESCRIPTIONS:	16
MON_ADDCOMMAND()	17
MON_APPEXIT()	18
MON_COM()	19
MON_CPRINTF()	20
MON_CRC16()	21
MON_CRC32()	22
MON_DECOMPRESS()	23
MON_DOCOMMAND()	24
MON_FREE()	25
MON_GETARGV()	26
MON_GETBYTES()	27
MON_GETCHAR()	28
MON_GETENV()	29
MON_GETLINE()	30
MON_GOTACHAR()	31

MON_INTSOFF()	32
MON_INTSRESTORE()	33
MON_MALLOC()	34
MON_PIOCLR()	35
MON_PIOGET()	36
MON_PIOSET()	37
MON_PRINTF()	38
MON_PUTCHAR()	39
MON_RESTART()	40
MON_SETENV()	41
MON_SETUSERLEVEL()	42
MON_SPRINTF()	43
MON_TFSADD()	44
MON_TFSCLOSE()	45
MON_TFSCTRL()	46
MON_TFSEOF()	48
MON_TFSFSTAT()	49
MON_TFSGETLINE()	50
MON_TFSINIT()	51
MON_TFSIPMOD()	52
MON_TFSNEXT()	53
MON_TFSOPEN()	54
MON_TFSREAD()	56
MON_TFSRUN()	57
MON_TFSSEEK()	58
MON_TFSSTAT()	59
MON_TFSTRUNCATE()	60
MON_TFSUNLINK()	61
MON_TFSWRITE()	62

The Monitor/Application Relationship

In much of the discussion regarding the monitor, there is distinction made between the application and the monitor code. This is done simply to make it clear that the monitor and the application are two totally isolated programs as far as their linkage is concerned. In general, if the monitor is modified, there is no need to rebuild the application to compensate for this; and likewise, if the application is modified, there is no need to rebuild the monitor to compensate. Compare the monitor and the application program to a PC and some DOS program. The PC, without application programs doesn't have much value, but when you combine the two, they compliment each other very nicely. The PC provides the "platform" for the DOS application just as the monitor provides a "platform" for the embedded program.

Ideally, the monitor and the application are thought of as standalone programs, meaning that neither of them need the other to run. This is *almost* true. The monitor is truly standalone. It boots the target system and interfaces to the user through a console and/or network connection. The application, on the other hand, *may* be standalone. It can be installed into the memory of the target system and can be designed to assume nothing regarding the environment provided to it by the monitor; but that defeats the whole purpose of the monitor.

In a typical MicroMonitor based system, the monitor provides a platform that the application can assume exists when it starts up. This platform includes several very useful capabilities that can be taken for granted by the application. For example, in a MicroMonitor based system, the application doesn't have to know anything about field upgradeability, it just comes with the monitor. Compare this to a program on a DOS machine. That program doesn't know how it will be put on the PC, it just knows that it has to run on the PC. The underlying PC, with its disk drive and communication interfaces will take care of getting the program onto the PC.

So, the relationship between monitor and application is complimentary. The standalone monitor doesn't do much application-specific stuff; however, when you put an application ontop of a monitor-based embedded system, the two live very happily together.

The Monitor-to-Application Connection

Like an application/BIOS relationship, the application/monitor relationship can vary. The application can use the functionality provided by BIOS, or it can be clever and get around it. Similarly, the application that runs out of TFS through the monitor can use certain monitor facilities, or it can totally ignore the fact that the monitor is present, and execute 100% on its own.

In most cases, the facilities provided by the monitor (particularly TFS) are considered useful in application space, so the application must "connect" to the monitor. For each target, there is some "well known address" provided by the monitor so that the application can make the connection. This well known address is a location that is assumed to contain a pointer to the moncom() function within the monitor. In addition, the application includes one object file that is built from a file that is considered one of the "common" blocks of code within the monitor, monlib.c. This file could just as easily be compiled by the application build, but it is kept as part of the monitor because it is generic code, independent of the target or application that uses it.

The first call that must be made by the application is a call to monConnect(ulong addr,void (*lock)(), void (*unlock)()). This function must be given the "well-known-address" and optionally, two additional function pointers that will provide the monitor interface with an application-provided mutual-exclusion mechanism (more on that later). For basic connection between monitor and application, the second and third arguments to monConnect() can be NULL. After monConnect() returns, it is then safe for the application to assume that the whole set of "mon_" API functions are available for use. For a detailed discussion on the use of application-provided lock and unlock functions, refer to the appnote: "Use of Application-Provided Lockout for the Monitor API".

Application-Provided Functionality

To this point, our discussion has been with regard to how we can hook up the monitor and the application so that the application can use some of the monitor's functionality. Now we will discuss the fact that the monitor must be given some function pointers to be able to run as part of the application.

When the monitor is running stand-alone, it has established its own driver interface to the target hardware. For example, it calls its own function to write a character to the console (UART) port of the hardware. Now that the application has taken over the hardware, the application may have re-established these interfaces, so the application must now provide the monitor with pointers to functionality that it needs (for example, to interface to the console port). Notice in monlib.c that all of the calls to moncom() are "GET_XXX" functions. This is the application "getting" connections to each of the "mon_" functions. The file monlib.h contains all of these #define definitions, but notice that it also has other macros that are not "GET_XXX" type macros. These macros provide the interface to the monitor that allows the application to give the monitor pointers to certain functions...

- CACHEFTYPE_DFLUSH: allows the application to give the monitor a pointer to a data-cache flush routine with the following prototype: dcache_flush(char *addr, int size).
- CACHEFTYPE_IINVALIDATE: allows the application to give the monitor a pointer to an instruction cache invalidate routine with the following prototype: icache_invalidate(char *addr, int size).
- CHARFUNC_PUTCHAR: allows the application to give the monitor a pointer to a putchar function.
- CHARFUNC_GETCHAR: allows the application to give the monitor a pointer to a getchar function.
- CHARFUNC_GOTACHAR: allows the application to give the monitor a pointer to a function that, when called, will return 1 if there is a character queued up on the console port; else 0.
- CHARFUNC_RAWMODEON: allows the application to give the monitor a pointer to a function that will enable "raw" mode on the console port.
- CHARFUNC_RAWMODEOFF: allows the application to give the monitor a pointer to a function that will disable "raw" mode on the console port.

For example, a call to moncom(CHARFUNC_PUTCHAR,app_putchar,0,0) will tell the monitor to use the function "app_putchar()" instead of its own putchar function for sending characters to the console port.

Application-Provided Mutual Exclusion

Now, the monitor claims to be independent of the application running on top of it, and potentially there will be a multi-tasking operating system as part of the application. How does the monitor keep its non-reentrant code from being reentered? This is where the second and third arguments to the monConnect() function come in. Like the other "application-provided" functions, the second and third arguments to monConnect() give the monitor a function to call for locking and unlocking the fact that some piece of application code is accessing a monitor facility. Each of the "mon_" functions is wrapped with a call to the lock() and unlock() functions that are provided to the monitor by monConnect() (refer to mon_putchar() as an example). Ideally then, the functions handed to monConnect() for this purpose, should provide priority inversion protection; otherwise, you must be aware of that (as you would any other use of a semaphore).

Application-Monitor Hookup: a small, single-threaded example

The following example shows the code that is needed to establish a basic hookup between a small, single-threaded application and the monitor. The code is compiled/linked with start() being the entrypoint and runs off the stack of the monitor. Note the USE_EXIT definition. Refer to the discussion on script nesting for details on this.

```
#include "monlib.h"

int
```

MicroMonitor
Application Programmer's Interface

```

main(int argc, char *argv)
{
    mon_printf("Hey, application %s is running!\n",argv[0]);
    mon_printf("Here are the args...\n");
    for (i=1;i<argc;i++)
        mon_printf("arg[%d] = %s\n",argv[i]);

    return(0);
}

void
start()
{
    char    **argv;
    int argc;
    extern uchar bss_start, bss_end;
    register uchar *ramstart;

    /* Clear out .bss space: */
    /* (not really necessary, done already by the TFS loader) */
    ramstart = &bss_start;
    while(ramstart < &bss_end)
        *ramstart++ = 0;

    /* Connect the application to the monitor. This MUST be done */
    /* prior to the application making any other attempts to use the */
    /* "mon_" functions provided by the monitor. Note that the value */
    /* of the first argument is the value returned by mstat (moncom */
    /* pointer) in the monitor. */

    monConnect((int(*)())*(unsigned long *)0x20,(void *)0,(void *)0);

    /* Extract argc/argv from structure and call main(): */
    mon_getargv(&argc,&argv);

#ifdef USE_EXIT
    /* Call main, then re-enter the monitor. */
    mon_appexit(main(argc,argv));
#else
    /* Call main, then return to monitor. */
    return(main(argc,argv));
#endif
}

```

Application-Monitor Hookup: a VxWorks example

The following example shows the code that is needed to establish a connection between a VxWorks application and the monitor. The majority of this code would be inserted into the usrConfig.c file of the BSP.

