

MICRO-MONITOR

An Embedded System Boot Platform

Getting Started

Revision date: August 18, 2003

MicroMonitor: Getting Started

Table of Contents

This document will step a new user through the early stages of working with an embedded platform running the MicroMonitor¹. It will cover basic system startup, the command line interface, downloading and executing files, and finally, creating basic applications to run as scripts or as executable images. Upon completion of this text, the reader should have a good understanding of how to use the monitor and what it provides. Note that throughout this text, other MicroMonitor documents will be referenced, so it is best to have all of them at your disposal before continuing.

DISCLAIMER AND ACKNOWLEDGEMENTS:	3
CONNECTING POWER TO THE TARGET SYSTEM	4
CONNECTING TO THE TARGET'S SERIAL PORT	4
<i>The Physical Connection</i>	4
<i>The Startup Header</i>	4
GETTING COMFORTABLE WITH THE COMMAND LINE INTERFACE (CLI)	5
<i>Help</i>	5
<i>Command Line Editing and History</i>	6
THE STARTUP ENVIRONMENT	7
<i>APPRAMBASE</i>	7
<i>Downloading To Useable RAM Space</i>	7
CONNECTING TO THE TARGET'S ETHERNET PORT	8
<i>The Physical Connection</i>	8
<i>Establishing the Network Address</i>	8
ADDING A STARTUP SCRIPT AND INTRODUCING TFS	8
<i>Autobootables & monrc</i>	9
TFTP CLIENT AND SERVER	10
BUILDING AN APPLICATION	11
<i>main.c:</i>	11
<i>map.lnk:</i>	13
<i>makefile:</i>	14
VARIOUS DOWNLOAD OPTIONS	16
<i>Download the .elf file to TFS</i>	16
<i>Download the .elf.ezip file to TFS</i>	16
<i>Download the .bin file to RAM</i>	16
RUNNING THE APPLICATION	16
<i>Monitor and application synchronization:</i>	16

¹ The term "Getting Started" here refers to using the monitor, not porting the monitor. If porting information is what you are looking for, then refer to the MicroMonitor App Notes document.

BURNING IN A NEW MONITOR IMAGE..... 17
xmodem -B: 17
newmon: 17

Disclaimer and Acknowledgements:

This document and the MicroMonitor platform discussed are provided "AS-IS". There are absolutely no guarantees of any kind regarding this documentation or the MicroMonitor platform. Lucent Technologies has allowed me (the primary author, Ed Sutter) to make this publicly available from either Lucent's Software Distribution Site (<http://www.bell-labs.com/topic/swdist/>) or through the book "Embedded Systems Firmware Demystified" (available from CMP publishers). Please report comments, questions or deficiencies in this documentation (or the MicroMonitor platform) directly to me at esutter@lucent.com. I will respond to all queries, I just can't guarantee a quick turn-around time!

This monitor package originated by me, Ed Sutter, and in many places both in this text and in the source code there are references to the fact that I was the originator. While it is still reasonable for me claim to have written the bulk of the code, there have been several contributors to the package over the years. A contributor is not necessarily someone who has written code (although this is certainly appreciated!), it can be a user that reports a bug or makes a suggestion on how to improve or add to the package's capabilities and usefulness. So, as you gaze through this text/code, be aware that as the number of contributors increases, putting one name next to the term "author" is a mis-representation of credit. Many thanks to those who have contributed!

Connecting Power to the Target System

This procedure is dependent on your target system. For the CSBXXX² boards, simply plug in the 5-volt supply. Throughout all of this, be aware of static discharge while handling the hardware.

Connecting to the Target's Serial Port

The Physical Connection

Once again, this section is dependent on your target system. All CSBXXX serial ports are configured as a DTE; hence, connection between a computer's serial port and the CSB will require a NULL-Modem. The boards are configured to boot up at 38400 baud, no parity, and 1 stop bit, and in systems that have more than one serial port, usually serial port '0' is the MicroMonitor console port. Upon establishing this physical connection, the board can be reset or powered up and the MicroMonitor header will be displayed shortly after. Note that depending on the CSB board (hereafter referred to as "the target system"), varying degrees of initialization of periphery must take place, so the output of the header may occur a few seconds after reset.

If there is no communication between your computer and the target system after power-up, verify the following...

- Are you physically connected to the COM port you are running your terminal emulator from?
- Are you physically connected to the correct DB-9 port on your target system (some targets have more than one DB9 connector)?
- Is there a NULL modem installed?
- Verify baud rate, parity and stop bit as described above.

If all of these tests are verified and there is no connectivity between your computer and target system, then contact the manufacturer.