First of all, VxWorks uses the BOOT_LINE_ADRS definition to point to some location in memory that it can use to configure itself. The address is assumed to contain a string that is parsed during the early stages of VxWorks startup. This address should be set to some location in RAM space that is not used by the monitor or the application. Typically, the application is configured to be loaded somewhere above the value of \$APPRAMBASE in the monitor. The BOOT_LINE_ADRS address should be set to some block of space between \$APPRAMBASE and the actual starting point of the application. For the sake of this example, we will assume \$APPRAMBASE is 0x10000 and the starting point of the VxWorks application is 0x18000. We will then build the VxWorks application with BOOT_LINE_ADRS set to 0x17000 and have the following added to our monrc file (obviously each board's details will be different, this is just an example)...

```
# Basic set of bootup shell variables:
```

MicroMonitor
Application Programmer's Interface

```

set ETHERADD 00:60:1d:02:0b:fe
set IPADD 135.3.94.136
set HIPADD 135.3.94.76
set GIPADD 135.3.94.1
set NETMASK 255.255.255.0

# Build the VxWorks BOOT_LINE based on the variables established above...
# The "pm -s" command builds a string starting at the specified address
# The "pm -S" command concatenates a string to the end of the string that
# starts at the specified address
set BOOT_LINE_ADRS 0x17000
pm -s $BOOT_LINE_ADRS "cpm(0,0)"
pm -S $BOOT_LINE_ADRS " e=" $IPADD
pm -S $BOOT_LINE_ADRS " h=" $HIPADD
pm -S $BOOT_LINE_ADRS " g=" $GIPADD
pm -S $BOOT_LINE_ADRS " tn=target u=anonymous pw=vxworks"

```

The result of these entries in the monrc file is that the location 0x17000 will contain the following string...

```
cpm(0,0) e=135.3.94.136 h=135.3.94.76 g=135.3.94.1 tn=target u=anonymous pw=vxworks
```

Notice that the content of the BOOT_LINE_ADRS string is dependent on the content of the shell variables established above.

In the BSP (probably in usrConfig.c), the following code makes some of the initial connections. Typically this first call to monConnect() is done near the top of usrInit() just after the .bss space is initialized (usually this is a call to bzero() in usrInit()). It's important that monConnect be called AFTER this point because the initialization of the .bss space will undo some of the stuff set up in monConnect()...

```

#include "monlib.h"

/* Connect the application to the monitor. This must be done          */
/* prior to the application making any other attempts to use the "mon_" */
/* functions provided by the monitor. The value of 0x80000010 below is the */
/* "moncom pointer". This can be retrieved from the monitor with the      */
/* "mstat" command. */
monConnect((int(*)())(*((unsigned long *)0x80000010),(void *)0,(void *)0);

/* Tell uMON to use a few of this application's functions... */
mon_com(CHARFUNC_PUTCHAR,iasPutchar,0,0);
mon_com(CHARFUNC_GETCHAR,iasGetchar,0,0);
mon_com(CHARFUNC_GOTACHAR,iasGotachar,0,0);
mon_com(CHARFUNC_RAWMODEON,iasRawon,0,0);
mon_com(CHARFUNC_RAWMODEOFF,iasRawoff,0,0);
mon_com(CACHEFTYPE_DFLUSH,iasDcacheFlush,0,0);
mon_com(CACHEFTYPE_IINVALIDATE,iasIcacheInvalidate,0,0);

```

Following are the functions that are referenced by the above calls...

```

void
iasRawoff( void )
{
    ioctl(0,FIOSETOPTIONS,OPT_CRMOD | OPT_TANDEM | OPT_7_BIT);
}

int
iasGetchar( void )
{
    char    onechar;

    read(0,&onechar,1);
}

```

MicroMonitor
Application Programmer's Interface

```

        return((int)onechar);
    }

    int
    iasPuchar(char onechar)
    {
        return(write(1,&onechar,1));
    }

    int
    iasGotachar(void)
    {
        int avail;

        avail = 0;
        if (ioctl(0,FIONREAD,(int)&avail) != ERROR) {
            if (avail)
                return(1);
        }
        return(0);
    }

    int
    iasDcacheFlush(char *addr, int size)
    {
        return(cacheFlush(DATA_CACHE,addr,size));
    }

    int
    iasIcacheInvalidate(char *addr, int size)
    {
        return(cacheInvalidate(INSTRUCTION_CACHE,addr,size));
    }

```

Once the application's ROOT task has started, and prior to starting up any other tasks, monConnect() should be called a second time to establish the mutual exclusion protection mentioned above...

```

/*****
/* Call monConnect a second time (it was already called in
/* usrConfig.c). This time, pass in the lock/unlock arguments.
/* Note that this MUST be done prior to starting up any other tasks
/* that may use the monitor.
/* Note also, that if the semMCreate fails, there is no need to
/* call monConnect again, since it was called in usrConfig.c.
*****/
monSemId = semMCreate(SEM_INVERSION_SAFE | SEM_Q_PRIORITY);
if (!monSemId)
    printf("Could not create monitor access semaphore.\n");
else
    monConnect(0,monLock,monUnlock);

```

Following are the functions that are referenced by the above calls...

```

/* monLock() & monUnlock():
* Pointers to these functions are passed to monConnect() so that the
* monitor is guaranteed not to have any re-entrancy problems.
*/
SEM_ID monSemId;

void
monLock()
{
    semTake(monSemId, WAIT_FOREVER);

```

MicroMonitor Application Programmer's Interface

```

}

void
monUnlock()
{
    semGive(monSemId);
}

```

Finally, if the application has a command interpreter and wants to be able to hook its command interpreter to the script runner in TFS, do the following..

```

/*****/
/* Tell TFS to use this application's command interpreter when */
/* executing scripts. */
/*****/
err = mon_tfsctrl(TFS_DOCOMMAND, (long)docommand, 0);
if (err != TFS_OKAY)
    printf("TFS docommand reassign failed: ", mon_tfsctrl(TFS_ERRMSG, err, 0));

```

Use of Application-Provided Lockout for the Monitor API

The purpose of this section is to raise the user's awareness of the issues around the use of the monitor's API in a multi-tasking environment. The monitor itself is a single threaded program. There are no interrupts hence, aside from recursion, the code within the monitor does not have to deal with the issue of reentrancy or sharing of resources. However, when the monitor API is used in a multi-tasking environment things get more complicated. Since multiple tasks can access the monitor API, without some type of lock out mechanism, it is very likely that two different tasks may try to use a facility in the monitor that is not prepared for multi-task access.

The primary mechanism to deal with this is to wrap each monitor API function with an application-provided lock/unlock function that will treat the monitor API as a resource and provide mutual exclusion for that resource. For example, we never want to be inside the `tfsadd()` function more than once. It is very important that this function be protected from reentrancy because it accesses the underlying flash and while the flash is being modified, we don't want to be interrupted by another function that will also attempt to modify the flash. The function in application space that hooks to the monitor's `tfsadd()` function is called `mon_tfsadd()`. The code for `mon_tfsadd()` follows...

```
int
mon_tfsadd(char *name, char *info, char *flags, unsigned char *src, int size)
{
    int ret;

    TFS_MONLOCK();
    ret = _tfsadd(name, info, flags, src, size);
    TFS_MONUNLOCK();
    return(ret);
}
```

All of the monitor API functions look very similar to this one. The `_tfsadd()` function pointer and the code within `monLock()` and `monUnlock()` are all established by the call to `monConnect()` when the application first hooks itself up to the monitor. Assuming the application has established some lockout mechanism, this guarantees that while `tfsadd()` is being executed, no other monitor API function will run. As a general solution, this works ok. It is left up to the application to pick the best solution for the lockout (semaphore, mutex, scheduler disable, critical section, etc...). The underlying hardware platform, the monitor API facilities being used and the way they are used by the application are the major factors used to determine which mechanism fits best. The right choice will vary, and it is possible that there will be no need for a lockout mechanism at all.

While this solution is generally applicable, it does have some limitations and raise some questions...

- **All of the monitor facilities use one lockout mechanism, despite the fact that they do not all have reentrancy conflicts.**

For example...The monitor facilities used by `getenv()` do not interfere with those of `tfsadd()` so it seems like I should be able to call `mon_getenv()` even if `mon_tfsadd()` is in progress.

- **If one task calls a monitor API function that blocks, then no other task can access a monitor facility until that first API call completes.**

If I have one task in my application that calls `mon_getline()`, then no other task will be able to access a monitor API function until `mon_getline()` returns. Since `mon_getline()` blocks waiting for a full line of console input, this will be very inefficient.

- **If I am inside an monitor API function, I can't call another monitor API function because the first call will lock out the second one.**

If I use `mon_docommand()` in my application's CLI, plus I've previously installed some application-specific commands into the monitor's command table using `mon_addcommand()`, and within the code of one of my application-specific commands I call `mon_printf()` it will lock up because the call to

mon_docommand() earlier in the function nest has locked out any further API calls until the return from mon_docommand().

These are all valid concerns, and depending on the situation, they may or may not be an issue for your project. The remaining discussion will attempt to solve the above problems, or at least make it clear to the reader just what the problem is.

First of all, the major "single user" resource in the monitor is the flash device. At first glance, this accurately implies that protection is needed in a multi-tasking environment so that two tasks never attempt to call tfsadd(). This is certainly a valid concern; however, a more subtle issue is the fact that the monitor itself may be executing directly out of the same flash that is used by TFS. Now we have to deal with the fact that while tfsadd() is modifying flash, we do not want ANY OTHER monitor API function to run because it would fetch instructions from the same flash device that has an operation (erase or write) in progress. In generic terms, this is illegal for flash devices. Typically, if a flash operation is in progress, fetching from that same device will fail. This is only a problem if the monitor is fetching instructions from the same device that TFS uses for file storage. There are a few solutions to this, some of which simply require coding discipline and others which require a hardware modification. Certain solutions can make the lock/unlock restriction much more relaxed...

- Limit the monitor API function accesses to one task. This entirely eliminates the need for the lockout because as far as the monitor is concerned, it is only being accessed by a single thread.
- Do not use any of the blocking monitor API functions (mon_getchar(), mon_getline() and mon_getbytes()). This eliminates the chance that some other task will be indefinitely locked out because one of these functions is blocked and holding the lock.
- Configure the monitor so that it runs out of RAM. If the monitor is configured to copy itself into ram at startup and fetch instructions only from ram, then operations on flash will not conflict with execution of the monitor in ram space. In this situation, most of the non-flash related monitor API functions can run without the lock.
- Separate the boot flash device from the TFS storage device. The boot flash device may not be the same device that is to be used for TFS file storage. In this case, even if the monitor is fetching from the boot flash, the flash operations (erase and/or write) are being done on some other device; hence no conflict.
- Use a flash device that supports simultaneous execution and operation. Typically, these devices are architected such that the device is essentially in two halves. One half can be executed from while the other half is being operated on. Then the only issue is to make sure that the monitor executable is in one half and the TFS storage space is in the other half.
- Use a lockout mechanism that disables context switches. If this is done, then only the monitor API functions that deal with flash access need to be wrapped with the lock functions. This is because once the lockout is enabled, no other monitor API function will be callable until that one completes. An extension to this would be to make the lockout function aware of itself, so that it deals with nesting. Following is a generic example of a lockout function that would accomplish this (code contributed by Jim Apgar)...

```
static OSPRIORITY save_pri;
static U32 umon_nest_level = 0;

static void
umon_lock (void)
{
    ENTER_CRITICAL_SECTION();
    if (umon_nest_level == 0) {
        // Raise priority to prevent other processes
        // from interrupting monitor operations
        save_pri = GET_CURRENT_TASK_PRIORITY();
        SET_CURRENT_TASK_PRIORITY(HIGHEST_PRIORITY);
    }
}
```

MicroMonitor Application Programmer's Interface

```

    }
    umon_nest_level++;
    EXIT_CRITICAL_SECTION();
}

static void
umon_unlock(void)
{
    ENTER_CRITICAL_SECTION();
    if (umon_nest_level > 0) {
        umon_nest_level--;
    }
    if (umon_nest_level == 0) {
        // Restore original priority
        SET_CURRENT_TASK_PRIORITY( save_pri );
    }
    EXIT_CRITICAL_SECTION();
}
}

```

There are probably other solutions, but as you can see from the list above, there are at least several choices. Each one has different pros and cons, so its hard to say which is best. The most important thing is to be aware of your situation so you can make the right choice. As mentioned at the top of this appnote, the default means of lockout is to assign a single application-specific facility to all monitor API functions. Depending on your solution, you may be able to eliminate the need for all lockouts that are not related to flash. Or, you may find that you need to establish more than one lock out mechanism... one for flash access, one for environment access, etc... In monlib.c, there is a set of macros at the top of the file that establish the default monLock/monUnlock facility. Following is the code and comments taken directly from that sourc file...

```

/*****
*
* The following macros support the default monitor lock/unlock mechanism when
* they point to monLock and monUnlock. If something other than the default
* is to be used, then simply redefine them here. Refer to the monitor
* app note that discusses multi-tasking access to the monitor API for more
* information.
*
* TFS_MONLOCK/UNLOCK:
* Lock/unlock for functions that access TFS flash space:
*/
#define TFS_MONLOCK    monLock
#define TFS_MONUNLOCK  monUnlock

/* ENV_MONLOCK/UNLOCK:
* Lock/unlock for functions that access monitor shell variables:
*/
#define ENV_MONLOCK    monLock
#define ENV_MONUNLOCK  monUnlock

/* CONSOLE_MONLOCK/UNLOCK:
* Lock/unlock for functions in the monitor that deal with console output.
*/
#define CONSOLE_MONLOCK  monLock
#define CONSOLE_MONUNLOCK monUnlock

/* HEAP_MONLOCK/UNLOCK:
* Lock/unlock for functions in the monitor that deal with the heap.
*/
#define HEAP_MONLOCK    monLock
#define HEAP_MONUNLOCK  monUnlock

/* BLOCKING_MONLOCK/UNLOCK:

```

MicroMonitor Application Programmer's Interface

```
* Lock/unlock for functions in the monitor that block waiting for
* console input.
*/
#define BLOCKING_MONLOCK monLock
#define BLOCKING_MONUNLOCK monUnlock

/* GENERIC_MONLOCK/UNLOCK:
* Lock/unlock for all functions not covered by the above macros.
*/
#define GENERIC_MONLOCK monLock
#define GENERIC_MONUNLOCK monUnlock
```

Since monlib.c is actually part of the application, it is 100% legal to make application specific adjustments to this file. These macros should make those adjustments a bit less painful.

Acknowledgements:

Most of this section is a result of discussion with Jim Apgar, a monitor user, who ran into some issues that caused him to deviate from the standard use of the monitor API lockout facilities. Thanks Jim!

Application Installation on a Monitor Based System

The above discussions assume that the application is resident on the target system. There are several different ways the application can be installed on a MicroMonitor based system. How it is transferred and where in the system it is transferred to depends on the project. Here are a few common ways it is done with MicroMonitor...