The Startup Header

Assuming the above connection has been established, resetting the target should have generated a MicroMonitor startup header that looks something like this...

```
MICRO MONITOR
CPU: PowerPC 405GP
Platform: CSB272
Built: Dec_17,2002 @ 08:02:30
Monitor RAM: 0x03faa0-0x083308
Application RAM Base: 0x300000
MAC: 00:30:23:00:00:01
IP: 192.168.0.214
uMON>
```

The output of most interest here is the RAM memory map, the network address information, and the time/date of the MicroMonitor build. Note, if the board has just been shipped, the IP and MAC addresses are values that may need to be adjusted based on your network (discussed later). For now, we will be working with the serial port only, so the default values are fine³.

At this point, basic connectivity has been confirmed, and you should be able to hit ENTER on your terminal emulator and see the uMON> prompt. This is clear indication that 2-way connectivity over the serial port is alive and well.

² CSBXXX refers to one of several different Cogent Single Board systems (www.cogcomp.com).

³ For a more detailed discussion on the monitor's startup sequence, refer to section "System Startup" of MicroMonitor document "[App Notes](#)".

Getting Comfortable with the Command Line Interface (CLI)

Help

All MicroMonitor commands are white-space delimited, case-sensitive strings of characters followed by ENTER. The first command to get comfortable with is "help". Type "help" followed by ENTER (hereafter the ENTER is assumed). All commands currently configured in the monitor will be displayed in a tabular format similar to the following...

```
Micro-Monitor Command Set:
argv      arp      bbc      call     cast     cm
dhcp      dis      dm      echo     edit     ether
exit      flash    fm      gosub    goto     heap
help      ?        history  icmp     if       item
let       mstat    mt      mtrace   pm       read
reg       reset    return   set      sleep    sm
strace    ulvl     tftp     tfs      unzip    xmodem
version   date     i2c     dcr      lboot    pci
pci_tst   ps2_tst  sc      sed_tst  sed      spr
trap
```

To get a bit more information about each command, type "help -d"⁴. This will list each command with its required user level, plus a brief description of the purpose/usage of the command...

```
Micro-Monitor Command Set:
argv      0 Build argv list
arp       0 Address resolution protocol
call      0 Call embedded function
cast      0 Cast a structure definition across data in memory.
cm        0 Copy Memory
dhcp      0 Issue a DHCP discover
dis       0 Disassemble memory
dm        0 Display Memory
echo      0 Print string to local terminal
edit      0 Edit file or buffer
ether     0 Ethernet interface
exit      0 Exit a script
flash     0 Flash memory operations
fm        0 Fill Memory
gosub     0 Call a subroutine
goto      0 Branch to file tag
heap      0 Display heap statistics.
help     0 Display command set
?         0 Display command set
history   0 Display command history
icmp      0 ICMP interface
if        0 Conditional branching
item      0 Extract an item from a list
let       0 Set shellvar equal to result of expression
mstat     0 Monitor status
mt        0 Memory test
mtrace    0 Configure/Dump memory trace.
pm        0 Put to Memory
read      0 Interactive shellvar entry
reg       0 Display/modify content of monitor's register cache
reset     0 Reset monitor firmware
```

⁴ Note that all commands within MicroMonitor have a Unix-style command line interface: command [options] [arguments]. In this example, "help" is the command and "-d" is an option that tells the command to include the command descriptions.

```

return      0 Return from subroutine
set         0 Shell variable operations
sleep      0 Second or msec delay (not precise)
sm         0 Search Memory
strace     0 Stack trace
ulvl       0 Display or modify current user level.
tftp       0 Trivial file transfer protocol
tfs        0 Tiny File System Interface
unzip      0 Decompress block of memory.
xmodem     0 Xmodem file transfer
version    0 Version information

```

For more details on any single command type “help “ followed by the name of the command⁵. The output will contain at least three lines...

```

uMON>help help
Display command set
Usage: help -[d] [commandname]
Options:
-d    list commands and descriptions

Required user level: 0

```

In the above output of “help help”, the first line is the same abstract that is shown with “help -d” for the specific command. The second line is a summary of the usage of the command on the CLI, and the final line is the required user level to execute the command. The additional lines describing each option will vary from one command to the next.

Command Line Editing and History

There are two possibilities regarding command line editing and history, depending on how the monitor was built: *vt100-arrows* or *readline-vi* style⁶...

VT100-mode: This is certainly the simpler and more intuitive of the two; however, it doesn't have as many features. After having entered the different versions of help above, you can now use the up-arrow key to retrieve them from the command line history. The up arrow steps back through the history and the down arrow steps forward through the history. At any given point in typing at the CLI, the backspace key can be used to undo a character, plus the arrow keys can be used to shift left and right through the text. The CLI will be in “insert” mode; thus, using the left arrow to shift left a few characters, then typing a new character will cause that new character to be inserted into the string. To delete a character, use the arrow keys to move the cursor over that character, then hit the delete key. At any point in this operation, ENTER can be used to cause the entire line to be passed into the monitor's command line processor.