Binary Image Copied to RAM:

In the simplest (least versatile) case, an application is built to reside at some specified location in memory. Through the compile/link/build process, a file is created that represents the image of the application as it would exist starting at some specified location. That file can be transferred to memory on the target system using Xmodem or TFTP and then the monitor can use the *call* command to branch into the entrypoint of the application.

Binary Image Copied to Raw Flash:

As an alternative to the above approach, the same image could be transferred to flash on the target and the monitor could be used to transfer the image in flash to some pre-defined location in RAM, then once again, the *call* command would be used to branch to the image's entrypoint.

Binary Image Copied to TFS:

The two above options do not require TFS (Tiny File System: MicroMonitor's flash file system). If TFS is included, then the image could simply be copied to a file in TFS as a raw image; then transferred to RAM and executed.

Formatted File Copied to TFS and Loaded:

The most common way the application is stored on-target is to transfer the actual formatted file (usually ELF, COFF or AOUT, but there are others). TFS understands ELF, so it can then be used to extract the memory map information from the file and transfer the various sections (.text, .data, .bss) into RAM space. The formatted file also contains information about the entrypoint and TFS extracts that and automatically turns over control to the image at that point.

Network Boot:

The monitor supports DHCP/BOOTP/TFTP, so all of the above options could be run over an automatic network boot environment if needed.

Extending the Monitor's Heap

The monitor has its own memory manager (*malloc.c* & *sbrk.c*). By default, the monitor is typically configured to have 8K of memory dedicated to the monitor's *malloc*. There are cases where this may be insufficient and the monitor configuration can be modified so that a larger amount of memory is statically allocated for this purpose. An alternative to this is to extend the space using the *-X* option of the heap command. This allows the monitor's default amount of memory to remain relatively small, but does not prohibit the ability to increase the size if necessary.

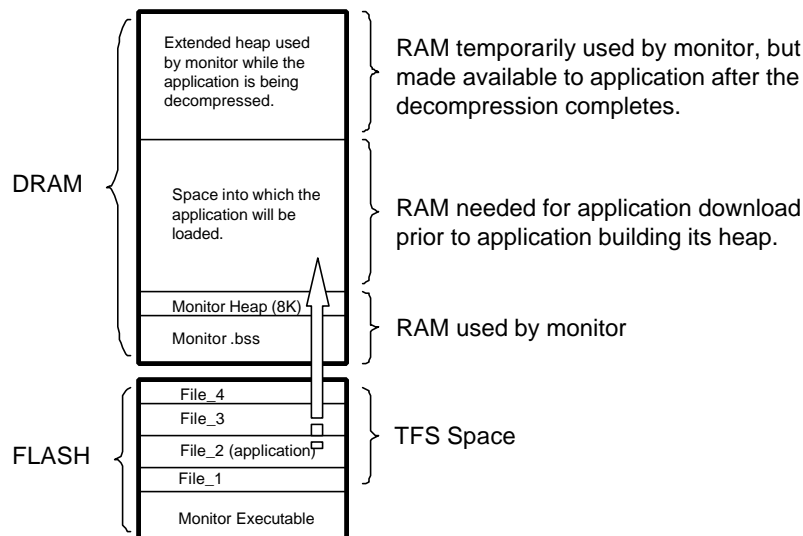
Usually, a project will put some operating system on top of the monitor and use that OS's *malloc*; however, the monitor code itself still maintains its own small heap independent of the OS. The monitor's heap is used mostly for shell variable storage and a few operations in TFS, which generally don't need much memory. Larger allocations can be made if the application does use the monitor's *malloc* and also if the ZLIB code is active for decompression. Following are two examples of situations where heap *-X* would be used to compensate for the small amount of memory configured for use by the monitor's *malloc*, but still not require that the monitor have a large heap compiled in. Note that an alternative to the "heap *-X*" command at the monitor's command line is to use the API call *mon_heapextend()*.

An Application that uses the monitor's malloc:

Usually, an application has its own malloc and when it takes over the target, the monitor's malloc is only used by the monitor. This doesn't have to be the case. A small application that does not need an OS may still need some way to allocate memory temporarily, so `mon_malloc()` can be used for this. This being the case, then it is likely that the 8K of space compiled in as the monitor's heap will not be sufficient. This is where `heap -X {start,size}` comes in... In the `monrc` file (or anywhere prior to the application starting to allocate memory beyond the basic 8K), set up these two parameters. The value of `start` should be loaded with a starting address from which malloc will get additional memory and `size` is the size of that block of memory. Once this is done, the monitor's malloc will be able to allocated memory from that block of space if it cannot fulfill a request using its initial 8K of space.

Using ZLIB to decompress files in TFS:

The use of zlib to decompress is likely to need more than the basic 8K of memory that the monitor's heap is configured with. Like the above example, the `heapsize` can be increased to allow zlib to do its thing. This is good, but once zlib has completed the decompression of the application, it is very likely that heap extension will no longer be needed by the monitor. To compensate for this, the extension can be "de-allocated" from the monitor as long as there is no memory currently allocated in the space that spans the extension. For example, the heap can be extended, the executable decompressed into DRAM, then the heap can be "un-extended" and control can then be turned over to the application. The only obvious requirement is that the space into which the application is being loaded can not overlap with the extended heap space. Following is a pictorial view of the extended heap space and a script to extend the heap, load a compressed executable, "unextend" the heap and run the executable...



Example of Memory Space Allocations for Unzipping an Application to DRAM
(sizes implied by divisions of memory space should be ignored)

```
# Extend the heap as necessary:
heap -X 0x1100000,0x80000

# Load the application, allowing zlib to use the extended
# heap. (this will load ENTRYPOINT with the entry point of
# the application 'File_2'):

```

MicroMonitor
Application Programmer's Interface

```
tfs ld File_2

# Eliminate the heap extension:
heap -x

# Transfer control to the entry point of
# the application previously loaded:
call $ENTRYPOINT
```

MicroMonitor API Function Descriptions:

mon_addcommand()

API FUNCTION:

mon_addcommand()

SYNOPSIS:

Add an application-specific command list to the monitor.

PROTOTYPE:

```
int mon_addcommand(struct monCommand *cmd);
```

DESCRIPTION:

This command is typically used in an application that is being debugged and breakpoints are being set. When a breakpoint is hit, control is transferred from the application to the monitor. Sometimes it is convenient to be at the monitor prompt, but still have some application-specific commands to run. This provides that functionality.

Parameters:

- struct monCommand *cmd;
pointer to a table of command structures that is used by the monitor's command interpreter. The monitor assumes that the final element in the table has a NULL name member.

Return:

0 if successful; else -1.

SOURCE FILE CONTAINING CODE:

docmd.c

mon_appexit()

API FUNCTION:

mon_appexit()

SYNOPSIS:

Allows the application return control to the monitor after exiting.

PROTOTYPE:

```
void mon_appexit(int exitval);
```

DESCRIPTION:

An alternative to exit(). This simply returns control to the monitor and the monitor will print out the exit status passed to it.

Parameters:

- int exitval
The value considered to be the exit status.

Return:

Void.

SOURCE FILE CONTAINING CODE:

main.c

mon_com()

API FUNCTION:

mon_com()

SYNOPSIS:

This function is the lowest-level hook provided by the monitor so that the application can connect.

PROTOTYPE:

```
int mon_com(int cmd, long arg1, long arg2, long arg3);
```

DESCRIPTION:

This function provides the basic interface needed for hookup between application and monitor. Refer to moncom.c and monlib.c for details.

Parameters:

- int cmd
Operation to be performed.
- arg{1-3}
Operation-specific arguments.

Return:

Void.

SOURCE FILE CONTAINING CODE:

moncom.c

mon_cprintf()

API FUNCTION:

mon_cprintf()

SYNOPSIS:

A centered version of mon_printf().

PROTOTYPE:

```
int mon_cprintf(char *format, arg1, arg2, ...);
```

DESCRIPTION:

Provides the application with a small and simple printf() with limited formatting capability. The text in the format is automatically centered across an 80-character line. Refer to sprintf() for formatting characters supported.

Parameters:

- char *format
pointer to a format buffer
- argN
arguments referred to (if any) by format buffer).

Return:

The size of the final string printed out the console port.

SOURCE FILE CONTAINING CODE:

mprintf.c

mon_crc16()

API FUNCTION:

mon_crc16()

SYNOPSIS:

CRC16 calculation.

PROTOTYPE:

unsigned short mon_crc16(char *buffer, long nbytes);

DESCRIPTION:

Perform a crc16 calculation across the specified buffer.

Parameters:

- char *buffer
data space over which the crc16 calculation is to be performed.
- long nbytes
size of the buffer..

Return:

crc.

SOURCE FILE CONTAINING CODE:

misc.c

mon_crc32()

API FUNCTION:

mon_crc32()

SYNOPSIS:

CRC32 calculation.

PROTOTYPE:

unsigned long mon_crc16(char *buffer, long nbytes);

DESCRIPTION:

Perform a crc32 calculation across the specified buffer.

Parameters:

- char *buffer
data space over which the crc16 calculation is to be performed.
- long nbytes
size of the buffer..

Return:

crc.

SOURCE FILE CONTAINING CODE:

misc.c

mon_decompress()

API FUNCTION:

`mon_decompress()`

SYNOPSIS:

Decompress a block of data that has been previously compressed.

PROTOTYPE:

```
int mon_decompress(char *src, int srcsize, char *dest);
```

DESCRIPTION:

Allows the application to decompress memory (or a data file) that was previously compressed with gzip.

Parameters:

- `char *src`
pointer to starting location of compressed data.
- `int srcsize`
size of the block of compressed data in bytes.
- `char *dest`
point to location into which the decompressed data is to be placed.

Return:

The size of the decompressed data or -1 if failure.

SOURCE FILE CONTAINING CODE:

`zlib/gzio.c`

mon_docommand()

API FUNCTION:

mon_docommand()

SYNOPSIS:

A mechanism by which the application can invoke a command that is part of the monitor.

PROTOTYPE:

```
int mon_docommand(char *cmd_string, int verbose);
```

DESCRIPTION:

This command allows the application code to execute commands that are normally executable from the console port of the monitor user interface. This allows, for example, an application to have a console interface with its own set of commands; then if the command entered does not match any of those commands in the application's command table, the command string can be passed to the monitor to see if the command is actually a monitor command. The end result is that the application code can inherit the command table of the monitor with almost zero overhead.

Parameters:

- char *cmd_string;
string of characters that would be typed as if they were a command entered at the console interface.
- int verbosity
this can be 0, 1 or 2 to provide different levels of verbosity.

Return:

- CMD_SUCCESS:
Everything worked ok.
- CMD_FAILURE:
Command parameters were valid, but command itself failed for some other reason.
- CMD_LINE_ERROR:
Command line itself was invalid. Too many args, invalid shell var syntax, etc.. Somekind of command line error prior to checking for the command name-to-function match.
- CMD_ULVL_DENIED:
Command's user level is higher than current user level, so access is denied.
- CMD_PARAM_ERROR:
Command line did not parse properly. There was a syntax error on the command line that did not even allow the command function to get going.
- CMD_NOT_FOUND:
Since these same return values are used for each command function plus the docommand() function, this error indicates that docommand() could not even find the command in the command table.

SOURCE FILE CONTAINING CODE:

docmd.c

mon_free()

API FUNCTION:

mon_free()

SYNOPSIS:

Similar to standard free().

PROTOTYPE:

```
void mon_free(char *buf);
```

DESCRIPTION:

Partner to mon_malloc(). Used to deallocate memory that was previously allocated by mon_malloc(). If the pointer passed to mon_free() is not a pointer that was previously returned by mon_malloc() an error will result. Refer to malloc() for other details.

Parameters:

- char * buffer
pointer to a buffer that was previously allocated by mon_malloc().

Return:

Void.

SOURCE FILE CONTAINING CODE:

malloc.c

mon_getargv()

API FUNCTION:

mon_getargv()

SYNOPSIS:

Retrieve an argument list.

PROTOTYPE:

```
void mon_getargv(int *argc, char ***argv);
```

DESCRIPTION:

This function provides the hook needed by the application to retrieve an argument list that was created previously by either the argv command or by direct command line invocation. The integer pointed to by argc is loaded with the argument count and a the pointer to a character array is loaded with the location of the char *argv[] table.

Parameters:

- int *argc;
pointer to an integer that is to be loaded with the number of arguments.
- char ***argv
pointer to an array of string pointers that is loaded with the char *argv[] table.

Return:

Void.

SOURCE FILE CONTAINING CODE:

go.c

mon_getbytes()

API FUNCTION:

mon_getbytes()

SYNOPSIS:

Retrieve some number of bytes from the console port.

PROTOTYPE:

```
int mon_getbytes(char *buf, int count, int block);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still read in characters from the console port.

Parameters:

- char *buf
pointer to a buffer into which mon_getbytes() will place the characters
- int count
number of characters to retrieve.
- int block
if set, then wait for 'count' characters; else return as soon as there are no more characters present.

Return:

The same character that was passed to the function is returned.

SOURCE FILE CONTAINING CODE:

chario.c

mon_getchar()

API FUNCTION:

`mon_getchar()`

SYNOPSIS:

Provide similar functionality as standard `getchar()`.

PROTOTYPE:

`int mon_getchar(void);`

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still receive characters from the console port. This function will block waiting for the next incoming character into the console port. If one is already there prior to the call, then the return is immediate.

Return:

The character read on the console port.

SOURCE FILE CONTAINING CODE:

`cpuio.c`

mon_getenv()

API FUNCTION:

mon_getenv()

SYNOPSIS:

Similar to standard getenv().

PROTOTYPE:

```
char *mon_getenv(char *varname);
```

DESCRIPTION:

The monitor can establish shell variables at the command line using the set command or by another application using the setenv(). This function allows an application to retrieve the value that corresponds to a specified shell variable name.

Parameters:

- char *varname
the name of the shell variable of which to retrieve the content.

Return:

The content of the shell variable specified by varname, or (char *)NULL if the variable does not exist.

SOURCE FILE CONTAINING CODE:

env.c

mon_getline()

API FUNCTION:

mon_getline()

SYNOPSIS:

Retrieve characters from console until CR or LF is received.

PROTOTYPE:

```
int mon_getline(char *buffer, int max,int ledit);
```

DESCRIPTION:

Retrieve characters from the console port until a carriage return or line feed is received (or 'max' characters are received).

Parameters:

- char *buffer
a pointer to the space into which the incoming characters are to be placed.
- int max
the max number of bytes to place into the buffer.
- int ledit
if set to 1, then allow the use of the monitor's line-editing functions also.

Return:

The number of bytes retrieved.

SOURCE FILE CONTAINING CODE:

chario.c

mon_gotachar()

API FUNCTION:

mon_gotachar()

SYNOPSIS:

Return status of character presence on console port.

PROTOTYPE:

```
int mon_gotachar(void);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still query for the presence of a character on the console.

Return:

1 if there is a character present; else 0.

SOURCE FILE CONTAINING CODE:

cpuio.c

mon_intsoff()

API FUNCTION:

mon_intsoff()

SYNOPSIS:

Allows the application to turn off interrupts.

PROTOTYPE:

unsigned long mon_intsoff(void);

DESCRIPTION:

Allows the application to turn off interrupts. Note that this is probably not useful if you are running with some vendor-supplied OS. Use the function that is supplied with the package.

Return:

The value returned can be used as an argument to intstore() to re-establish previous interrupt state.

SOURCE FILE CONTAINING CODE:

main.c

mon_intsrestore()

API FUNCTION:

mon_intsrestore()

SYNOPSIS:

Allows the application to restore interrupt state.

PROTOTYPE:

```
void mon_intsrestore(ulong ival);
```

DESCRIPTION:

Allows the application to restore interrupts. Note that this is probably not useful if you are running with some vendor-supplied OS. Use the function that is supplied with the package.

Parameters:

- unsigned long ival
CPU-specific value that was previously returned by intsoff().

Return:

Void.

SOURCE FILE CONTAINING CODE:

main.c

mon_malloc()

API FUNCTION:

mon_malloc()

SYNOPSIS:

Similar to standard malloc().