Readline-vi-mode: This mode is a small subset of the vi-mode of command line editing that comes with unix shells. “ESCAPE-k” steps back through cli history, and “ESCAPE-j” steps forward. While typing a command, hit “ESCAPE” to enter edit mode at any time. The following sequences are supported...

```

ESC k    step back through command history
ESC j    step forward through command history
h        move left on line
l        move right on line
0        move to beginning of line
$        move to end of line
x        delete character

```

⁵ For further details on each of the commands, refer to the MicroMonitor document: “Introduction and Command Set”.

⁶ Generally, the CSBXXX boards have a version of MicroMonitor that uses *VT100-arrows* style.

I	begin insert mode at current position
a	begin append mode at current position
R	begin replace mode at current position
r	replace single character at current position
D	delete from current position to end of line
dw	delete from current position to first whitespace
cw	replace word from current position to first whitespace

To determine which mode your monitor is running, simply type ESC-k (escape-key then a 'k'), if the previous command does not appear, then you have VT100 mode.

The Startup Environment

APPRAMBASE

When the monitor first starts up, it configures a few shell variables that can be retrieved by the user to reference various aspects of the target configuration. The first one to note is APPRAMBASE. The address stored in this shell variable is the beginning of RAM memory space that can be used by the application. This is the default address that is used by XMODEM and TFTP when transferring data to the target.

Type the command "set"...

```
uMON>set
      PROMPT = uMON>
      APPRAMBASE = 0x300000
      BOOTROMBASE = 0xffff80000
      PLATFORM = CSB272
      CONSOLEBAUD = 38400
      MONITORBUILT = Feb_04,2003@08:48:20
      ETHERADD = 00:30:23:00:00:01
      IPADD = 192.168.0.214
```

The above list is an example, and your output may be slightly different. Generally speaking, this displays all of the shell variables (and their content) currently established in the MicroMonitor environment. To display any one shell variable, the "echo \$VARNAME" command can be used (where VARNAME is the name of the variable to be displayed).

Use the command "pm" (put memory) to write some data to the memory at \$APPRAMBASE, then the "dm" (display memory) command to verify that it is there... The CLI will convert shell variables to their values, so enter the commands:

```
uMON>pm $APPRAMBASE 0x31 0x32 0x33 0x34
uMON>dm $APPRAMBASE 4
00300000: 31 32 33 34                                1234
```

The pm command writes four bytes to the address stored in the APPRAMBASE shell variable, then the dm command dumps those four values back to the console, in both hex and ASCII. Notice that this is safe because the monitor guarantees that the memory starting at this address is not used by the monitor itself.

Downloading To Useable RAM Space

Now, use Xmodem to transfer a file from host to target, once again using the address stored in \$APPRAMBASE; however, this time XMODEM uses it internally... At the monitor prompt type "xmodem -d". This starts up the target side of the protocol and leaves the target in a mode waiting for the protocol to startup on the host side (depending on the terminal emulator you use, you may see a binary character being sent from the target, this is part of the protocol). At your terminal emulator transfer some small ASCII file using the Xmodem protocol (details here will vary from one terminal emulator package to the next). Upon completion of this transfer, type "dm \$APPRAMBASE" and notice that the content of the file

MicroMonitor
Getting Started

just transferred is being dumped by the dm command. The content of the \$APPRAMBASE shell variable was used by XMODEM to safely transfer a file from the host to memory on the target not used by the monitor itself.

For a more thorough discussion on shell variables used by MicroMonitor, refer to section “Using Shell Variables and Symbols” of MicroMonitor document: [“Introduction and Command Set”](#).

Connecting to the Target’s Ethernet Port

The Physical Connection

There are a few different ways the target can be connected at this point. The simplest would be to do a direct peer-to-peer connection from the target to your host system. This would require that the cable used be a crossover type. Alternatively, if a hub is available, then both the host and target can be cabled into the hub without using a crossover cable (the hub would do the crossover internally). Finally, if a hub is used, then it is best to start off with it disconnected from the rest of the network just until we have a basic understanding of the whole setup.

Establishing the Network Address

For the sake of this discussion, we will assume that the host and target are the only two devices on the network. The host’s network mask is 255.255.255.0 and its IP address is 192.168.0.200 (on the PC, this information can be retrieved with the command “ipconfig” or “winipconfig”). The Ethernet interface of the target system must now be given an address. In the Ethernet/IP world, there are actually two different addresses needed: a MAC address and an IP address. The MAC and IP addresses are stored in the shell variables ETHERADD and IPADD respectively within MicroMonitor’s environment space. Execute the following commands:

```
set ETHERADD 00:11:22:33:44:55
set IPADD 192.168.0.100
set NETMASK 255.255.255.0
ether on
```

The “set” commands establish the environment variables, then “ether on” restarts MicroMonitor’s Ethernet driver⁷. The driver can now properly initialize because it knows its own address⁸. The target’s Ethernet interface is now be active, and will respond to a ping. Issuing the command “ping 192.168.0.100” on the host system should successfully generate a reply from the target. Similarly, at the target console (uMON> prompt), the user can execute

```
icmp echo 192.168.0.200
```

and a response should be received from the host. If at some point the target was connected to a live network with a gateway, then the GIPADD shell variable should be set to the address of that gateway. This allows the target to interact with network devices outside of its own subnet. Now, without resetting the target, immediately read the next section...