PROTOTYPE:

```
char *mon_malloc(int size);
```

DESCRIPTION:

This provides the application with a malloc() whose heap is maintained by the monitor. In addition, the heap is queriable through the heap command. At each allocation/deallocation, the entire control structure of the heap is checked for sanity. Also, in addition to control verification, overrun and underrun checks are made because the space returned by malloc also has additional wrapper space that is checked to verify that the space is not used beyond (or prior to) its limit. This implies overhead which may or may not be desired, but does provide a fairly strong amount of debugging.

The monitor is built with an amount of heap space that is needed for the monitor itself plus a bit extra for application space. To allow additional memory allocations to be made, the heap command allows the user to define a starting point and size of the additional space. Refer to the discussion on extending the monitor's heap for more information on this.

If interested, refer to text in the specified files below for more details on this malloc() implementation.

Parameters:

- int size
the size of the block of memory to be allocated.

Return:

A pointer to the new block of memory or (char *)NULL if error.

SOURCE FILE CONTAINING CODE:

malloc.c & sbrk.c

mon_pioclr()

API FUNCTION:

mon_pioclr()

SYNOPSIS:

Provide a mechanism for the application to reset (make 0) status of CPU-specific parallel IO lines.

PROTOTYPE:

```
void mon_pioclr(char portname, int bitnum);
```

DESCRIPTION:

This function somewhat CPU specific. It provides access to the CPU's parallel IO pins (if any).

Parameters:

- char portname
single-character representing the pio port to be accessed (cpu-specific).
- int bitnum
the bit within the specified port to be cleared or reset.

Return:

Void

SOURCE FILE CONTAINING CODE:

pio.c

mon_pioget()

API FUNCTION:

mon_pioget()

SYNOPSIS:

Provide a mechanism for the application to retrieve status of CPU-specific parallel IO lines.

PROTOTYPE:

```
int mon_pioget(char portname, int bitnum);
```

DESCRIPTION:

This function somewhat CPU specific. It provides access to the CPU's parallel IO pins (if any).

Parameters:

- char portname
single-character representing the pio port to be accessed (cpu-specific).
- int bitnum
the bit within the specified port to be checked.

Return:

If successful, return 1 or 0 depending on the status of the PIO line; else return -1.

SOURCE FILE CONTAINING CODE:

pio.c

mon_pioreset()

API FUNCTION:

mon_pioreset()

SYNOPSIS:

Provide a mechanism for the application to set (make 1) status of CPU-specific parallel IO lines.

PROTOTYPE:

```
void mon_pioreset(char portname, int bitnum);
```

DESCRIPTION:

This function is CPU specific. It provides access to the CPU's parallel IO pins (if any).

Parameters:

- char portname
single-character representing the pio port to be accessed (cpu-specific).
- int bitnum
the bit within the specified port to be set.

Return:

Void

SOURCE FILE CONTAINING CODE:

pio.c

mon_printf()

API FUNCTION:

mon_printf()

SYNOPSIS:

Similar to printf() but limited in the formatting capability.

PROTOTYPE:

```
int mon_printf(char *format, arg1, arg2, ...);
```

DESCRIPTION:

Provides the application with a small and simple printf() with limited formatting capability. Refer to sprintf() for formatting characters supported.

Parameters:

- char *format
pointer to a format buffer
- argN
arguments referred to (if any) by format buffer).

Return:

The size of the final string printed out the console port.

SOURCE FILE CONTAINING CODE:

mprintf.c

mon_putchar()

API FUNCTION:

mon_putchar()

SYNOPSIS:

Provide similar functionality as standard putchar().

PROTOTYPE:

```
int mon_putchar(char c);
```

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still transfer characters to the console port.

Parameters:

- char c
the character destined for the console port of the target.

Return:

The same character that was passed to the function is returned.

SOURCE FILE CONTAINING CODE:

chario.c

mon_restart()

API FUNCTION:

mon_restart()

SYNOPSIS:

Allows the application to re-start the monitor.

PROTOTYPE:

```
void mon_restart(int val);
```

DESCRIPTION:

Allows the application to re-enter the monitor at various points.

Parameters:

- int val
restart value

Return:

This function does not return.

SOURCE FILE CONTAINING CODE:

main.c

mon_setenv()

API FUNCTION:

mon_setenv()

SYNOPSIS:

Similar to standard setenv().

PROTOTYPE:

```
void mon_setenv(char *varname, char *value);
```

DESCRIPTION:

The monitor can establish shell variables at the command line using the set command. This function allows an application to also establish a shell variable.

Parameters:

- char *varname
the name of the shell variable to be created.
- char *value
the value that the shell variable represents. If this pointer is NULL, then the shell variable with the name *varname* is removed from the environment.

SOURCE FILE CONTAINING CODE:

env.c

mon_setUserLevel()

API FUNCTION:

mon_setUserLevel()

SYNOPSIS:

Modify or retrieve the current user level of the monitor.

PROTOTYPE:

```
int mon_setUserLevel(int level, char *password);
```

DESCRIPTION:

Allows the application to query or modify the monitor's user level. At this API level, the password can be ignored. An incoming password of (char *)NULL tells this function not to check the password, but to simply adjust the user level. This allows the application to decide whether or not it actually wants to use the password protection of user levels in the monitor.

Parameters:

- int level
value to be used as the new level. If -1, then simply return the current level. The current valid range for user levels in the monitor is 0-3.
- char *password
the string that corresponds to the password needed to get to the specified user level. Note that (char *)NULL tells this function to ignore the password entry.

Return:

The current (if level = -1) or old user level.

SOURCE FILE CONTAINING CODE:

misccmds.c

mon_sprintf()

API FUNCTION:

mon_sprintf()

SYNOPSIS:

Similar to sprintf() but limited in the formatting capability.

PROTOTYPE:

```
int mon_sprintf(char *buffer, char *format, arg1, arg2, ...);
```

DESCRIPTION:

Provides the application with a small and simple sprintf() with limited formatting capability. Both cprintf() and printf() call this function with a buffer of 128 bytes, so make sure you don't build a string that will exceed that. A maximum of 12 args can be passed into mon_sprintf (arg1 through arg12). The %s, %x, %c, and %d formats are supported to some degree.

Also, non-standard format conversions for MAC and IP address formats are supported... %l assumes the argument is a long and it is converted to a string in IP format (1.2.3.4); %M assumes the argument is a pointer to an array of 6 bytes and it is converted to a string in the MAC address format (xx:xx:xx:xx:xx:xx).

Parameters:

- char *buffer
buffer into which the format conversion is to be placed
- char *format
pointer to a format buffer
- argN
arguments referred to (if any) by format buffer).

Return:

The size of the final string printed out the console port.

SOURCE FILE CONTAINING CODE:

mprintf.c

mon_tfsadd()

API FUNCTION:

mon_tfsadd()

SYNOPSIS:

Add a new file to TFS.

PROTOTYPE:

```
int mon_tfsadd(char *name, char *info, char *flags, uchar *src, int size);
```

DESCRIPTION:

The application can use this function to create a file from some block of memory without having to go through a typical open, write, close scenario. If the file already exists, it will first see if the incoming data is identical to that of the file already in TFS; if it is, then no flash operation is performed and TFS_OKAY is returned. If there are differences, then the old file is first removed and the new file is added.

Parameters:

- char *name
name of the file being created.
- char *info
content to be placed in the info field of the header.
- char *flags
flags to be assigned to the file.
- uchar *src;
location of the data to become the file content.
- int size
size of the data copied to the file.

<blink>Warning:</blink>

Although it is allowed, it is dangerous to pass a source address into this function that is within TFS flash space. This would seem to be the intuitive thing to do to copy one file to another; however, if a defragmentation occurs as a result of this addition and the source file is shifted, then the add is aborted and the TFSERR_FLAKEYSOURCE error code is returned. Whenever copying from one file to another, it is best to copy the source file to local ram space then transfer the content of the ram to tfsadd(). Then, even if a defragmentation does occur, it will not affect anything in the source space.

Return:

TFS_OKAY if successful; else...
TFSERR_BADARG
TFSERR_CORRUPT
TFSERR_FILEEXISTS
TFSERR_FLASHFULL
TFSERR_FLASHFAILURE
TFSERR_BADCRC
TFSERR_FLAKEYSOURCE

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfsclose()

API FUNCTION:

mon_tfsclose()

SYNOPSIS:

Close a TFS file that had previously been opened.

PROTOTYPE:

```
int mon_tfsclose(int tfd, char *info);
```

DESCRIPTION:

When all interaction with a previously opened file has been completed, mon_tfsclose() must be called to release the file descriptor used with the opened file and possibly initiate a transfer of the data to flash (if the file was opened for some type of modification). The tfd argument is the value that was returned from the initial mon_tfsopen(), and the info argument is a string (optionally NULL) that is used to store the "info" field of the file header.

Parameters:

- int tfd
the same value that was returned when the initial mon_tfsopen() was called.
- char *info
a pointer to a string that is stored in the "info" field of the file header.

Return:

TFS_OKAY if successful, else...
TFSERR_BADARG
TFSERR_BADFD
TFSERR_FLASHFAILURE

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsctrl()

API FUNCTION:

mon_tfsctrl()

SYNOPSIS:

Perform some type of "control" operation on TFS or a file in TFS.

PROTOTYPE:

```
int mon_tfsctrl(int rqst, long arg1, long arg2);
```

DESCRIPTION:

Similar in purpose to a standard ioctl(), this function allows the user to perform some type of control function on TFS or a file in TFS.

Parameters:

- int rqst
type of control function to be performed.
- long arg1
depending on the value of rqst, this argument may or may not be used.
- long arg2
depending on the value of rqst this argument may or may not be used.

Valid rqst values:

- TFS_ERRMSG
Returns a pointer to a character string that corresponds to the verbose description of the error. The value in arg1 is some error value that was returned by some other tfs system call.
- TFS_MEMUSE
Returns the amount of flash memory that is currently being used by files in TFS. This includes space used by active and deleted files.
- TFS_MEMAVAIL
Returns the amount of flash memory that is still available for use by TFS.
- TFS_MEMDEAD
Returns the amount of flash memory that is currently being used by deleted files.
- TFS_DEFRAG
Run a TFS defragmentation, to remove any "dead" flash space taken up by deleted files. If arg1 is non-zero, then after defragmentation, the target is reset; the value of arg2 is considered the verbosity level to use during defragmentation.
- TFS_UNOPEN
If a TFS file was previously opened for creation or append, and for some reason, the need to create/modify the file no longer exists, this function essentially calls mon_tfsfclose() but does not make any modifications to the flash. The value of arg1 is the file descriptor returned by the initial call to mon_tfsopen().
- TFS_FATOB
This request converts a string of flag characters (see tfs attributes) to a binary value that is stored in the file header. Arg1 contains a pointer to the string of characters. The return value is a long that represents the binary value used by TFS internally. The return value is -1 if any character of the incoming string is invalid.
- TFS_FBTOA
Just the opposite of TFS_FATOB... It takes a binary value and converts it to an ascii string. Arg1 is the binary value, arg2 is a pointer to the buffer (should be at least 16 bytes) into which TFS stores the string.

- **TFS_TELL**
Return the offset into the file specified by arg1 which is the file descriptor returned by mon_tfsopen() sometime prior.
- **TFS_TIMEFUNCS**
Arg1 is a pointer to the getLtime() function and arg2 is a pointer to the getAtime() function. Refer to tfs time-of-day discussion. Returns TFS_OKAY if successful.
- **TFS_DOCOMMAND**
Arg1 is a pointer to the application's command interpreter function whose prototype is *void docommand(char * cmdline, int verbosity)*. If NULL, then the standard command interpreter is used. Arg2 is a pointer to a location into which the monitor will place the current command interpreter function being used by the script runner. If NULL, then this value is not loaded. Returns TFS_OKAY if successful.
- **TFS_DEFRAGDEV**
Arg1 is a char pointer to the name of the TFS device to be defragmented (cleaned). Returns TFS_OKAY if successful.
- **TFS_CHECKDEV**
Arg1 is a char pointer to the name of the TFS device to be checked. Returns TFS_OKAY if file system on the specified device is not found to be corrupt.
- **TFS_INITDEV**
Arg1 is a char pointer to the name of the TFS device to be initialized. Returns TFS_OKAY if successful.

If in the above descriptions, one or both of the 'arg' values is not mentioned, then assume it is not used.

Return:

The value returned depends on the 'rqst' value passed into the function.

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfseof()

API FUNCTION:

mon_tfseof()

SYNOPSIS:

Return EOF (end of file) status on specified file.

PROTOTYPE:

```
int mon_tfseof(int tfd);
```

DESCRIPTION:

Allows the application to check to see if a currently opened-for-read file has reached the end-of-file.

Parameters:

- int tfd
TFS file descriptor returned from a previous call to tfsopen().

Return:

1 if TFS's internal pointer has reached the end of the file; else 0.

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsfstat()

API FUNCTION:

mon_tfsfstat()

SYNOPSIS:

Populate a TFILE structure with the designated file's file header structure.

PROTOTYPE:

```
int mon_tfsfstat(char *filename, TFILE *tfsstruct);
```

DESCRIPTION:

Allows the application to retrieve a TFILE structure (struct tfs_hdr) attached to the specified file (if it exists). This API function is a replacement for tfsstat() in cases where the pointer returned by tfsstat() is used for more than just determining file existence.

Parameters:

- char *filename
name of file in TFS.
- TFILE *tfsstruct
pointer to a TFILE structure that tfsfstat will populate if the file exists.

Return:

Return 0 if the file exists; else -1.

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfsgetline()

API FUNCTION:

mon_tfsgetline()

SYNOPSIS:

Retrieve the next line from an assumed ascii file.

PROTOTYPE:

```
int mon_tfsgetline(int tfd, char *buffer, int max);
```

DESCRIPTION:

This is a "convenience" function that retrieves the next line of characters from an opened file. Retrieval continues until either 'max-1' characters are loaded or a CR and/or LF is found. If max-1 chars are loaded, the final character placed in the buffer is a NULL. If CR and/or LF is loaded, it is followed by a NULL. The returned buffer will always be NULL terminated.

Parameters:

- int tfd
the descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
a pointer to the space into which TFS is to place the specified number of bytes.
- int max
the max number of bytes to place into the buffer.

Return:

The number of bytes retrieved if successful, else the error returned from tfsread().

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsinit()

API FUNCTION:

mon_tfsinit()

SYNOPSIS:

Initialize the flash space that is used by TFS.

PROTOTYPE:

int mon_tfsinit(void);

DESCRIPTION:

All of the flash space is erased by this function, so be careful!

Return:

Always returns TFS_OKAY.

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfsipmod()

API FUNCTION:

mon_tfsipmod()

SYNOPSIS:

Make a modification to a file directly in the flash space that it currently resides in.

PROTOTYPE:

```
int mon_tfsipmod(char *name, char *buffer, int offset, int size);
```

DESCRIPTION:

This function is unique to TFS. It provides the user with the ability to modify an existing file without deleting the old one then writing a new one. This requires some knowledge of the underlying flash. Refer to the description of the in-place-modify capability within TFS for a detailed discussion.

Note that this method of access does not require the typical open(), read/write, close() calls to modify the flash.

Parameters:

- char *name
name of the file to be modified
- char *buffer
new data to be written to flash.
- int offset
offset into file into which new data is to be written. If this value is -1, then TFS will automatically use the first location that contains 0xff.
- int size
the number of bytes for TFS to write.

Return:

TFS_OKAY if successful, else...

TFSERR_NOFILE

TFSERR_NOTIPMOD

TFSERR_BADARG

TFSERR_WRITEMAX

TFSERR_FLASHFAILURE

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsnext()

API FUNCTION:

mon_tfsnext()

SYNOPSIS:

Go to next header pointer in TFS flash space.

PROTOTYPE:

```
struct tfshdr *mon_tfsnext(struct tfshdr *tfp);
```

DESCRIPTION:

Allows the application to step through the list of files in TFS..