Adding a Startup Script and Introducing TFS

Assuming you’ve read the previous sections of this document and followed the steps outlined, then at this point you have a working serial interface and a working network interface. The serial interface comes up initialized automatically; however, the network interface required you to establish a few shell

⁷ You can also try the command “ether -V on” to turn on some verbosity in the driver. This will show you the packets as they are received and transmitted.

⁸ While it is possible to build the monitor with the IP and MAC address burned in, this is usually not the case; hence, the Ethernet interface can’t startup properly until this information is provided. The actual ethernet and IP addresses used will depend on the availability of addresses within your facility.

variables prior to initializing the Ethernet driver. Before you go any further, execute the command “tfs ls”⁹. If this target system is fresh “out of the box”, then it will not have any files stored in flash space and the output will look like this...

```
uMON>tfs ls
Name                               Size  Location  Flags  Info
Total: 0 items listed (0 bytes).
```

It will simply contain the monitor’s executable image (which is not a file). Now execute the command “netcfg” and note that the system tells you that there is no command called netcfg...

```
uMON>netcfg
Command not found: netcfg
```

Next, execute the command “set -f netcfg”, then, once again execute “tfs ls”. Now, the output of “tfs ls” should indicate that there is one file (netcfg) in the TFS storage space (the “set” command with the “-f netcfg” option created the file)...

```
uMON>set -f netcfg
uMON>tfs ls
Name                               Size  Location  Flags  Info
netcfg                             206   0xff00035c e      envsetup
Total: 1 item listed (206 bytes).
```

One last command... execute “set”, and notice that the environment variables previously defined (IPADD, ETHERADD, NETMASK) are still in the environment (along with others).

Now reset the target, and once again execute “set”. Note that IPADD and NETMASK are no longer set to what you previously loaded them with, they are back to their default values. This is because the environment is totally volatile and the contents are re-initialized at startup (wiping out the IPADD, ETHERADD and NETMASK set up earlier). We could just manually re-enter the values; however, now execute “netcfg”, then “set” and notice that the values are back! Cool!

What just happened? After the ETHERADD, IPADD and NETMASK shell variables were established, we ran the “set -f netcfg” command to build an executable script that can be used later to reconstruct the current environment. Now type “tfs cat netcfg” and notice that it contains a bunch of “set” commands. In fact, it contains all the “set” commands that are needed to reconstruct the environment when the “set -f” was executed; hence, our IPADD, ETHERADD and NETMASK shell variables are re-established. Now just type “ether on” and the Ethernet interface is re-initialized with the same environment we established prior to the reset. This is an example of an executable script in TFS.

Autobootables & monrc

This is nice, but after every reset or powerup we still have to manually type “netcfg” and “ether on”. That can quickly become tedious. Using TFS, we can turn the script into an “auto-bootable” script and it will be automatically run at startup. There are three different types of “auto-bootables” within TFS...

- Autoboot with query
This adds a flag to the file so that when the monitor starts up, it queries the user at the console to see if the auto-boot should be aborted. If no interaction is detected within about 2 seconds, the autoboot continues; otherwise it is aborted.
- Autoboot without query

⁹ If this causes an error message that indicates that there is no command called “tfs”, then the version of the monitor must be updated. Contact the manufacturer for information on this update.

This adds a flag to the file so that when the monitor starts up, it autoboots the executable without allowing the user to intervene. This is very useful for an application's security in the field because it does not allow an attacker to terminate the autoboot process; however, note that **this is dangerous during development of an application because "there's no return"**. If the application hangs the target system, you can't abort from this. Recommendation: until the application is quite sane, use the "Autoboot with query" (or just don't autoboot at all). If security is not an issue, then just use "Autoboot with query" all the time.

- Autoboot monrc
This is a special case autobootable (the **monitor's** **run-control** file), that is run prior to the monitor's Ethernet driver being initialized; hence, environment variables established here can be used by the driver initialization to eliminate a second call to "ether on" later in the startup.

Each of the above have their value. The monrc file is typically used to establish the network interface address and perhaps some other very basic environment setup. **It cannot be aborted**; thus, provides some guaranteed environment setup at startup. The ability to query the user is very handy during development because if the executable hangs the system, the query can be used to abort prior to the hang. If the autoboot is without query and it hangs the system, it may require hardware intervention to regain control of the bootup process. So, now let's use TFS to copy our netcfg file to the monrc file so that the network information is in the monitor's environment prior to the Ethernet driver initialization... Execute the command: "tfs cp netcfg monrc", followed by "tfs ls". Notice the new file called "monrc" now in TFS storage space.

Once again reset the target. Notice this time that the header dumped by the monitor at startup will contain the MAC and IP address that we stored in the ETHERADD and IPADD shell variables earlier. Also, if you ping the target now, you will see that the Ethernet interface is up and running. Since the environment setup is now in monrc, the Ethernet driver automatically comes up on its own.

TFTP Client and Server

MicroMonitor contains a TFTP client and server. In a typical configuration, the TFTP server is started up as soon as the Ethernet interface is started up. To verify this, use a tftp client on your host machine to transfer a small file to the target. For this example, assume the file on the host system is called "host_file" and we want to copy it to the target system, but with the name "target_file". If the "tftp" tool is used (comes with the monitor package and runs on windows and unix), the command line would be...

```
tftp 192.168.0.100 put host_file target_file
```

Yes, there are two t's in "tftp" above (don't ask, just use it!) At the target's console, you will see some acknowledgement of the transfer, and then "tfs ls" will show the presence of the file now in TFS flash space. Resetting the board does not remove the file because it was copied into TFS¹⁰. Similarly, the monitor's TFTP server can be used to retrieve files as well. The command...

```
tftp 192.168.0.100 get target_file copy_of_target_file
```

will pull the file just copied to flash back up to the host and give it the name "copy_of_target_file". At this point on the host system, the files "host_file" and "copy_of_target_file" should be identical.

The server also allows files to be transferred directly to/from RAM without any interaction with TFS. In the case of a transfer to ram, the server looks at the destination file name. If the name begins with "0x", then it will download the file from the host to the hex address specified after the leading 0x. For transfers of memory from target to host, the server looks at the source file name. It should be of the

¹⁰ Note that this copy is actually indirect. The file content is first transferred to memory starting at the address specified in \$APPRAMBASE, then once the transfer is complete, the data is burned into flash from RAM.

syntax 0xHHHH,LLLL where HHHH is the hex address and the LLLL after the comma is the length of the block to be transferred to the host. Note that this must be void of whitespace. For example..

```
tftp 192.168.0.100 put host_file 0x310000
```

transfers the file "host_file" to RAM at 0x310000, and

```
tftp 192.168.0.100 get 0x310000,400 target_file
```

transfers 400 bytes starting at 0x310000 to the host file target_file.

Recall earlier that we spoke of "autobootable" and "executable" files. These are attributes (or flags) associated with the file and if a file is to be transferred to TFS and is to contain one (or more) of these attributes¹¹, there must be some means of conveying that information in the download process. To deal with this, each flag is assigned a single-character (for example 'e' indicates executable script and 'B' indicates autobootable with query), and the complete filename is constructed with up to 2 comma delimiters. There are actually two additional fields that can be specified along with the filename: the flags and an optional information field. So, to transfer a file named "myprog" with the flags 'e' and 'B', and the information field set to "sys_init", the syntax would be: "myprog,eB,sys_init", or, within in a tftp command...

```
tftp 192.168.0.100 put host_file myprog,eB,sys_init
```

Finally, and this is a very important point, when using the TFTP protocol the data transfers can be in ASCII or binary. The "tftp" client that comes with the monitor defaults to binary; however, if some other tftp client is used, you need to **be aware of whether or not it defaults to ASCII or binary**. In ASCII mode, it does the translation that has been a nuisance for decades between UNIX and Windows text files, the adjustment of carriage-return (0x0d) and line-feed (0x0a). Anyway, if the file being downloaded is an executable image, then you certainly do not want the image to be modified in any way, so make sure in this case that your client is requesting a binary transfer.

Building an application

This section assumes the toolset is MicroCross X-Tools. A fully functional beta version of these tools is available on the CD of the book "[Embedded Systems Firmware Demystified](#)", or an up-to-date version can be purchased from [Microcross.com](#). The tools are standard GNU-GCC, so regardless of the vendor, the following text is fairly applicable.