Parameters:

- struct tfshdr *tfp
pointer to header to be used to retrieve the next header. If this pointer is NULL, then the header to the first valid file in TFS is returned.

Return:

A pointer to the header of the next valid file after the header passed as a parameter or (struct tfshdr *)NULL if no more headers.

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfsopen()

API FUNCTION:

mon_tfsopen()

SYNOPSIS:

Open up a TFS file for read and/or write access.

PROTOTYPE:

```
int mon_tfsopen(char *filename, long flagmode, char *buffer);
```

DESCRIPTION:

Similar to a standard open() of a file, this function allows the user to open a TFS file for access. The final buffer argument is needed only for files that are to be created or modified. This is the space that is used by TFS for building the file as multiple mon_tfswrite() calls are made, the data written is placed in this buffer; then when mon_tfsclose() is called to complete the file transaction, the buffer is transferred to flash to become a permanent part of the file system.

Note that the final buffer argument should be NULL if the file is opened for read-only.

Parameters:

- char *filename
name of the file to be read, written or created.
- long flagmode
the flags to be applied to the file when closed and the mode that the file is to be opened in
- char *buffer
a pointer to memory space that will be used by TFS while the file is being generated.

Valid modes:

- TFS_RDONLY: file is assumed to already exist, and it is being opened for reading only.
- TFS_APPEND: file is assumed to already exist, and it is being opened to append to the end of the current file. If the file does not exist, then an error (TFSERR_NOFILE) is returned.
- TFS_CREATE: file is assumed to not exist, and it is being created. If the file does exist, an error (TFSERR_FILEEXISTS) is returned.

In general, only one mode should be specified. An exception to this is TFS_APPEND|TFS_CREATE. If both of these modes are specified, then TFS will modify the mode based on the presence of the file... If the file exists, then it is opened with TFS_APPEND; if the file doesn't exist, it is opened with TFS_CREATE.

Valid flags:

- TFS_EXEC: executable
- TFS_BRUN: to be automatically executed at boot time
- TFS_QRYBRUN: to be automatically executed at boot time, after querying at the console.
- TFS_COFF: loadable executable is COFF format.
- TFS_ELF: loadable executable is ELF format.
- TFS_AOUT: loadable executable is AOUT format.
- TFS_CPRS: file is compressed.
- TFS_UNREAD: file cannot even be read at a user level lower than its own.
- TFS_ULVLN: file is accessible by user level N and above, where N can be 0-3.
- In general, multiple flags are specified for a file. For example...

- TFS_EXEC | TFS_QRYBRUN | TFS_ELF | TFS_ULVL2: is a valid flag specification indicating that the file is executable ELF that will autoboot with query and will only be executable by userlevels greater than or equal to 2.

Return:

Any number greater than or equal to zero (similar, but not to be confused with, a file descriptor) if successful, else...

TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_FILEEXISTS
TFSERR_BADARG
TFSERR_MEMFAIL
TFSERR_NOSLOT

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsread()

API FUNCTION:

mon_tfsread()

SYNOPSIS:

Access a file that has been previously opened for reading and retrieve data from that flash space.

PROTOTYPE:

```
int mon_tfsread(int tfd, char *buffer, int size);
```

DESCRIPTION:

Similar to a standard read() of a file, this function allows the user to retrieve data from a file that has been previously opened.

Parameters:

- int tfd
the descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
a pointer to the space into which TFS is to place the specified number of bytes.
- int size
the number of bytes to place into the buffer.

Return:

The number of bytes retrieved if successful, else...

TFSEERR_BADARG

TFSEERR_BADFD

TFSEERR_EOF

TFSEERR_MEMFAIL

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsrun()

API FUNCTION:

mon_tfsrun()

SYNOPSIS:

Execute a file in TFS.

PROTOTYPE:

```
int mon_tfsrun(char *arglist[], int verbose);
```

DESCRIPTION:

Allows the application to execute a file in TFS flash space.

Parameters:

- char *arglist[]
an argument list (similar to argv[] passed into main()) with a null pointer after the last entry.
- int verbose
verbosity level (0, 1 or 2)...
0: no verbosity
1: print list of arguments for each command in script after tokenization
2: print list of arguments showing value in shell variables.

Return:

TFS_OKAY if successful, else...

TFSERR_NOFILE

TFSERR_USERDENIED

TFSERR_NOTEXEC

TFSERR_BADCRC

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfsseek()

API FUNCTION:

mon_tfsseek()

SYNOPSIS:

Move the internal pointer maintained by TFS to some specified position.

PROTOTYPE:

```
int mon_tfsseek(int tfd, int offset, int whence);
```

DESCRIPTION:

Similar to a standard lseek() of a file, this function allows the user to adjust the current pointer maintained by TFS for the specified file.

Parameters:

- int tfd
descriptor of the file whose pointer is to be adjusted.
- int offset
offset relative to location specified by 'whence'
- int whence
base position from which the offset is assumed.

Valid values for whence:

- TFS_BEGIN: specified offset is relative to the beginning of the file.
- TFS_CURRENT: specified offset is relative to the current position in the file.

Return:

TFS_OKAY if successful, else...
TFSERR_BADARG
TFSERR_EOF

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsstat()

API FUNCTION:

mon_tfsstat()

SYNOPSIS:

Return a TFILE pointer to the file specified.

PROTOTYPE:

```
struct tfs_hdr *mon_tfsstat(char *filename);
```

DESCRIPTION:

Allows the application to retrieve a TFILE pointer (struct tfs_hdr *) to the specified file (if it exists).

Parameters:

- char *filename
name of file in TFS.

Return:

A pointer to the header of the specified file or (struct tfs_hdr *)NULL.

WARNING: if this system call is used in an application that will be automatically defragmenting occasionally, AND the pointer returned by tfsstat() is used for something more than just determining if the file exists, then refer to a warning in the MicroMonitor Modifications page and if possible, use tfsfstat() as a replacement function.

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfstruncate()

API FUNCTION:

mon_tfstruncate()

SYNOPSIS:

Truncate the size of a file that has been opened for append to a new size.

PROTOTYPE:

```
int mon_tfstruncate(int tfd, int size);
```

DESCRIPTION:

If a file is opened for writing (TFS_APPEND flag passed to tfsopen), and the modifications to this file require that the new file size be smaller, then the file size must be truncated. This function provides that capability. In early versions of TFS, when a file was opened for modification, when it was closed it was closed with a size that was dependent on the current position of the write pointer. This was wrong, and has been fixed as of Sept 2000 TFS. Code that originally depended on this TFS bug must now use mon_tfstruncate() prior to doing the call to tfsclose().

Parameters:

- int tfd
the descriptor of the file, returned previously by mon_tfsopen().
- int size
the new, smaller size of the file.

Return:

TFS_OKAY if successful else...
TFSERR_BADARG
TFSERR_BADFD

SOURCE FILE CONTAINING CODE:

tfsapi.c

mon_tfsunlink()

API FUNCTION:

mon_tfsunlink()

SYNOPSIS:

Remove a file from TFS flash space.

PROTOTYPE:

```
int mon_tfsunlink(char *filename);
```

DESCRIPTION:

Allows the application to remove a file from TFS flash space.

Parameters:

- char *filename
name of the file to be removed.

Return:

TFS_OKAY if successful, else...
TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_FLASHFAILURE

SOURCE FILE CONTAINING CODE:

tfs.c

mon_tfwrite()

API FUNCTION:

mon_tfwrite()

SYNOPSIS:

Access a file that has been previously opened for writing and transfer data to TFS for eventual write to flash.

PROTOTYPE:

```
int mon_tfwrite(int tfd, char *buffer, int size);
```

DESCRIPTION:

Similar to a standard write() of a file, this function allows the user to place data into a file that was previously opened for writing.

Parameters:

- int tfd
the descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
a pointer to the space from which TFS is to copy the specified number of bytes.
- int size
the number of bytes for TFS to copy from the bufferr.

Return:

TFS_OKAY if successful else...
TFSERR_BADARG
TFSERR_RDONLY
TFSERR_MEMFAIL

SOURCE FILE CONTAINING CODE:

tfsapi.c