In addition to the GNU-GCC tools, there are a few tools that are packaged with MicroMonitor. Some of these can be replaced with equivalent GNU-tools, and some cannot¹². There are basically three files needed to build a simple application: main.c, map.lnk & makefile, each of which will be described below.

main.c:

This is the MicroMonitor application equivalent of "hello world", consisting of two functions: start and main. These two chunks of code are separated into two functions for clarity. If you really wanted to, you could just do it with one function. The start() function is the entrypoint into the application. It is within this function that the application prepares itself for main(). In its simplest form, it attaches to the monitor, retrieves command line arguments and calls main in the familiar *main(int argc, char *argv[])*.

```
#include "monlib.h"
```

¹¹ For a much more detailed discussion on these flags and TFS in general, refer to section "TFS: Tiny File System" in the MicroMonitor document: "[Application Notes](#)"

¹² Complete documentation on the tools supplied with MicroMonitor can be found in the MicroMonitor document: "[Host Based Tools](#)".

```

#define MONCOMPTR (unsigned long *)XXXXXX

extern unsigned bss_start, bss_end;

int
start()
{
    int      argc;
    char     **argv;
    volatile register uchar *ramstart, *ramend;

    /* Clear bss.  TFS also does this, but we may not be starting
     * through TFS.
     */
    ramstart = (unsigned char *)&bss_start;
    ramend   = (unsigned char *)&bss_end;
    while(ramstart < ramend)
        *ramstart++ = 0;

    /* Connect application to monitor.  This must be done prior to
     * the application making any other attempts to use the "mon_"
     * functions provided by the monitor.
     */
    monConnect((int(*)())(MONCOMPTR),(void *)0,(void *)0);

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

    /* Call main, then return to monitor.
     */
    return(main(argc,argv));
}

int
main(int argc,char *argv[])
{
    mon_printf("hello embedded world\n");
    return(0);
}

```

This is a generic start() function for simple MicroMonitor based applications. It assumes that application is running off of the stack that the monitor has, and does not deal with any C++ constructor/destructor issues. The value of MONCOMPTR is target specific and can be determined by issuing the "mstat" command at the monitor. Among other things, the mstat output will contain a line that tells you where that installed monitor has its internal moncom() function pointer.

```

uMON>mstat
CPU: PowerPC STB04xxx
Platform: mPhase Set-Top-Box
Built: Dec_17,2002 @ 08:02:30
Monitor RAM: 0x03faa0-0x083308
Application RAM Base: 0x300000
MAC: 00:60:1d:02:0b:f4
IP: 135.3.94.136
Ptrs: moncom: 0x8010, etheradd: 0xffffa0000, ipadd: 0xffffa0020
App-assigned functions...
    getchar: 0x0
    putchar: 0x0

```

```

dcacheFlush: 0x0
icacheInvalidate: 0x0
uMON>

```

Note the highlighted text above. This is the entry that must be put into the MONCOMPTR definition, so our start() function will contain...

```
#define MONCOMPTR 0x8010
```

For specific details on what this hookup is and how it works, refer to section the MicroMonitor document: "Application Programmer's Interface".

The main() function is almost identical to hello world main, but with a few MicroMonitor hooks. Notice that the mon_printf() call is a function that is not part of this source code. This is a function that is part of the monitor's API, and is connected to the application as a result of the call to monConnect() in start(). Once again, refer to other MicroMonitor documentation for further details on this interaction.

map.lnk:

Before the program can be built and placed on the target, it must be configured into some absolute memory space that is legal space within the memory map of the MicroMonitor platform. This is done with a link editor file. The following example is typical, but exact details may vary depending on processor and toolset. The important thing to note is that this file specifies the location in RAM space at which the application will reside at runtime.

```

MEMORY
{
    dram :      org = 0x300000,    len = 0xfffff
}

SECTIONS
{
    .text :
    {
        *(.text)
    } >dram

    .data :
    {
        *(.data)
    } >dram

    .sdata :
    {
        *(.sdata)
    } >dram

    .sdata2 :
    {
        *(.sdata2)
    } >dram

    .rodata :
    {
        *(.rodata)
    } >dram

    .got :

```

```

    {
        *(.got)
    } >dram

.bss :
{
    bss_start = .;
    *(.bss) *(COMMON)
} >dram

.sbss :
{
    *(.sbss)
    bss_end = .;
} >dram
}

```

It is beyond the scope of this document to cover all the details of a link-editor file. The above file contains two main parts: MEMORY and SECTION declaration areas. The MEMORY declaration specifies one block of memory called dram (if you want to call it "bill" you can do that). Our entire application will reside in that memory block. The important thing for us is that the address of that memory block must be coordinated with the MicroMonitor platform on which it will reside. Recall our earlier discussion on the APPRAMBASE shell variable. The content of that variable is the address that can be used for the base of the memory block. The SECTION declarations are dependent on the toolset, with the only significant point to mention being the `bss_start` and `bss_end` tags inserted before (`bss_start`) the start of `.bss` section and just after the end (`bss_end`) of the `bss` sections. These tags can be used to mark the ram space that can be initialized to zero at application startup.

makefile:

This makefile simply compiles `main.c` and `monlib.c`, then links it to the absolute memory map specified by `map.lnk`. It shows the use of the `elf` tool to generate a simple dump of the memory map, plus the creation of a `.bin` file that is the equivalent of the image that would reside in RAM after TFS transfers it to its executable memory space. This makefile is run under the BASH shell on the PC (comes with Microcross X-Tools package). It assumes that the monitor's common code directory is `../../common/monitor`, that the "elf" and "monsym" tools (comes with monitor package) are in the system's PATH, and that the "obj" directory has been created. Changes can be made as needed...

```

#####
#
# Makefile for building a small application on MicroMonitor
#
MONITOR      = ../../common/monitor
TARGET       = ppc-elf
include      $(MONITOR)/tools.make
ELF          = C:/bin/elf

CFLAGS       = -c -Wall -O -fno-for-scope -mcpu=405 \
               -mno-sdata -msoft-float -nostdinc -fno-builtin -g -o $@ \
               -I. -I $(MONITOR)

PROG         = app
LIBS         = $(LIBGCC)

OBJS= obj/main.o obj/monlib.o

app.elf: $(OBJS)
$(LD) -e start -o $(PROG).elf -T app.lnk $(OBJS) $(LIBS)
$(NM) --numeric-sort $(PROG).elf | sort -u > $(PROG).sym
$(STRIP) $(PROG).elf

```

```

$(ELF) -m $(PROG).elf
$(ELF) -z6 $(PROG).elf
$(ELF) -B $(PROG).bin $(PROG).elf

symtbl:
    monsym -Sx -P0x -d1 -s3 $(PROG).sym >symtbl

obj/main.o: main.c
    $(CC) $(CFLAGS) main.c

obj/monlib.o: $(MONITOR)/monlib.c
    $(CC) $(CFLAGS) $(MONITOR)/monlib.c

dldzelf:
    ttftp $(TARGET_IP) put $(PROG).elf.ezip appz,Ec

dldelf:
    ttftp $(TARGET_IP) put $(PROG).elf app,E

dldbin:
    ttftp $(TARGET_IP) put $(PROG).bin 0x300000

dldsymb:
    ttftp $(TARGET_IP) put symtbl

clobber:
    rm -f $(PROG).elf $(PROG).elf.ezip $(PROG).sym symtbl
    rm -f obj/*.o

```

Notice the compilation of the monlib.c source code. This is a file that comes with the monitor and simply provides the hook up between the monitor and the application program. Also, the use of tools.make. Browse the tools.make file (under common/monitor) and note that this builds the NM, CC, LD macros (and others). This follows some suggestions that I received from the folks at Microcross so that my makefiles could be independent of the target platform (CC instead of ppc-elf-gcc for example).

This makefile produces several output files (aside from just the obj/*.o files)...

- **app.elf**
This is the basic ELF image. It can be downloaded directly into TFS and the loader in TFS will know how to transfer it from its ELF format in flash to the raw binary format in DRAM where it will be executed. It is transferred to the target system via “make dldelf”.
- **app.elf.ezip**
This is the compressed ELF image. It too can be downloaded directly into TFS and the loader in TFS will know how to decompress and transfer it from its ELF format in flash to the raw binary format in DRAM where it will be executed. It is transferred to the target system via “make dldzelf”.
- **app.bin**
This is a file that is the equivalent of the image that TFS’s loader would create in DRAM. It can be transferred directly to DRAM to quickly load the image into the memory space within the system that it will execute out of. In this case, TFS is out of the picture, so the user must issue the “call 0xENTRYPOINT” command (where ENTRYPOINT is the hex address of the start() function). It is transferred to the target using “make dldbin”. In this example the entry point would be 0x300000.
- **app.sym**
This file contains a list of all the symbols in the application. It is built by the GNU tools.
- **symtbl**
This file is built from the app.sym file by the monsym tool. This is simply a “squeezed” version of the symbol table, containing only the information that the monitor needs for basic symbol processing. It

can be installed onto the target to allow the monitor to access variables symbolically. This file is transferred to the target using “make dldsym”.

Various Download Options

The above section has already eluded to the fact that there are several different ways in which the application can be put into the system. These range from a basic XMODEM file or data transfer all the way up to a network boot using DHCP (or BOOTP) and TFTP. This section will discuss the basic three most commonly used during development.

Download the .elf file to TFS

Use the “dldelf” target in the makefile described above. This will install the app.elf file into TFS as “app”. After the file is transferred, at the uMON> prompt type “tfs ls” to see that the file “app” is installed with the flags “E”. The file is safe in flash. The board can be reset and the file will still be accessible.

Download the .elf.ezip file to TFS

Use the “dldzelf” target in the makefile described above. This will install the app.elf.ezip file into TFS as “app”. After the file is transferred, at the uMON> prompt type “tfs ls” to see that the file “app” is installed with the flags “Ec”. Once again, the file is safe in flash. The board can be reset and the file will still be accessible.

Download the .bin file to RAM

Use the “dldbin” target in the makefile described above to save on wear and tear of your flash. This will install the app.bin file directly into ram. The address into which it is loaded is the base address of the image as it was specified in the linker map file. For this example, that address is 0x300000. In this case, there is no TFS interaction, the image is copied directly to RAM via TFTP. Since it was not copied to TFS, a reset or power-cycle will corrupt the content of RAM; hence the download will have to be repeated.

Running the Application

At this point (assuming you’ve been reading through this document), you’ve built and downloaded the application. Now all that’s left to do is run it. This too can be done a few different ways, dependent on how you downloaded the file (described above). If the file is either .elf (file attribute “E”) or .elf.ezip (file attribute “Ec”), then the name of the application can simply be typed on the command line and TFS will automatically load it and jump to the entrypoint. If the file is a raw binary that has been directly transferred to DRAM, then the “call” command can be used to jump directly into the entrypoint of the application. Then the argument to the “call” command is the entrypoint of the image. This is the address of the “start” function (argument to –e in the LD line of the makefile above). This address is usually the same as the based address into which the image is loaded, but it doesn’t have to be, so the command “elf –f app” can be used to retrieve the entrypoint of the application, and that should be used as the argument to “call”.

Monitor and application synchronization:

In either case, if the application is going to be hooked up to the monitor (using the monConnect() function discussed earlier) then the user must be aware of the fact that the on-board monitor must be in sync with the monlib.h header file that is used by the application to connect the two. There is no actual linkage between the two; as discussed above, the only address that is shared by the application and the monitor is the address parameter used with monConnect(). All other connections are achieved based on a set of macros in monlib.h. Occasionally new features are added to the monitor, so new macros are added to the monlib.h file. If the on-board monitor is not updated, and the application uses this new monlib.h header file, then at the time of monConnect() a warning will be posted by the monitor to indicate that some of the newer features that the application attempted to connect to are not available. The warning will look something like this...

```
moncom unknown command: 0x32
```

MicroMonitor
Getting Started

where the value '0x32' is the macro in monlib.h that is newer than the on-board monitor build. Note that this does not necessarily mean that the application will fail, it simply means that those facilities will fail. If the application is not using the particular facility, then the warning can be ignored. It is obviously best; however, to keep the on-board monitor up-to-date with the application.

Burning in a new Monitor Image

The monitor allows you to install a new monitor over top of the one that is currently running. This convenience is both good and bad. Good because it allows you to update without the need of any kind of JTAG-like attachment, bad because **if you get it wrong, you destroy the boot image of your target system** and then you DO need a JTAG-like attachment to restore the boot image. The point here is that if you aren't sure about what you're doing, and you do not have the capability to restore your boot image with some external tools (JTAG, etc...) then don't just do this to "experiment" with your target.

There are two ways to do this update. One uses the serial port and the other uses ethernet...

xmodem -B:

The xmodem command allows the user to download images to memory. The -B option to xmodem tells it to do all the steps necessary to transfer the downloaded image into the portion of flash that contains the boot monitor. So, after building a binary image that is considered to be a new monitor image, it is simply transferred to the target using xmodem -B. Upon completion of the transfer, and prior to doing the actual boot-image update, xmodem -B will query the user one last time...

```
Reprogramming boot @ 0xB BBB from 0xA AAA, SSSS bytes.  
OK?
```

Translated... 'SSSS' bytes will be copied from ram space beginning at 0xA AAA to boot flash space beginning at 'B BBB. Respond to the query with 'y' to continue ONLY if this information is correct; else abort with any other character.

newmon:

The tool "newmon" can be used to burn a new monitor image. Depending on the platform on which it is run, newmon is either a script (unix) or a single executable (windows). In either case, newmon does two things (similar to xmodem -B, just faster!)..

```
Download the binary image  
Transfer the binary image to the boot space.
```

Both versions (unix script & windows .exe) take two arguments: the IP address of the target and the file containing the image destined for the boot flash. The windows version includes some additional options that generally are not needed. For more information on the newmon.exe tool, refer to the MicroMonitor document: Host-Based Tools for information on this tool. If you're on unix, then just read through the newmon script.

Filename: getting_started.doc
Directory: H:\umon_stuff\docs>manuals\doc
Template: C:\Program Files\Microsoft Office\Templates\els_template.dot
Title: The Monitor Revisited
Subject: The use of a boot monitor in embedded systems
Author: Ed Sutter
Keywords:
Comments:
Creation Date: 11/02/02 12:37 PM
Change Number: 69
Last Saved On: 08/18/03 12:36 PM
Last Saved By: Ed Sutter
Total Editing Time: 560 Minutes
Last Printed On: 08/18/03 12:36 PM
As of Last Complete Printing
Number of Pages: 17
Number of Words: 6,170 (approx.)
Number of Characters: 35,169 (approx.)